QOVIS: Understanding and Diagnosing Query Optimizer via a Visualization-assisted Approach

Zhengxin You
The Hong Kong Polytechnic University
Southern University of Science and Technology
Hong Kong, China
zhengxin.you@connect.polyu.hk

Qiaomu Shen✉
Beijing Institute of Technology, Zhuhai
Zhuhai, China
joyshen06@gmail.com

Man Lung Yiu
The Hong Kong Polytechnic University
Hong Kong, China
csmlyiu@comp.polyu.edu.hk

Bo Tang✉
Southern University of Science and Technology
Shenzhen, China
tangb3@sustech.edu.cn

ABSTRACT

Understanding and diagnosing query optimizers is crucial to guarantee the correctness and efficiency of query processing in database systems. However, achieving this is non-trivial as there are three technical challenges: (i) hundreds and thousands of query plans are generated for each query during the query optimization procedure; (ii) the transformation logic among query plans is not easy to investigate even for expert database system developers; and (iii) navigating users to the root causes of the bugs/errors is inherently hard as the changes of the operators among query plans are missing in the query processing log. In this work, we propose QOVIS to overcome these challenges, which identifies the query optimization bugs/issues and investigates their root causes via a visualization-assisted approach. Specifically, QOVIS consists of data preprocessing layer, transformation logic computation layer, and visual analysis layer. We conduct extensive experimental studies (e.g., user study, case study, and performance study) to evaluate the efficiency and effectiveness of QOVIS. In particular, our user study (on 24 database developers and researchers) confirms that QOVIS significantly reduces the time required to investigate the bugs/errors in the query optimizer. Moreover, the generality of QOVIS is verified by utilizing it to understand and diagnose the real-world reported bugs/errors in different query optimizers of three widely-used systems: Apache Spark, Apache Hive, and DuckDB.

PVLDB Reference Format:
Zhengxin You, Qiaomu Shen, Man Lung Yiu, and Bo Tang. QOVIS: Understanding and Diagnosing Query Optimizer via a Visualization-assisted Approach. PVLDB, 18(6): 1677 - 1690, 2025.
doi:10.14778/3725688.3725698

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBGroup-SUSTech/QOVIS.

✉ Corresponding authors: Prof. Bo Tang and Prof. Qiaomu Shen.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725698

1 INTRODUCTION

Big data and database systems have been widely used in various applications to process huge amounts of data, e.g., e-commerce shopping, and business intelligence reports. Query optimizer is one of the most important components in these systems. It transforms the user SQL query into an efficient physical execution plan [40]. Many research studies [40, 42, 62] have been proposed to improve the efficiency of the final generated execution plan of the query optimizer. However, the study on the *correctness* and *efficiency* of the query optimizer is not sufficient. In particular, the correctness of the query optimizer guarantees that the generated execution plan is semantically equivalent to the user input SQL query, and the efficiency of the query optimizer measures the time cost to transform a user SQL query to a physical execution plan.

Recently, Tang et al. [59] discovered logic bugs in relational database management systems (RDBMS) through automated testing methods. However, it can only answer "*whether*" the query optimizer works correctly, but cannot answer "*how*" the query optimizer works and "*where*" the optimization bug/issue occurs. Understanding and diagnosing query optimizers is the key to answering these two questions. Unfortunately, the query optimizer is inherently complex and its optimization procedure is very complicated even for database experts and system developers. The key reasons are twofold: (i) the query optimization procedure is intricate because it involves hundreds or even thousands of interrelated optimization strategies; and (ii) it lacks effective tools to facilitate users in analyzing tedious optimization logs from these systems.

In this paper, we propose QOVIS to understand and diagnose query optimizers via a visualization-assisted approach. The technical challenges to build QOVIS are three dimensions. First, the optimization log of a user-input SQL query always includes hundreds or thousands of intermediate query plans. Thus, it is difficult to provide an overview of the optimization procedure. Second, the transformation logic among the query plans is not easy to identify even with the expertise of the database system. To make matters worse, optimization bugs/errors probably occur across multiple query plans. Third, navigating users to explore the root causes of bugs/errors is inherently difficult as it needs to investigate the relationship of the changed operators among different query plans.

To overcome these challenges, we first investigated 57 optimization bugs/issues that are reported on the issue tracker websites [2, 4, 6] of three well-known systems: Apache Spark [17], Apache Hive [61], and DuckDB [49], in this work. We then classified these reported issues into two categories: (i) *transformation error*, which produces incorrect query plans due to the improper/incorrect implementation of the query optimization strategy; and (ii) *workflow error*, which affects the efficiency of the query optimizer (e.g., applying redundant optimization strategies). As we will elaborate shortly, existing work cannot automatically identify both types of errors in query optimizers.

To address them, we propose QOVIS to understand and diagnose the query optimizer via a visualization-assisted approach. In particular, it consists of three layers: *data preprocessing layer*, *transformation logic computation layer*, and *visual analysis layer*. The collected query optimization logs are processed into the optimization trace, which includes a query plan sequence and an optimization step sequence in the *data preprocessing layer*. To ease the understanding of optimization processes, the transformation logic among plans are computed in the *transformation logic computation layer* to reduce the manual effort for query optimization procedure understanding and diagnosing. An exhaustive search of the optimization rule sequence, which transforms one query plan to another, is obviously impractical. We adopt the A^* algorithm by devising novel estimation functions to improve the performance of the plan transformation problem. In the *visual analysis* layer, QOVIS offers a suite of carefully designed visualization views (i.e., trace hierarchy view, plan view, and transformation logic view) to facilitate users in identifying and investigating the above bugs/errors.

QOVIS is built on Apache Spark [17] as it is widely used in many leading IT companies (e.g., Google, Meta) for various applications, and many users frequently report the bugs of Apache Spark on its issue tracker website [4] during daily usage. To verify the generality of QOVIS, we extend it to two other well-known systems: Apache Hive [61] and DuckDB [49]. In particular, Apache Hive adopts Apache Calcite [19] as query optimizer, which is a volcano/cascade style optimizer and has been used in many systems, e.g., Apache Drill [36] and Apache Flink [22]. DuckDB is a popular lightweight database for analytical queries, its query optimizer is different from the Catalyst in Apache Spark. It is worth pointing out that QOVIS can be easily extended to other database systems as it only uses the query plans and optimization steps in the query optimization logs, and the source code of QOVIS is available at [9].

Although understanding and diagnosing query optimizers by utilizing QOVIS might be a little difficult at first. Compared to using limited information provided by the `EXPLAIN` command of database systems, or reading long and tedious textual logs, the visualization-assisted approach in QOVIS provides a well-organized and vivid representation of the optimization process. It will ease the analyzing procedure significantly as long as users become familiar with the visual encoding and interactions. We will explicitly verify it shortly.

Contributions. The contributions are summarized as follows:

- We analyze the optimization issues in three popular database systems and classify them into two categories by considering the correctness and efficiency of the underlying query optimizers: *transformation error* and *workflow error* (see Section 2).

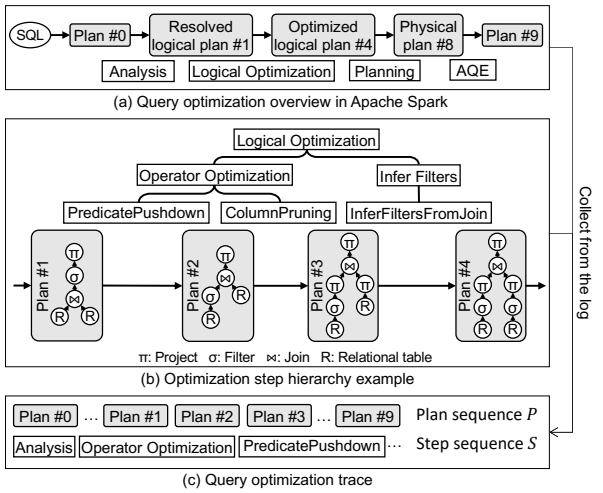


Figure 1: Query optimization procedure illustration

- We propose QOVIS to understand and diagnose query optimizers via a visualization-assisted approach, which facilitates the database system developers to analyze optimization trace visually at a fine-grained level (see Section 3).
- We devise plan transformation algorithms and novel visualization designs to support efficient and effective visual analysis of the optimization trace (see Section 4 and Section 5).
- We conduct extensive experimental studies (including case study, user study and performance study) on real queries to demonstrate the superiority of QOVIS (see Section 6).

2 PROBLEM DESCRIPTION

In this section, we first introduce the basic concepts of query optimizer in Section 2.1, then discuss two key issues of the query optimizer during query processing in Section 2.2.

2.1 Query Optimizer

The query optimizers in the database systems transform the user input SQL query to an optimized query plan. We next use the query optimizer, *Catalyst* [17], in Apache Spark to introduce the fundamental concepts in the query optimizers.

Figure 1(a) depicts the workflow of the query optimizer in Apache Spark 3.0.0. A given input SQL query is parsed into an initial plan, i.e., Plan #0. The optimization procedure consists of four sequential major optimization steps: *Analysis*, *Logical Optimization*, *Planning*, and *Adaptive Query Execution* (AQE). In each step, the optimizer transforms the input plan into an equivalent but more efficient one by applying pre-defined optimization strategies. As shown in Figure 1(a), the initial Plan #0 is transformed into Plan #1 to Plan #9 during the query optimization procedure. We next formally define the query plan and optimization step as follows.

Definition 1 (Query plan). *The query plan p is an ordered tree, where each leaf node is a relational table, each intermediate node is an SQL operator with a list of arguments, and each edge between two nodes shows the data dependency of the connected nodes.*

Example. Plans #1 to #4 in Figure 1(b) are the query plan examples. We use R to represent the leaf relational table in the query plan.

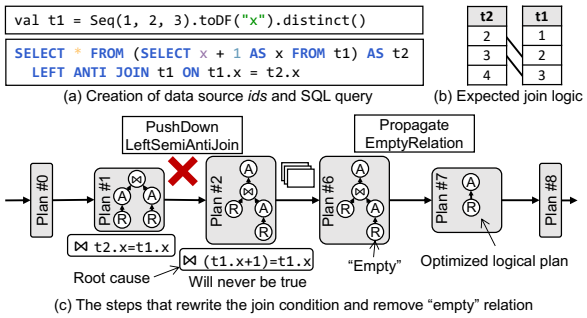


Figure 2: The illustration of *transformation error*

Definition 2 (Optimization step). *In the query optimizer, the optimization step s transforms the input query plan p to a more efficient query plan p′ by applying pre-defined optimization strategies.*

Example. Figure 1(b) shows examples of optimization steps, e.g., Plan #1 is transformed into Plan #4 via the optimization step *Logical Optimization*. Interestingly, the granularity of optimization steps in the optimizer varies, and they can be nested. For example, the step *Logical Optimization* includes two sub-steps: *Operator Optimization* and *Infer Filters*. Within *Operator Optimization* step, it consists of two optimization sub-steps *PredicatePushdown* and *ColumnPruning*, as the exemplified optimization step hierarchy shown in Figure 1(b).

