APPENDIX

A Supplemental Optimization Issue Examples

In this section, we provide three extra issue examples from Apache Spark and DuckDB, including both *transformation errors* and *workflow errors*. They are representatives of the issues related to join operator optimization, sort operator optimization, and SQL hint processing, respectively.

A.1 Incorrect Plan Transformation. We run the SQL query in Figure 2(a) on Apache Spark 3.3.0 with the input table `t1` in Figure 2(b). The error diagnosing process is shown in Figure 15.

Identify the error in the optimization trace. As shown in Figure 15(a), the two query plans are not equivalent after the *Optimization* step. Specifically, the *LeftAntiJoin* operator is missing in the query plan after the *Optimization* step. Obviously, there exists a *transformation error* during the query optimization.

Locate the root cause of the error. To investigate the root cause of this error, we explore the details of the *Optimization* step, which is illustrated in Figure 15(b). We find the *LeftAntiJoin* operator is removed at the last step, i.e., *PropagateEmptyRelation* step in the rightmost of Figure 15(d). This step is triggered as `Rel⟨empty⟩[id#4]` is empty, see (d2) in Figure 15(d). However, this empty relation is derived from the predicate `(id#4+1)=id#4` of the *Filter* operator in query plan (d1). Obviously, it is incorrect. We further verify the transformation logic of the predicate `(id#4+1)=id#4` by clicking the predicate of the node in query plan (d1), which is highlighted by the red rectangle in Figure 15(d). The purple curves show that the first appearance of the above predicate is in the query plan (c2) of Figure 15(c), which is transformed from query plan (c1) by applying *PushDownLeftSemiAntiJoin*.

A.2 Sub-optimal optimization. This issue is a *workflow error* reported on the GitHub repository of DuckDB v0.10.1 [6]. The developer reports that when running the SQL query shown in Figure 16(a) on the table created by the SQL shown in Figure 16(c), the execution time is extremely long. However, this error will not be triggered if a semantically equivalent query without a subquery is used (see Figure 16(b)).

Identify the error in the optimization trace. By checking the optimization trace visualization of two SQL queries, the developer finds that the when DuckDB v0.10.1 optimizes SQL in Figure 16(a), the optimized logical plan (d1) does not contain a *top_n* operator, while for SQL in Figure 16(b), a *top_n* operator is created (see plan (e1)). The error is detected in *Optimization* step.

Locate root cause of the error. The developer investigates the optimization trace of the fast query by expanding the *Optimization* step (see Figure 16(f)). S/he clicks the argument `100` of *LIMIT* operator to track the change, as shown in red rectangle in Figure 16(f). By clicking the right arrow button, the view scrolls to the next changed step of *LIMIT*. The visualization shows that the step *top_n* optimizes *LIMIT* and *ORDER_BY* operators to *TOP_N* correctly. However, in the visualization of the slow query, the developer finds that the *top_n* step is inefficient. After checking the source code, the developer confirms this step only optimizes adjacent *LIMIT* and *ORDER_BY* operators. The developer concludes it is a *workflow error* since another step, *limit_pushdown*, should be placed before this step, as implemented in the new version, DuckDB v1.0.0.

A.3 Improper Hint Elimination. This issue was first reported by Apache Spark users after an upgrade of Apache