In almost all (if not all) database systems, the system log intermingles optimization plans and optimization steps during query processing. For example, database developers should pre-process query execution logs of Apache Spark to identify the specific query plan or optimization step for bug fixing or performance improvement. In this work, we refer to the pre-processed execution log in these systems as *optimization trace*, which consists of (i) a sequence of plans and (ii) a sequence of steps, as shown in Figure 1(c).

2.2 Query Optimization Issues

To provide a comprehensive study of query optimization issues, we investigate 57 issues [13], which are reported on their issue tracker websites [2, 4, 6]. We classified them into two categories: (i) *transformation error* and (ii) *workflow error*, which correspond to the correctness and efficiency of the underlying query optimizer, respectively. In particular, *transformation errors* are the issues that affect the correctness of the optimizer, and *workflow errors* are the issues that influence the efficiency of the optimizer. We next elaborate on the details of the above two categories.

2.2.1 Transformation error. It is the case that the optimizer transforms the input plan p to an optimized plan p' incorrectly, thus p' is not semantically equivalent to p ($p \neq p'$). In general, p' is referred to as a problematic plan as it (i) is invalid, (ii) returns incorrect results, or (iii) takes an unexpectedly long execution time. Obviously, the *transformation error* shows the correctness issues of the optimizer during the query processing as the optimized plan p' is not equivalent to the input query plan p.

Example 1 (Incorrect plan transformation). Figure 2(a) shows a reported optimization issue [10]. In particular, the SQL query self-joins a table `t1` on the column `x`. Figure 2(b) shows the correct join logic with table `t1`. The expected result is `[4]` as the left anti-join only keeps the rows in table `t2` that do not occur in table `t1`. However, the returned result of Apache Spark 3.3.0 is `[2,3,4]`. The incorrect result is caused by the *transformation error* as the optimized logical plan (Plan #7) in Figure 2(c) is not equivalent to the initial plan (Plan #0), i.e., the join operator is removed unexpectedly. By tracking the optimization trace, we find a problematic predicate `(t1.x+1)=t1.x` results in an "Empty" relation, which is caused by the *PushDownLeftSemiAntiJoin* step, see Figure 2(c).

It is not trivial to identify the *transformation error* during the query processing. Existing commands, e.g., `EXPLAIN`, of the query optimizer only provide the optimized logical plan and do not detail the optimization steps. Hence, the system developers always have to verify the system log manually to pinpoint the optimization step that results in the problematic plan. Obviously, it is a time-consuming task. To make matters worse, it is hard to identify the root cause of the *transformation error* even if the system developers find the corresponding optimization step in the system log with their expertise. The reason is that the optimization step in the log does not show the changes of the query plans.

In the literature, verifying the equivalence of SQL queries is studied [24, 25, 30, 68, 69]. However, they cannot be adapted to find the *transformation error* in the query optimizer as (1) verifying the equivalence of query plans, i.e., p and p', is different from them as the query plans include physical operators and execution algorithms; and (2) there are hundreds or even thousands of intermediate query plans for the input SQL query, so it is almost impossible to verify the equivalence among all of them using the above expensive algorithms. In addition, automatic logic bug detection tools [27, 50, 59, 60] are proposed to identify the logic bugs during the query processing. Their core idea is to compare the execution results of the optimized query plan with the ground truth. First, these tools cannot be used to identify the *transformation error* which does not cause incorrect results. Second, many of their detected errors are in the query executor, thus, it is not easy to identify the *transformation errors* in query optimizers. Last but not least, almost all of these tools do not provide insights to pinpoint the root cause of the logical bug.

In this work, we propose QOVIS, which assists database system developers to investigate the transformation logic of the optimizer for these *transformation errors* in an explicit manner. For example, with the help of QOVIS, it is easy to identify that the *PushDownLeftSemiAntiJoin* optimization step caused the *transformation error*, as the red cross in Figure 2(c) shown. Moreover, it provides clues for developers to conclude that the above step fails to process the predicate that contains attributes from the same relational table (`t1` in this case). Consequently, it is safe to conclude that the optimization step may not be implemented correctly to process the above corner case and it sheds light to fix this bug in the query optimizer.

2.2.2 Workflow error. It is the case that the optimizer transforms the input plan p to an optimized plan p' correctly, i.e., the query plans p and p' are equivalent both semantically and practically ($p = p'$). However, the optimization steps from p to p' may result in (i) redundant optimizations, (ii) sub-optimal query plan, or (iii) even crashes during the optimization process. Even though optimizers with volcano/cascade architecture use memorization techniques to detect redundant query plans, the workflow error cannot be avoided in them, as the case study in Apache Hive shows (Section 6.1.3).

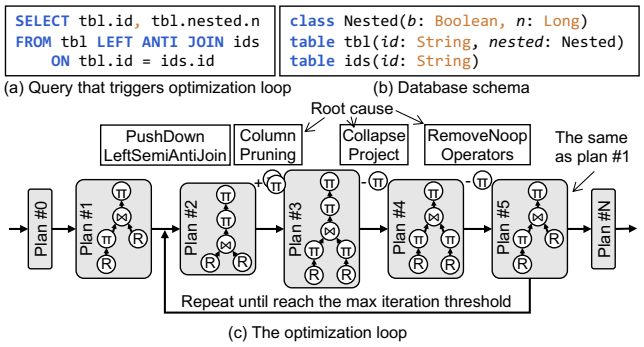


Figure 3: The illustration of *workflow error*

Example 2 (Loop optimization). Figure 3 illustrates an example of *workflow error*, which is reported on *Apache Spark issue tracker website* [11]. As shown in Figure 3(a), the SQL query joins two tables `tbl` and `ids`. Figure 3(b) shows the schema of the database, where table `tbl` has two columns: `id` and `nested`. When Apache Spark 3.3.0 optimizes this query, the optimizer keeps re-optimizing the plan with the same steps but leaves the plan unchanged, see Figure 3(c). As reported by the developers, these steps "go back and forth, undoing each other's work" until the max iteration threshold (i.e., 100) is reached. Obviously, this process includes many redundant optimization steps and wastes a lot of time on them.

Optimization issues caused by the *workflow errors* is even more difficult to identify than *transformation errors* as they return correct results unless it is crashed during the query processing. Logically, pinpointing the *workflow errors* in the query optimizer needs to analyze the whole query optimization log. However, it is almost impossible to achieve by checking the plain text in the system log manually as the number of steps is daunting. For example, it includes more than 1200 optimization steps for the query in Figure 3. In addition, to pinpoint these workflow errors, system developers need to analyze multiple optimization steps simultaneously and track the changes of operators/arguments across different intermediate query plans, e.g., the looped plan sequence in Figure 3(c).

Unfortunately, existing tools [16, 34, 47, 48, 56, 58, 63] cannot be used to investigate the large amount of optimization steps and associated intermediate query plans. In particular, Picasso [34] and pg4n [56] provide diagrammatic or textual information of final generated plans but do not include the optimization steps in optimizers. Existing visualization tools [16, 47, 48, 58, 63] focus on the correctness and execution performance of the final plan and only provide information about coarse-grained steps (e.g., *Analysis* and *Logical Optimization*). To sum up, they do not support the systematic analysis of the numerous fine-grained optimization steps. Hence, they cannot be adapted to identify the *workflow errors* in optimizers.

In this work, we devise a visualization-assisted approach in QOVIS, which helps database system developers to identify *workflow errors* in query optimizers. In particular, the optimization steps are visualized in a hierarchical structure in QOVIS, which allows developers to investigate optimization steps and corresponding changes of plans at different levels of granularity. Moreover, our proposal QOVIS provides the explicit transformation logic of query plans, which links the operators among different query plans. Thus, the

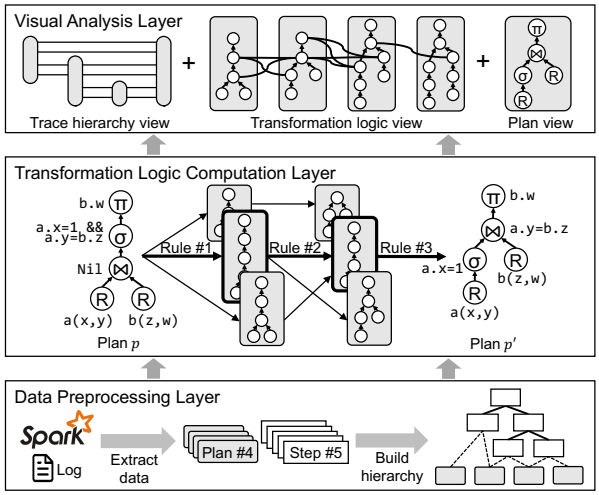


Figure 4: The QOVIS solution overview

cascading effects of different optimization steps on the same operator can be identified easily. For example, with the help of QOVIS, the root cause of the *workflow error* of Figure 3 is that the *Project* operators are repeatedly inserted in the *ColumnPruning* step but removed by other subsequent steps, as illustrated in Figure 3(c).

3 THE OVERVIEW OF QOVIS

Figure 4 provides a system overview of our proposal QOVIS, which is designed to assist database system developers in identifying *transformation error* and *workflow error* in the query optimizer and providing clues to fix them via a visualization-assisted approach. We introduce the layers of QOVIS from bottom to top as follows.

- The *data preprocessing layer* (at the bottom) collects system logs during query processing and extracts the optimization trace. The optimization steps are associated with the intermediate query plans and the hierarchy of optimization steps is constructed. We refer the interested readers to Appendix B at [9] for details.
- The *transformation logic computation layer* (in the middle) takes the query plans and computes the transformation logic between them. Returned transformation logic is crucial to analyze the *transformation error* and *workflow error*, which correspond to the correctness and efficiency of query optimizers, respectively.
- The *visual analysis layer* (at the top) is the front-end of QOVIS. With three carefully designed visualization views and interactions, it enables intuitive and explicit optimizer error identification and root cause localization.

Discussion. The generality of QOVIS stems from its flexible architecture as none of its components depends on the specific designs of the underlying database systems. Specifically, QOVIS only uses the optimization plans and steps during query processing, which are usually logged or can be easily obtained, for example, Apache Calcite provides hooks that can be invoked before and after executing an optimization step. Moreover, QOVIS is applicable to volcano/cascade style query optimizers as only the intermediate plans of the path (from the initial plan to final plan) among the explored plan space (e.g., via dynamic programming based search) will be investigated and analyzed.

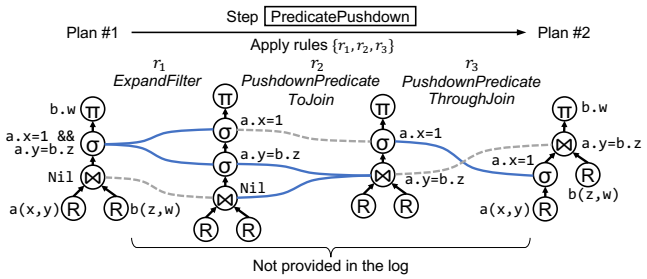


Figure 5: The plan transformation example

4 TRANSFORMATION LOGIC COMPUTATION

The transformation logic between intermediate query plans provides crucial clues for developers to understand and diagnose *transformation errors* and *workflow errors* in query optimizers. However, few existing systems provide such transformation logic in their logs. To address this, we first formally define the plan transformation problem in Section 4.1 and then present our approaches to computing logic during the transformation of the plan in Section 4.2.

4.1 Plan Transformation Problem

As illustrated by *transformation error* and *workflow error* examples (e.g., Figure 2 and Figure 3), the transformation logic shows the change of query plans (e.g., operators, arguments, hints), which is essential for database system developers to identify the correctness and efficiency issues in the underlying optimizer. However, existing solutions require manual efforts of the developers to analyze the transformation logic by utilizing their expertise. The reasons are twofold: (i) the system log does not include the transformation logic, and (ii) the optimization step is too coarse to analyze.

For example, as depicted in Figure 1(b), Plan #1 is transformed to Plan #2 after applying the optimization step *PredicatePushdown*, which is the most fine-grained optimization step in the query optimizer. However, Figure 5 illustrates the computed transformation logic of the *PredicatePushdown* step. Specifically, the solid blue curves show the changes of these operators and arguments in the query plans by applying specific optimization rules (e.g., r_1, r_2, r_3), and the dashed gray curves link those unchanged operators. These curves explicitly show that the predicate of the *Filter* (σ) in Plan #1 (i.e., `a.x=1 && a.y=b.z`) is split and pushed down to the *Join* ($\bowtie$) and another *Filter* operators in Plan #2.

Analyzing the above transformation logic between query plans manually is time-consuming. Moreover, computing such transformation logic is not trivial as (i) the correctness of the manually analyzed logic cannot be guaranteed; and (ii) it requires huge computation costs, as we will present shortly. To overcome them, we first define the optimization rule in Definition 3, then use it as the building block to formulate the plan transformation problem.

Definition 3 (Optimization Rule). *An optimization rule r transforms query plan p to p', i.e., $p' = p \oplus r$, and it guarantees p is semantically equivalent to p'.*

Example. Figure 6 depicts the example of applying optimization rule *ExpandFilter* on query plan p to obtain the optimized query plan p'. Specifically, the *Filter* operator (σ) in p is divided into two *Filter* operators in p' by separating the predicate `x=1 && y=2` via

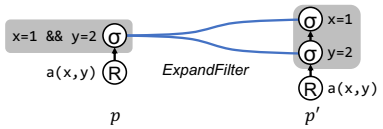


Figure 6: The optimization rule *ExpandFilter* example

conjunction property, i.e., $\sigma_{pred_1 \wedge pred_2}(\mathrm{R}) = \sigma_{pred_1}(\sigma_{pred_2}(\mathrm{R}))$, in relational algebra [55].

In each query optimizer, all these pre-defined optimization rules form the rule set $\mathcal{R}$ of the optimizer. For example, we include 25 different rules in the rule set $\mathcal{R}$ for Apache Spark 3.3.0, which are from the fundamental relational algebra [55] and its implemented optimization strategies. However, the rule set can be extended to support various transformation logics in different versions of Apache Spark or other database systems. Several recent studies [65, 69] are proposed to generate and verify additional optimization rules in the rule set $\mathcal{R}$. Our work is orthogonal to them as the rule set $\mathcal{R}$ of the query optimizer is the input of the following problem.

Problem 1 (Plan Transformation Problem). *Given a rule set $\mathcal{R}$ and two query plans p and p', the plan transformation problem is finding the transformation logic, which is a rule sequence $\Phi = \{r_1, \cdots, r_j\}$, such that (i) $\forall r_i \in \Phi$, $r_i \in \mathcal{R}$; (ii) $p' = p \oplus \Phi = p \oplus r_1 \oplus \cdots \oplus r_j$; and (iii) $|\Phi|$ is minimum.*

Example. Given the rule set $\mathcal{R}$ and two query plans p and p', see Plan #1 and Plan #2 shown in Figure 5. The plan transformation problem returns $\Phi = \{r_1 : \mathit{ExpandFilter}, r_2 : \mathit{PushdownPredicateToJoin}, r_3 : \mathit{PushdownPredicateThroughJoin}\}$, as the rules depicted in the middle of Figure 5.

The *minimum* constraint of Problem 1 ensures that the transformation logic between p and p' is concise. If there does not exist a Φ to transform p to p', it means the two plans are not equivalent with the given rule set $\mathcal{R}$. In other words, there may be a correctness issue, which should be investigated by the developers.

4.2 Problem Reduction and Solutions

Solving the above plan transformation problem is not easy as the number of possible rule sequences is exponential, i.e., $O(|\mathcal{R}|^{|\Phi|})$. In this section, we first reduce the problem to a shortest path problem. Then, we devise solutions to address it efficiently.

4.2.1 Problem Reduction. Logically, Problem 1 can be reduced to a shortest path problem on a directed graph. In particular, each node in the graph is a query plan and each edge in the graph is an optimization rule $r \in \mathcal{R}$ that transforms the incoming query plan to the out-going query plan. By setting the edge weight as 1, the shortest path between two query plans represents the minimum rule sequence that transforms the source plan into the target plan. Given the query plans p and p', Problem 1 is equivalent to finding the shortest path from p to p' in the graph.

Example. The plan transformation between p and p' can be reduced to find the shortest path between source node p and destination node p' in Figure 7. It is worth noting that we enumerate all possible rules and generate all intermediate query plans, see the dotted nodes in Figure 7. The shortest path between p and p' is $p \rightarrow p_6 \rightarrow p_{12} \rightarrow p'$, and the corresponding rule sequence $\Phi = \{r_1, r_2, r_3\}$, as shown in Figure 7.

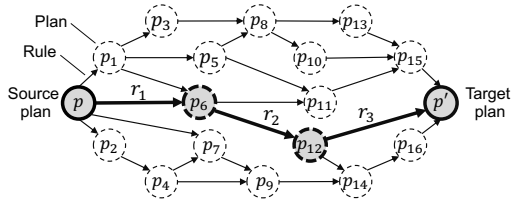


Figure 7: Reducing Problem 1 to a shortest path problem

In practice, it is almost impossible to enumerate all possible intermediate query plans and construct the graph then conduct the short path search for the result. The reasons are two-fold: (i) the number of possible intermediate plans is exponential with the rule sequence length; and (ii) it is expensive to generate every intermediate plan. Inspired by the heuristic search algorithms, we adapt the A^* algorithm to solve the reduced shortest path search problem. Specifically, A^* starts at query plan p. In each iteration, it selects the next intermediate plan $\widetilde{p}$ to explore such that $f(\widetilde{p}) = g(\widetilde{p}) + h(\widetilde{p})$ is minimized. $g(\widetilde{p})$ is the number of edges in the path from p to $\widetilde{p}$, i.e., it is the minimum number of the applied optimization rules that transform p to $\widetilde{p}$. $h(\widetilde{p})$ is a heuristic function that estimates the minimum number of edges in the path from $\widetilde{p}$ to p', i.e., the number of optimization rules that should be used to transform $\widetilde{p}$ to p'. A^* guarantees the optimality of the result if the heuristic function $h(\widetilde{p})$ is *admissible*, see the well-established conclusion in Lemma 1 [35].

Lemma 1 (Result optimality of A^*). *If the heuristic function $h(\widetilde{p})$ is admissible, meaning that it never overestimates the actual minimum rule number to get to p' from $\widetilde{p}$, then A^* is guaranteed to return the shortest path from p to p'.*

With the above property in mind, the technical challenge to address Problem 1 is to devise the heuristic function $h(\widetilde{p})$, which should not exceed the actual minimum number of optimization rules that transforms $\widetilde{p}$ to p', we denote it as $|\Phi^*|$. In addition, the estimated value of the heuristic function $h(\widetilde{p})$ should be as close as possible to the ground truth $|\Phi^*|$, thus, the high efficiency of A^* algorithm can be achieved. In the following, we introduce our solutions to reduce the computation cost.

4.2.2 The zero *Heuristic Function $\mathcal{H}_0$.* One of the most straightforward designs of heuristic functions for Problem 1 is $\mathcal{H}_0(\widetilde{p}) = 0$, which is admissible. With this heuristic function, A^* is equivalent to a BFS-based exhaustive search algorithm as edge weights are 1. Suppose the depth from p to p' is d in the above search process, the time complexity and space complexity of the BFS-based exhaustive search algorithm are both $O(|\mathcal{R}|^d)$. Obviously, this approach is impractical as the time and space costs are exponential.

4.2.3 Difference-based Heuristic Function $\mathcal{H}$. Using $\mathcal{H}_0(\widetilde{p})$ is inefficient as it does not take the information of p' into consideration during the search process. Intuitively, the number of optimization rules to transform $\widetilde{p}$ to p' is related to the difference between $\widetilde{p}$ and p'. However, there are many methods to measure the difference between two query plans, e.g., operator types, arguments of operators, and join orders. In this section, we introduce a simple yet effective measurement to quantitatively measure the difference between two query plans by only considering the operators. We left the extension to other query plan difference measurements as future work. In particular, we consider the difference of the given operator in two plans, see Definition 4.

Definition 4 (Operator type measurement). *Given two query plans p_i and p_j and the operator op, the difference of p_i and p_j on the operator op is defined as* $\text{odiff}_{op}(p_i, p_j) = |N(p_i, op) - N(p_j, op)|$, *where $N(p_i, op)$ is the number of the operator op in query plan p_i.*

Example: Taking the query plans p and p' in Figure 6 as an example, the difference between p and p' with regard to *Filter* operator σ is $\text{odiff}_{\sigma}(p, p') = |N(p, \sigma) - N(p', \sigma)| = |2 - 1| = 1$. It is worth pointing out that the operator type measurement $\text{odiff}_{op}(\cdot, \cdot)$ is a metric, we omit the proof due to page limits.

With the different number of given operators in two query plans $\widetilde{p}$ and p', we next consider how many optimization rules should be applied to transform $\widetilde{p}$ to p', which is estimated with Definition 5.

Definition 5 (Rule influence on Operator op). *Given a query plan p_i, the rule influence of $r \in \mathcal{R}$ w.r.t the operator op is defined as the number of affected operator op in the transformed $p_j = p_i \oplus r$, denoted as $\mathcal{I}_{op}(r)$.*

Example: The optimization rule *ExpandFilter* always splits one *Filter* operator in a query plan into two *Filter* operators in the generated query plan, see the example in Figure 6. Thus, we have $\mathcal{I}_{op}(\textit{ExpandFilter}) = 1$.

Given two query plans $\widetilde{p}$ and p', it is intuitive to derive the lower bound of the required optimization rules that transform $\widetilde{p}$ to p' when we know the difference of operator op between them (via Definition 4) and the rule influence on op (via Definition 5), which can be one of the heuristic function $\mathcal{H}(\widetilde{p})$ to estimate the number of required rules from $\widetilde{p}$ to p'. We formally define it as: $\mathcal{H}(\widetilde{p}) = \mathcal{H}_{op}(\widetilde{p}) = \left\lceil \frac{\text{odiff}_{op}(\widetilde{p}, p')}{max_{r \in \mathcal{R}} \mathcal{I}_{op}(r)} \right\rceil$.

To guarantee the returned result is minimal for Problem 1, we next prove $\mathcal{H}_{op}(\widetilde{p})$ is admissible via Theorem 1.

Theorem 1. *$\mathcal{H}_{op}(\widetilde{p})$ is a lower bound of the minimum number of required rules to transform $\widetilde{p}$ to p'.*

Proof. There are two cases: (I) If there does not exist a path from $\widetilde{p}$ to p', then $\mathcal{H}_{op}(\widetilde{p}) \leq |\Phi^*| = \infty$; (II) Otherwise, there is a shortest path $\Phi^* = \{r_1, \cdots, r_l\}$. Let $\{p_1, \cdots, p_{l+1}\}$ be the plan sequence by applying each rule in Φ^* sequentially, where $p_1 = \widetilde{p}$ and $p_{l+1} = p'$, and $\forall i \in [1, l]$, $p_{i+1} = p_i \oplus r_i$. Thus, it holds $\text{odiff}_{op}(\widetilde{p}, p')$

$= \text{odiff}_{op}(p_1, p_{l+1}) \leq \text{odiff}_{op}(p_1, p_2) + \cdots + \text{odiff}_{op}(p_l, p_{l+1})$

$\leq \mathcal{I}_{op}(r_1) + \cdots + \mathcal{I}_{op}(r_l) \leq max_{i \in [1,l]} \mathcal{I}_{op}(r_i) \times l$

$\leq max_{r \in \mathcal{R}} \mathcal{I}_{op}(r) \times l = max_{r \in \mathcal{R}} \mathcal{I}_{op}(r) \times |\Phi^*|$.

Thus, $\mathcal{H}_{op}(\widetilde{p})$ is proven to be admissible. □

Moreover, $\mathcal{H}_{op}(\widetilde{p})$ is admissible for every operator op in the operator set $\mathcal{O}$, thus, the heuristic function $\mathcal{H}(\widetilde{p})$ can be further tightened by setting: $\mathcal{H}(\widetilde{p}) = \mathcal{H}_{\mathcal{O}}(\widetilde{p}) = \max_{op \in \mathcal{O}} \{\mathcal{H}_{op}(\widetilde{p})\}$.

4.2.4 Enhanced Heuristic Function $\mathcal{H}^+$. The heuristic function $\mathcal{H}_{\mathcal{O}}(\widetilde{p})$ is obviously not tight as it takes the maximum $\mathcal{H}_{op}(\widetilde{p})$ among all possible operators. It means it only considers the number of optimization rules relative to the changes of a specific operator, and does not consider other operators in the query plans.

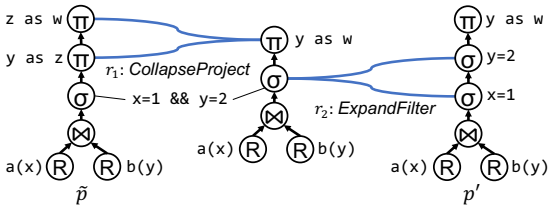
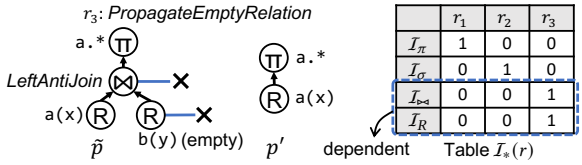


Figure 8: Two independent optimization rules



	r_1	r_2	r_3
$\mathcal{I}_\pi$	1	0	0
$\mathcal{I}_\sigma$	0	1	0
$\mathcal{I}_{\bowtie}$	0	0	1
$\mathcal{I}_R$	0	0	1

Figure 9: Optimization rule r_3 affects both $\mathcal{H}_{\bowtie}(\widetilde{p})$ and $\mathcal{H}_R(\widetilde{p})$

Considering two query plans $\widetilde{p}$ and p' in Figure 8, it applies *CollapseProject* and *ExpandFilter* optimization rules to transform p to p'. The estimated number of rules is 1 as it takes a maximum between $\mathcal{H}_\pi(\widetilde{p}) = 1$ and $\mathcal{H}_\sigma(\widetilde{p}) = 1$. However, the changes of the operators *Project* (π) and *Filter* (σ) are independent, e.g., applying *CollapseProject* and *ExpandFilter* separately. Logically, the estimated minimum rule number should be the sum of $\mathcal{H}_\pi(\widetilde{p})$ and $\mathcal{H}_\sigma(\widetilde{p})$. Inspired by the above observation, we improve the tightness of the heuristic function $\mathcal{H}(\widetilde{p})$ by taking the rule influence on different operators into account.

Definition 6 (Independent rule influence on operators). *Two heuristic functions of different operators $\mathcal{H}_{op_i}(\widetilde{p})$ and $\mathcal{H}_{op_j}(\widetilde{p})$ are independent iff $\nexists\, r \in \mathcal{R}, \mathcal{I}_{op_i}(r) > 0 \wedge \mathcal{I}_{op_j}(r) > 0$.*

Example. Considering all cases in Figure 8 and Figure 9, $\mathcal{H}_\pi(\widetilde{p})$ and $\mathcal{H}_\sigma(\widetilde{p})$ is independent, while $\mathcal{H}_{\bowtie}(\widetilde{p})$ and $\mathcal{H}_R(\widetilde{p})$ is not, since r_3 affects both *Join* and *Relation*, as shown in Figure 9.

To identify the relationship of the heuristic functions among different operators, we first compute the relationship table $\mathcal{I}_*(r)$ (see the exemplified table in Figure 9) by considering every op in $\mathcal{O}$, then classify these operators into a set of groups via Definition 6, i.e., $\mathcal{G} = \{G_1, G_2, \cdots, G_t\}$. For any two operators op_i and op_j in the same group, their heuristic functions $\mathcal{H}_{op_i}(\widetilde{p})$ and $\mathcal{H}_{op_j}(\widetilde{p})$ are independent, For operators op_i and op_j in different groups, their heuristic functions $\mathcal{H}_{op_i}(\widetilde{p})$ and $\mathcal{H}_{op_j}(\widetilde{p})$ are correlated. The relationship table $\mathcal{I}_*(r)$ can be precomputed as it only relies on the optimization rule properties. We last define the improved heuristic function $\mathcal{H}^+_O(\widetilde{p}) = \max_{\forall G_i \in \mathcal{G}}\{\sum_{op \in G_i} \mathcal{H}_{op}(\widetilde{p})\}$ for A^* algorithm. We omit the admissible proof of $\mathcal{H}^+_O(\widetilde{p})$ as it is straightforward.

5 VISUAL ANALYSIS DESIGN

The query plans, optimization steps, and transformation logic are provided after the processing of *data preprocessing layer* and *transformation logic computation layer* in QOVIS. However, database system developers face two major challenges in understanding and diagnosing optimization issues: (i) the overwhelming number of plans, steps, and logic in the query optimization procedure and (ii) the complex interrelationships among them. Both (i) and (ii) exceed the cognitive abilities of humans to effectively understand and analyze. To overcome these challenges, we propose a visualization-assisted approach in QOVIS that enables users to perform their analysis tasks efficiently and visually.

5.1 Task Analysis and Abstraction

Database system developers, i.e., the target users of QOVIS, perform analysis tasks when they are understanding and diagnosing the optimizer. To design a visualization-assisted approach that facilitates them, it is crucial to determine these user tasks [29] and abstract them into domain-independent terms [20].

5.1.1 User Task Analysis. The high-level goal of users is to identify and investigate the optimization issues (i.e., *transformation errors* and *workflow errors*) during query processing. To achieve this, users perform several low-level analysis tasks. We conduct a scenario-based task analysis [29] to determine the necessary user tasks. It is based on the issues we collected from *Apache Spark issue tracker website* [4]. We briefly summarize the identified tasks as follows.

(I) Plan analysis task. It is identifying the details of plans to understand their semantics and verify their equivalence. The detailed information includes operators, dependencies of operators and arguments associated with each operator. Taking Figure 2 as an example, the *transformation error* can be identified after performing plan analysis tasks as Plan #1 and Plan #2 in it are not equivalent.

(II) Step analysis task. It is investigating the used optimization steps in the optimizer to identify redundant steps and locate positions of errors. It includes step names, step hierarchy, and the associated plans (i.e., input/output plans). For example, the *workflow error* can be easily identified in Figure 3 as it has a repeated sequence of steps. Furthermore, the problematic steps can be found by investigating the plans associated with them.

(III) Transformation logic analysis task. It is analyzing the transformation logic among query plans to verify the correctness of transformations and track the problematic operators/arguments. For example, as shown in Figure 5, by checking the relationship depicted explicitly by curves, the plan transformation can be easily explained and verified.

5.1.2 User Task Abstraction. Based on the above commonly used user analysis tasks, we next abstract them to three domain-independent properties that the visualizations should effectively illustrate.

- **Topology.** It is involved in tasks (I) and (III). The visualization design should provide a clear topology of the query plans, e.g., operators and the data dependency of two connected operators, to illustrate the topological structure of the plans and highlight the changes of operators/arguments among them.
- **Hierarchy.** It is involved in tasks (II) and (III). The hierarchy of optimization steps should be visualized concisely to assist the users in understanding the hierarchical structure of steps and locating the associated query plans.
- **Relationship.** It is involved in the task (III). The transformation logic among different operators should be visualized without visual clutter, which helps the users to analyze the relationship of the operators/arguments between adjacent plans and among multiple plans and diagnose the optimization steps.

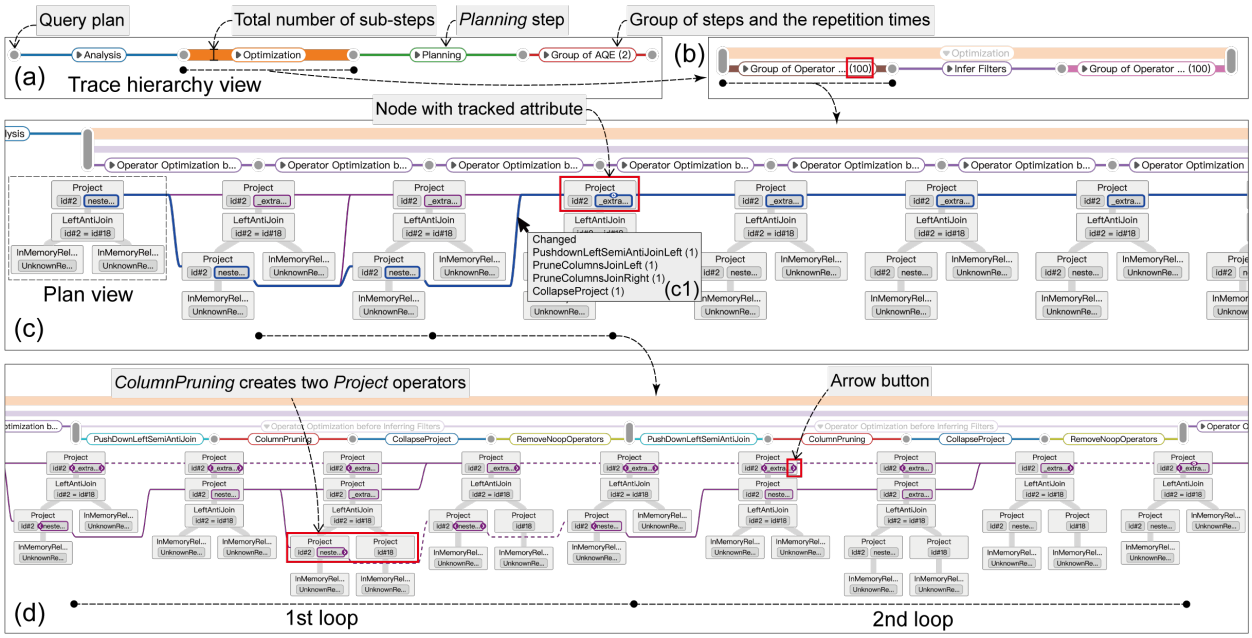


Figure 10: The visual analysis process for a *workflow error* (see Figure 3) in QOVIS

5.2 Visualization and Interaction Design

With our user tasks and abstracted visualization terms, we now introduce our visual design. Our interface comprises three components: *trace hierarchy view*, *plan view*, and *transformation logic view*. The design follows an "overview+details" scheme, displaying both overview and detailed views in separate presentation spaces [26].

Trace hierarchy view. This view serves as the overview and is positioned at the top of the interface. It is designed to (i) illustrate the steps and related plans during the optimization; and (ii) support hierarchical exploration to enable long optimization trace analysis. Figure 10(a) shows the *trace hierarchy view* in QOVIS with a *hierarchy-based sequence* visualization [32, 38, 46].

It utilizes gray dots to represent plans and lines connecting these dots to depict the steps that transform one plan into another. Step names are displayed as texts on the lines. Figure 10(a) demonstrates the highest level of a trace, comprising 4 major optimization steps: *Analysis*, *Optimization*, *Planning*, *AQE*. Specifically, colors encode steps, and the height of lines encodes the total number of optimization steps under the current step. For instance, the *Optimization* step has a higher line than the *Planning* step, indicating the *Optimization* step has more sub-steps. Furthermore, each step can be expanded or collapsed by clicking the line to show or hide its optimization steps. For example, when clicking the *Optimization* step in Figure 10(a), all sub-steps will be placed under *Optimization* step, see Figure 10(b), and the plans will be updated. Continuous repeated steps are grouped and visualized as one, using a number in a bracket to indicate the repetition time, shown as the red rectangle in Figure 10(b). Given a large volume of steps and plans, the *trace hierarchy view* enables developers to progressively explore the optimization process in a top-down manner, thereby preventing them from becoming overwhelmed by the vast amount of data.

Plan view. This view is embedded within the *transformation logic view* to demonstrate a single plan with its detailed information. The goal of the *plan view* is to intuitively display the topological structure of operators within a query plan, along with the arguments of these operators. The query plan is visualized using a *node-link diagram*, which is widely used to represent the topological structure of tree data [8, 12, 14, 44, 46, 47, 51, 63]. As the dashed rectangle in Figure 10(c) shows, the nodes present the operators, with their arguments placed within these nodes. The nodes are arranged from top to bottom and from left to right, to accurately depict parent-child and sibling relationships. For large and complex plans, we simplify the node by only visualizing the operator symbol and expanding it when it needs to be further analyzed. This layout ensures consistent position encoding of operators across adjacent plans and eases change tracking in large and complex plans.

Transformation logic view. This view is located at the bottom of the interface to show detailed plans. It is consistent with *trace hierarchy view*. It visually presents transformation logic between plans and guides users to identify the root causes of optimization errors. The *transformation logic view* presents a sequence of *plan views* with visible links showing transformation logic between adjacent plans. These links are the results of the plan transformation problem in Section 4.1. Initially, plans are displayed under the corresponding gray dots in the *trace hierarchy view*. When users click a specific argument within a plan, purple links appear to show the transformation logic of this argument. For example, as shown in Figure 10(c), after clicking `"_extra..."` within the node in a red rectangle, the transformation logic of this attribute is displayed by purple lines among all plans. Lines are designed to avoid *overlapping* with nodes and minimize *crossings*, which are key factors in reducing visual clutter in graph visualizations [21, 64]. As shown in Figure 10(d), links connecting the same operators in two plans are depicted with purple dashed lines, and arrow buttons at both ends allow users to jump directly past unchanged steps. Hovering over a line highlights the transformation logic path in blue (Figure 10(c))

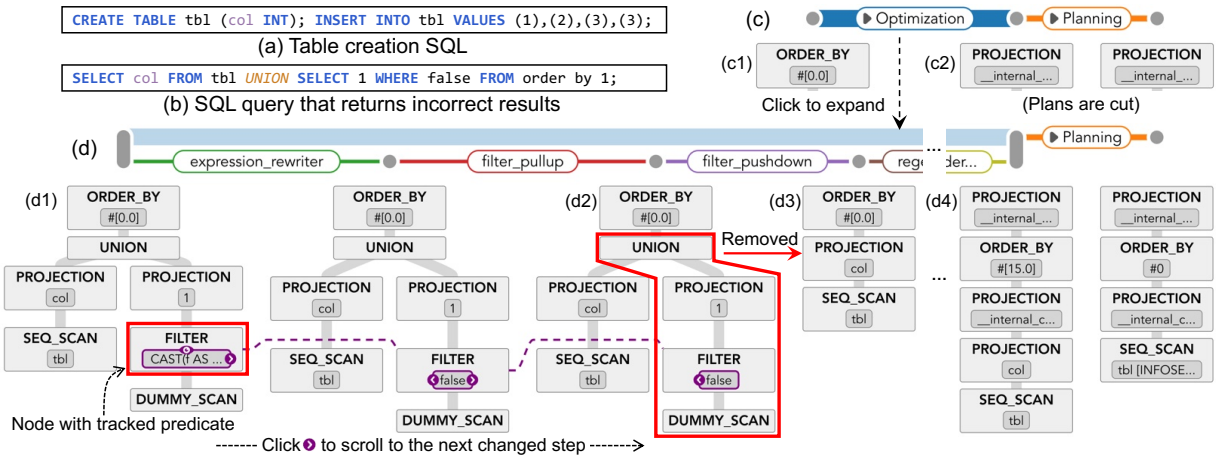


Figure 11: The diagnosis process for a *transformation error* of DuckDB using QOVIS, where a subquery is pruned incorrectly.

and displays a tooltip with the optimization rules affecting the argument (see (c1) in Figure 10(c)). The *transformation logic view* simplifies tracking the evolution of a plan, its operators, and its arguments from the initial plan, enabling users to quickly identify issues and root causes within the optimization trace.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of QOVIS by conducting extensive experimental studies.

6.1 Case Study

In this section, we use QOVIS to analyze three real cases in Apache Spark, Apache Hive, and DuckDB, respectively. All issues are reported and fixed by the developers in the community. We provide three other real-world cases of these systems in Appendix A at [9].

6.1.1 Loop Optimization in Apache Spark. Suppose the SQL query in Figure 3(a) is running on Apache Spark 3.3.0 with the data schema in Figure 3(a). Its *trace hierarchy view* is depicted in Figure 10(a), which shows the overview of its query optimization trace.

Identify the error in the optimization trace. We can observe that the height of the *Optimization* step, as the orange line shown in Figure 10(a), is higher than the heights of the other three steps (i.e., *Analysis*, *Planning*, *AQE*). It means the *Optimization* step has a large number of sub-steps. We next explore the details of the *Optimization* step by expanding it, which are illustrated in Figure 10(b). The repetition count of *Operator Optimization before Inferring Filters* step is 100, which is obviously larger than the usual case. Thus, we further investigate this abnormal step by clicking its triangle icon, the result is visualized in Figure 10(c). Interestingly, the query plan is not changed during the 100 optimization steps. Thus, there exists a *workflow error* in the optimization trace, i.e., loop optimization.

Locate root cause of the error. To locate the root cause of the loop optimization in the optimization trace, we investigate two consecutive *Operator Optimization before Inferring Filters* steps, as the dotted line shown in Figure 10(c). In particular, these two consecutive steps incur two optimization loops, as illustrated in Figure 10(d). Each loop consists of 4 optimization steps. We observe that two *Project* operators are generated during the *ColumnPrunning* of the 1st loop, shown in the red rectangle of Figure 10(d). The right *Project* operator is removed in *RemoveNoopOperators* of the 1st loop. The left one is merged during the *CollapseProject* in the 2nd loop, as the purple lines highlighted in Figure 10(d). In other words, the *ColumnPrunning* step is eliminated by the *CollapseProject* and *RemoveNoopOperators* steps during the query optimization trace, which is the root cause of the above loop optimization issue.

6.1.2 Incorrect Subquery Pruning in DuckDB. It is a *transformation error* of DuckDB v0.10.0, which is reported on its GitHub issue page [5]. In particular, the SQL query in Figure 11(b) is equivalent to selecting distinct values from table `tbl` (in Figure 11(a)). However, DuckDB v0.10.0 returns `[1,2,3,3]`, which is incorrect.

Identify the error in the optimization trace. We start our investigation by examining the optimization trace overview in Figure 11(c). We identify an incorrect transformation from the plan (d1) to plan (d4). Obviously, the *UNION* and its associated right branch of plan (d1) are pruned in (d4), which results in the loss of "distinct" property of the *UNION* operator. Thus, there is a *transformation error* in the *Optimization* step.

Locate root cause of the error. To identify the root cause of this issue, we first expand *Optimization* step to examine its detailed procedure in Figure 11(d). We trace the elimination of the right branch of *UNION* by clicking operator arguments on it, such as the predicate of the *FILTER* operator (highlighted by the red rectangle in plan (d1)). After clicking the right arrow button associated with this predicate, the view automatically scrolls to the next changed step of this predicate, i.e., the *filter_pushdown* step. We find this step is incorrectly implemented — transforming plan (d2) to (d3). The bug is fixed by a patch for the *filter_pushdown* step [7]. This fix matches our analysis and confirms the effectiveness of QOVIS.

6.1.3 Recursive Application of Optimization Steps in Apache Hive. It is a *workflow error* in Apache Hive v4.0.0, which is reported on its issue tracker website [1]. With the created tables by Figure 12(a), the optimizer fails to compile the SQL query in Figure 12(b) and throws an Out-Of-Memory (OOM) exception.

Identify the error in the optimization trace. Figure 12(c) shows the overview of the optimization trace and the last step is not finished before exiting, see the warning glyph. We thus expand

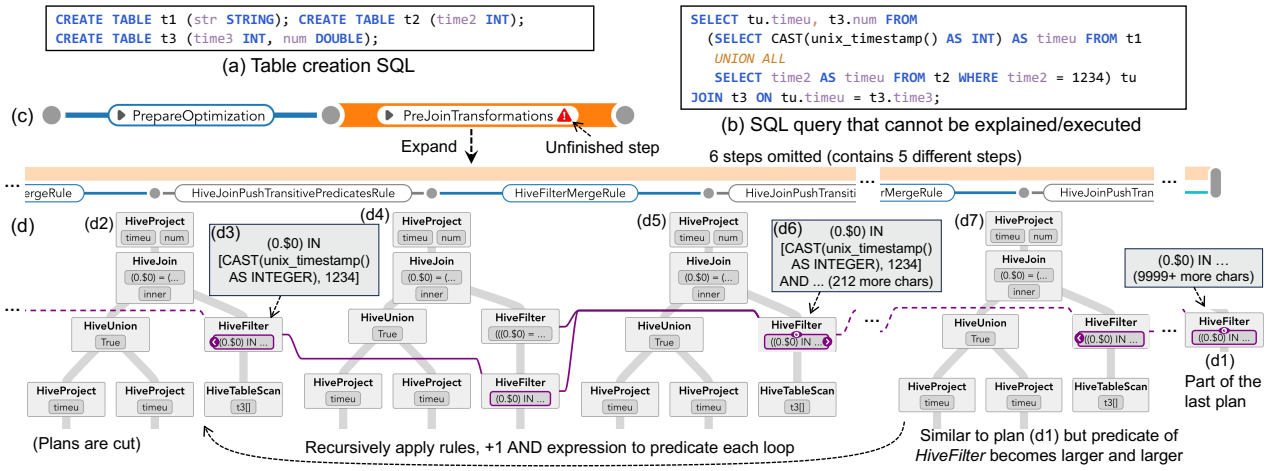


Figure 12: The diagnosis process for a *workflow error* of Apache Hive using QOVIS, where a loop occurs in the optimization.

and check this step, as depicted in Figure 12(d). In the rightmost plan (see d1), we find a *HiveFilter* operator with an extremely large predicate. By hovering over this predicate, a tooltip shows that the string of it has more than 9,999 chars. Moreover, we find that this predicate becomes larger and larger abnormally from a small one during the optimization, see the tooltips (d3) and (d6). We thus speculate that the expansion of this predicate results to the crash.

Locate root cause of the error. Clicking the problematic predicate in *HiveFilter* operator in the last plan (d1), all related predicates of different plan operators are highlighted and linked by purple lines in Figure 12(d). By following these links, we observed a loop optimization on this predicate. In the first loop, starting from plan (d2), the problematic predicate is pushed down by *HiveJoinPushTransitivePredicatesRule* step, see the solid purple line between (d2) and (d4). Then it is merged to another *HiveFilter* operator to generate (d5) by *HiveFilterMergeRule* step, see two solid purple lines between (d4) and (d5). After executing six additional steps, plan (d7) is created. Interestingly, plan (d7) is identical to the plan (d2), except the *HiveFilter* operator with the problematic predicate. Subsequently, the *HiveJoinPushTransitivePredicatesRule* step is repeatedly applied to plan (d7), creating a new *HiveFilter* operator. This new operator triggers an explosive expansion of the predicate in the next iteration. It is a *workflow error* as it is caused by incorrect interactions among five optimization steps, even though each step individually is correct. As the description in the fix patch [3], all five steps are modified to prevent the recursive loop.

6.2 User Study

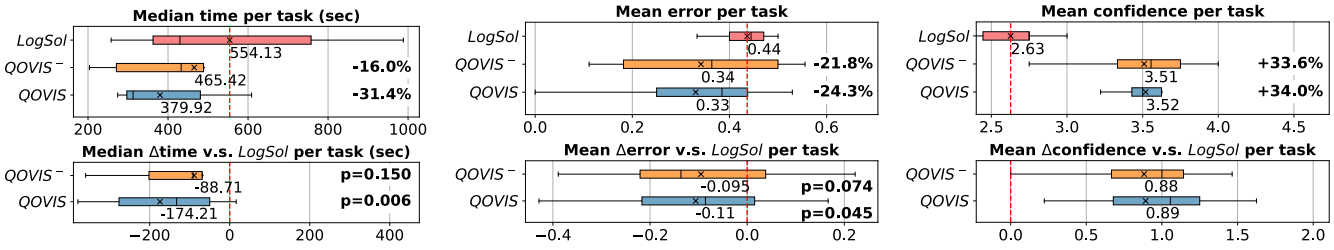
In this section, we conduct a user study to verify whether users can identify the errors/issues in the query optimization trace in *less time* and with *fewer errors*. The user study details are at [15].

Compared methods. The traditional methods to analyze the query optimization trace are based on logs. We denote them as LogSol. In particular, participants analyze optimization logs via a text editor with several interactive features, e.g., keyword searching. For our solutions, we evaluated two versions of our proposal: (i) QOVIS, which uses all techniques proposed in QOVIS, and (ii) QOVIS$^-$, which is a constrained version of QOVIS by removing transformation logic view and related interaction in *visual analysis* layer.

Study participants. We recruited 24 participants (21 males and 3 females, aged 19 to 31) for our user study. These participants include PhD students researching database systems and database system developers working in the industry. All of them had prior experience with database systems for about 4.7 years on average. Twelve of them have experience in developing core components (e.g., executor, optimizer) in database systems.

Studied cases. To evaluate QOVIS on various cases, we prepared 8 cases from three systems (i.e., Apache Spark, Apache Hive, and DuckDB) for the user study. In particular, the queries in these cases vary in join types, the number of joins (ranging from 1 to 4) and the scalability of optimization steps (from 47 to 1044 steps). They involve different features of optimizers, such as subquery rewrites and hint processing. Five of them are real queries that include optimization issues, i.e., *workflow error* and *transformation error*.

Study procedure and tasks. During the study, participants are asked to perform designed tasks via a web-based system. Each study lasts approximately two hours, beginning with a 40-minute tutorial. During the tutorial, we introduce the motivation of study, background, and the visualization and interaction designs in QOVIS. After that, participants conduct an exemplified study to familiarize themselves with the interface and the tasks. Then, they perform 9 tasks within approximately 80 minutes. Finally, participants submit their feedback and suggestions for QOVIS via a questionnaire. For each participant, the first 6 tasks are created by assigning one of three methods (LogSol, QOVIS$^-$, and QOVIS) to 6 cases selected from 8 cases. The assignments follow the Latin square design principle [39, 43] to ensure balance and unbiasedness. Participants use the assigned method to analyze the cases and determine whether there is an optimization error. The last 3 tasks focus on the three problematic cases. Each participant needs to locate the root causes of them with the previously assigned method. To complete each task, participants answer a multiple-choice question (with 4 options), designed following the guidelines in [70]. Additionally, participants respond to a single-choice question (with 5 options) regarding their confidence in their answers. The confidence levels are: “Not Confident”,



(a) Efficiency analysis (b) Accuracy analysis (c) Confidence analysis

Figure 13: Median time, mean error rates and mean confidence levels for all 3 methods on these tasks (× is the mean value). The results strongly support the conclusion that QOVIS assists users in completing tasks in *less time* ($p = 0.006 < 0.05$) and with *fewer errors* ($p = 0.045 < 0.05$), while users have the highest confidence score (3.52).

"Slightly Confident", "Confident", "Very Confident", and "Extremely Confident". The system records the time taken, the answers for each task, and the responses to the questionnaire.

Analysis method. The *time* and *error* are typical measurements in the evaluation of *user performance* [37, 41]. The tested hypotheses of the user study on LogSol, QOVIS$^-$ and QOVIS are:

- **H1**: in terms of efficiency, the time cost of the participants for the user study is $\mathcal{T}$(QOVIS) < $\mathcal{T}$(QOVIS$^-$) < $\mathcal{T}$(LogSol).
- **H2**: in terms of accuracy, the accurate rate of the participants for the user study is $\mathcal{A}$(QOVIS) > $\mathcal{A}$(QOVIS$^-$) > $\mathcal{A}$(LogSol).

We conduct traditional null hypothesis significance tests to evaluate these hypotheses. The normality of distribution for the collected user study results is tested by the Shapiro-Wilk test [52] with $\alpha = 5\%$. We next utilize non-parametric statistical tests [54] (i.e., one-tailed Wilcoxon signed-rank [66] tests with $\alpha = 5\%$) on them as the tested results above confirm that it is not a normal distribution. The returned p-value shows the compatibility of null hypotheses with the tested data, which $p < 0.05$ means that there is strong evidence against the null hypothesis; in other words, the tested hypothesis is statistically significant.

Efficiency analysis. The top figure in Figure 13(a) utilizes the boxplot to show the distribution of the median time participants took to answer each task by using three different methods. The mean values are marked by × within each bar. The time cost of the participants using QOVIS (and QOVIS$^-$) is obviously smaller than that of using LogSol. In particular, compared with LogSol, QOVIS and QOVIS$^-$ reduce 31.4% and 16.0% time cost, respectively. The bottom chart of Figure 13(a) illustrates the time difference between QOVIS (resp. QOVIS$^-$) and LogSol. Specifically, the median time cost of QOVIS and QOVIS$^-$ is faster than LogSol by 174.21 and 88.71 seconds. Moreover, QOVIS and QOVIS$^-$ are consistently faster than LogSol in almost all (if not all) tested cases. It confirms the effectiveness of QOVIS for all participants. The p-value for QOVIS is 0.006, which strongly supports the conclusion that QOVIS assist users in completing tasks in *less time*. In addition, as shown in Figure 13(a), the median time cost of QOVIS is less than its of QOVIS$^-$. In general, QOVIS is better than QOVIS$^-$ for the participants to explore optimization issues. However, some participants report that QOVIS$^-$ is more efficient in some special cases, as it visualizes fewer elements so that it is easier to focus on key query plans.

Accuracy analysis. Figure 13(b) presents the error rates of participants using three different methods. We say the answer is not correct (i.e., the above reported "error rate") when the participants give the wrong answer to the task. It is clear that the participants could identify the corresponding errors in the optimization trace more accurately by using QOVIS and QOVIS$^-$. In particular, the mean error rates of QOVIS and QOVIS$^-$ are 0.343 and 0.338, respectively. Both methods are obviously better than LogSol, which has a mean error rate of 0.424, see the top figure in Figure 13(b).

The p-value for QOVIS is 0.045, which strongly supports the conclusion that QOVIS assist users in completing tasks with *few errors*. Interestingly, QOVIS also has a smaller box size than QOVIS$^-$, which suggests that QOVIS has more consistent performance than QOVIS$^-$ among all participants. It can also be confirmed by the p-values of the accuracy hypotheses for QOVIS and QOVIS$^-$ (i.e., 0.045 < 0.074), which are depicted at the bottom of Figure 13(b). Furthermore, the negative mean error difference per task shows that QOVIS$^-$ and QOVIS effectively reduce the error rates.

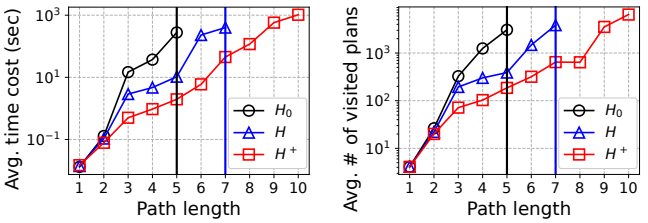
Confidence analysis. Figure 13(c) illustrates the participants' confidence distribution of the three methods in the study. The confidence levels are on a scale from 1 to 5, which correspond to "Not Confident", "Slightly Confident", "Confident", "Very Confident", and "Extremely Confident" respectively. The top chart in Figure 13(c) shows that the confidence scores for LogSol range from 2 to 3, indicating levels from "Slightly Confident" to "Confident". However, the confidence levels of the participants by using both QOVIS and QOVIS$^-$ are from "Confident" to "Very Confident" as their scores are consistently above 3. Comparing with LogSol, QOVIS and QOVIS$^-$ significantly enhance the confidence of the participants to their answers for the study tasks, which is confirmed by the mean difference (e.g., 0.89 and 0.88) in the bottom of Figure 13(c).

User feedback. All participants agree QOVIS is the most user-friendly and efficient method among three competitors. For example, the positive feedback highlights exploring with QOVIS reduces the amount of data to analyze and enables participants to focus on interested parts, which will be more efficient for large and complex queries. Several participants also report that the system is hard to use at first, and it is alleviated by practicing with more tasks.

6.3 Performance Study

In this section, we evaluate the performance of our proposed algorithms (i.e., $\mathcal{H}_0$, $\mathcal{H}$, and $\mathcal{H}^+$) for the plan transformation problem.

Test dataset. To obtain the test dataset for the plan transformation problem, we collect query optimization traces by executing two



(a) Average time cost (sec) (b) Average number of visited plans

Figure 14: The performance study result

categories of queries in Apache Spark 3.3.0: (i) 13 queries in the SSB [45] with a scale factor of 1; and (ii) 6 real-world queries from *Apache Spark issue tracker website*, 3 of them include optimization issues. From the searching process of *each transformation* in these traces, we randomly sample 1000 query plan pairs and compute their minimum rule sequence using breadth-first search as ground truth. The sequence length is evenly distributed from 1 to 10. We gather both simple cases that are easier to characterize using our heuristic functions (e.g., differ only in operator types) and difficult cases (e.g., differ in operator types, join order and operator arguments).

Experimental setup. We implement three algorithms in Python3 and run experiments on a server with a 3.80GHz Intel(R) Core i7-10700K CPU and 64GB memory. The operating system is Ubuntu 20.04. For each plan pair, we run three different algorithms separately and measure the time cost and the number of visited unique plans. We use the number of visited plans to show the memory consumption of each algorithm as the visited query plans are stored in the main memory. The timeout is set as 1800 seconds per case as it cannot find the rule sequence if there is a transformation error.

Result analysis. Figure 14 reports the performance of our devised two heuristic functions (i.e., $\mathcal{H}$ and $\mathcal{H}^+$) against $\mathcal{H}_0$ by varying the length of rule sequences from 1 to 10. The lines are cut whenever more than 25% of cases with the same sequence lengths cannot be completed within the timeout (shown as a colored vertical line). Compared to $\mathcal{H}_0$, our proposed heuristic functions significantly reduce the time cost, as depicted in Figure 14(a). In particular, the transformation logic of all input plan pairs can be returned by using our $\mathcal{H}^+$. The maximum time cost using $\mathcal{H}^+$ is 1032 seconds. In addition, $\mathcal{H}_0$ times out when the sequence length exceeds 5. The time cost of $\mathcal{H}_0$ increases exponentially with the rising sequence length, which is consistent with our theoretical complexity analysis (i.e., $|\mathcal{R}|^d$). Figure 14(b) illustrates the number of visited query plans for all three methods. As shown in Figure 14(b), the trend of unique visited plans in each algorithm is consistent with its time cost. The gap of the visited plans between $\mathcal{H}_0$ and our proposed methods $\mathcal{H}$ and $\mathcal{H}^+$ becomes obvious when the sequence length is large. Moreover, $\mathcal{H}^+$ always takes less time (resp. the number of visited plans) than $\mathcal{H}$ due to the power of the enhanced heuristic function.

7 RELATED WORK

Automatic Query Process Diagnosing. Many studies have been proposed to automatically detect the issues in the relational database management system (RDBMS) [18, 50, 59]. Rigger et al. [50] focus on detecting logical bugs when the generated plans produce incorrect results w.r.t the ground truth. Tang et al. [59] discover logic bugs in RDBMS via the automated testing method. These tools can identify issues in RDBMS, but they do not provide further explanations for the issues. Moreover, they can only detect logic bugs, and they cannot identify the errors and locate the root causes.

Interactive Query Optimization Investigation. Recently, several research work are proposed to analyze the query optimization process by involving domain experts [16, 34, 47, 48, 56–58, 63]. Many tools are also devised in the industry, e.g., Spark UI [12], Microsoft SQL Server Management Studio (SSMS) [8], and Tez UI [14]. Our QOVIS differs from them by three-fold: (i) these systems are not designed to understand and diagnose the query optimizer systematically; (ii) they do not illustrate fine-grained optimization steps and related plans; and (iii) QOVIS visualizes the transformation logic among multiple intermediate query plans, which assists the developers in identifying the root causes of the occurred errors.

Interactive Query Execution Investigation. Perfopticon [44], QEVIS [53], and DHive [67] provide visual analytics interface with coordinated views and flexible interactions, which allow users to thoroughly observe and diagnose the query execution process. The major difference of our work is that QOVIS focuses on the query optimizer, which has not been studied by these existing work.

SQL Query and Query Plan Visualization. Many work focus on improving the understanding of queries and query plans for the database end users. The visualizations can illustrate the high-level structure of queries in an intuitive way, so users can observe the query logic easily [23, 31, 41]. QueryVis [41] and STRATISFIMAL LAYOUT [28] develop a node-link diagram to visualize the query logic to facilitate the query understanding and communication. Query plans are one of the most straightforward ways to understand the execution logic of a query, many tools [33, 44, 47, 48, 58, 63] have been proposed to visualize them. For example, Starburst [33] develops the Query Graph Model (QGM) to represent the query plan in a graph structure. However, both SQL query and query plan visualizations cannot be adapted to understand and diagnose the query optimizer in our work.

8 CONCLUSION

In this work, we propose QOVIS to identify the bugs/errors and locate their root causes in query optimizers. Specifically, it consists of three layers: data preprocessing layer, transformation logic computation layer, and visual analysis layer. It is generic as it only uses the optimization step and query plans on the system log, and does not rely on any specific designs of the underlying database systems. To verify its generality, we utilize it to understand and diagnose different query optimizers on three widely used systems (Apache Spark, Apache Hive, and DuckDB). Two possible future research directions are: (i) proposing other measurements to quantify the difference between query plans; and (ii) supporting the analysis of parameterized queries and large queries.

ACKNOWLEDGMENTS

We sincerely thank the reviewers for their insightful comments. This work was partially supported by National Science Foundation of China (NSFC No. 62422206), Hong Kong Research Grants Council (GRF 152043/23E), and a research gift from AlayaDB.AI.

REFERENCES

[1] 2024. *Apache Hive issue: Recursive application of rules.* https://issues.apache.org/jira/browse/HIVE-25758

[2] 2024. *Apache Hive issue tracker website.* https://issues.apache.org/jira/projects/HIVE/issues

[3] 2024. *Apache Hive patch for issue: Recursive application of rules.* https://github.com/apache/hive/pull/2840

[4] 2024. *Apache Spark issue tracker website.* https://issues.apache.org/jira/projects/SPARK/issues

[5] 2024. *DuckDB issue: Incorrect subquery pruning.* https://github.com/duckdb/duckdb/issues/11294

[6] 2024. *DuckDB issue tracker website.* https://github.com/duckdb/duckdb/issues

[7] 2024. *DuckDB patch for issue: Sub-optimal optimization.* https://github.com/duckdb/duckdb/pull/11315

[8] 2024. *Microsoft SQL Server Management Studio (SSMS) documentation.* https://learn.microsoft.com/zh-cn/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16

[9] 2024. *The source code of QOVIS.* https://github.com/DBGroup-SUSTech/QOVIS.

[10] 2024. *Spark issue: Incorrect plan transformation.* https://issues.apache.org/jira/browse/SPARK-41162.

[11] 2024. *Spark issue: Loop optimization.* https://issues.apache.org/jira/browse/SPARK-37222.

[12] 2024. *Spark UI documentation.* https://spark.apache.org/docs/latest/web-ui.html

[13] 2024. *Studied optimization issues.* https://docs.google.com/spreadsheets/d/1A4VSKQkVnhceMOUp3N071a7TMtKQVhWk8iyj6g8U5-8/edit?usp=sharing.

[14] 2024. *Tez UI documentation.* https://tez.apache.org/tez-ui.html

[15] 2024. *User study materials of QOVIS.* https://github.com/DBGroup-SUSTech/QOVIS/tree/master/eval/user_study.

[16] Christoph Anneser, Mario Petruccelli, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, Ryan Marcus, and Alfons Kemper. 2023. QO-Insight: Inspecting Steered Query Optimizers. *PVLDB* 16, 12 (2023), 3922–3925.

[17] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.

[18] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *ICSE*. 2060–2071.

[19] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*. 221–230.

[20] Matthew Brehmer and Tamara Munzner. 2013. A Multi-Level Typology of Abstract Visualization Tasks. *TVCG* 19, 12 (2013), 2376–2385.

[21] Michael Burch, Corinna Vehlow, Natalia Konevtsova, and Daniel Weiskopf. 2012. Evaluating Partially Drawn Links for Directed Graph Edges. In *International Symposium on Graph Drawing*. 226–237.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[23] Claudio Cerullo and Marco Porta. 2007. A System for Database Visual Querying and Query Visualization: Complementing Text and Graphics to Increase Expressiveness. In *18th International Workshop on Database and Expert Systems Applications*. 109–113.

[24] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *PVLDB* 11, 11 (2018), 1482–1495.

[25] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.

[26] Andy Cockburn, Amy Karlson, and Benjamin B Bederson. 2009. A review of overview+detail, zooming, and focus+context interfaces. *CSUR* 41, 1 (2009), 1–31.

[27] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alex Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. Guided automated learning for query workload re-optimization. *PVLDB* 12, 12 (2019), 2010–2021.

[28] Sara Di Bartolomeo, Mirek Riedewald, Wolfgang Gatterbauer, and Cody Dunne. 2021. STRATISFIMAL LAYOUT: A modular optimization model for laying out layered node-link network visualizations. *TVCG* 28 (2021), 324–334.

[29] Dan Diaper and Neville Stanton. 2003. The handbook of task analysis for human-computer interaction. (2003).

[30] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving query equivalence using linear integer arithmetic. *Proc. ACM Manag. Data* 1, 4 (2023), 227:1–227:26.

[31] Wolfgang Gatterbauer. 2011. Databases will Visualize Queries too. *PVLDB* 4, 12 (2011), 1498–1501.

[32] Yi Guo, Shunan Guo, Zhuochen Jin, Smiti Kaul, David Gotz, and Nan Cao. 2021. Survey on visual analysis of event sequence data. *TVCG* 28, 12 (2021), 5091–5112.

[33] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. 1989. Extensible Query Processing in Starburst. In *SIGMOD*. 377–388.

[34] Jayant R. Haritsa. 2010. The Picasso Database Query Optimizer Visualizer. *PVLDB* 3, 1-2 (2010), 1517–1520.

[35] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[36] Michael Hausenblas and Jacques Nadeau. 2013. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big data* 1, 2 (2013), 100–104.

[37] Tobias Isenberg, Petra Isenberg, Jian Chen, Michael Sedlmair, and Torsten Möller. 2013. A Systematic Review on the Practice of Evaluating Visualization. *TVCG* 19, 12 (2013), 2818–2827.

[38] Po-Ming Law, Zhicheng Liu, Sana Malik, and Rahul C. Basole. 2018. MAQUI: Interweaving Queries and Pattern Mining for Recursive Event Sequence Exploration. *TVCG* 25, 1 (2018), 396–406.

[39] Johannes Ledolter and Arthur J. Swersey. 2007. *Testing 1-2-3: experimental design with applications in marketing and service operations.* Stanford University Press.

[40] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[41] Aristotelis Leventidis, Jiahui Zhang, Cody Dunne, Wolfgang Gatterbauer, HV Jagadish, and Mirek Riedewald. 2020. QueryVis: Logic-based Diagrams help Users Understand Complicated SQL Queries Faster. In *SIGMOD*. 2303–2318.

[42] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. 1275–1288.

[43] Douglas C Montgomery. 2017. *Design and analysis of experiments.* John wiley & sons.

[44] Dominik Moritz, Daniel Halperin, Bill Howe, and Jeffrey Heer. 2015. Perfopticon: Visual Query Analysis for Distributed Databases. *CGF* 34, 3 (2015), 71–80.

[45] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The Star Schema Benchmark (SSB). *Pat* 200, 0 (2007), 50.

[46] Aditeya Pandey, Uzma Haque Syeda, Chaitya Shah, John A Guerra-Gomez, and Michelle A Borkin. 2021. A State-of-the-Art Survey of Tasks for Tree Design and Evaluation With a Curated Task Dataset. *TVCG* 28, 10 (2021), 3563–3584.

[47] Yue Pang, Linglin Yang, Lei Zou, and M. Tamer Özsu. 2023. gFOV: A Full-Stack SPARQL Query Optimizer & Plan Visualizer. In *CIKM*. 5081–5085.

[48] Zhibo Peng, Mitch Cherniack, and Olga Papaemmanouil. 2014. Devel-op: An optimizer development environment. In *ICDE*. 1278–1281.

[49] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*. 1981–1984.

[50] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *FSE/ESEC*. 1140–1152.

[51] Hans-Jorg Schulz, Steffen Hadlak, and Heidrun Schumann. 2010. The Design Space of Implicit Hierarchy Visualization: A Survey. *TVCG* 17, 4 (2010), 393–411.

[52] Samuel Sanford Shapiro and Martin B Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3-4 (1965), 591–611.

[53] Qiaomu Shen, Zhengxin You, Xiao Yan, Chaozu Zhang, Ke Xu, Dan Zeng, Jianbin Qin, and Bo Tang. 2023. QEVIS: Multi-grained Visualization of Distributed Query Execution. *TVCG* 30, 1 (2023), 153–163.

[54] David J Sheskin. 2003. *Handbook of parametric and nonparametric statistical procedures.* Chapman and hall/CRC.

[55] Abraham Silberschatz, Henry F Korth, and Shashank Sudarshan. 2011. *Database system concepts.* McGraw-Hill Education.

[56] Toni Taipalus. 2023. Query Execution Plans and Semantic Errors: Usability and Educational Opportunities. In *CHI*. 1–6.

[57] Toni Taipalus and Hilkka Grahn. 2023. Framework for SQL Error Message Design: A Data-Driven Approach. *TOSEM* 33, 1 (2023), 1–50.

[58] Jess Tan, Desmond Yeo, Rachael Neoh, Huey-Eng Chua, and Sourav S Bhowmick. 2022. MOCHA: A Tool for Visualizing Impact of Operator Choices in Query Execution Plans for Database Education. *PVLDB* 15, 12 (2022), 3602–3605.

[59] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1 (2023), 55:1–55:26.

[60] Xiu Tang, Sai Wu, Dongxiang Zhang, Ziyue Wang, Gongsheng Yuan, and Gang Chen. 2023. A Demonstration of DLBD: Database Logic Bug Detection System. *PVLDB* 16, 12 (2023), 3914–3917.

[61] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009), 1626–1629.

[62] Fang Wang, Xiao Yan, Man Lung Yiu, Shuai Li, Zunyao Mao, and Bo Tang. 2023. Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation. *Proc. ACM Manag. Data* 1, 1 (2023), 28:1–28:25.

[63] Hu Wang, Hui Li, Sourav S Bhowmick, and Baochao Xu. 2023. ARENA: Alternative Relational Query Plan Exploration for Database Education. In *Companion of the 2023 International Conference on Management of Data*. 107–110.

[64] Yong Wang, Qiaomu Shen, Daniel Archambault, Zhiguang Zhou, Min Zhu, Sixiao Yang, and Huamin Qu. 2015. AmbiguityVis: Visualization of Ambiguity in Graph Layouts. *TVCG* 22, 1 (2015), 359–368.
[65] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *SIGMOD*. 94–107.
[66] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 196–202.
[67] Chaozu Zhang, Qiaomu Shen, and Bo Tang. 2023. DHive: Query Execution Performance Analysis via Dataflow in Apache Hive. *PVLDB* 16, 12 (2023), 3998–4001.
[68] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *PVLDB* 12, 11 (2019), 1276–1288.
[69] Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *ICDE*. 2735–2748.
[70] Dawn M. Zimmaro. 2010. Writing Good Multiple-Choice Exams. *Center for Teaching and Learning, UT Austin* (2010).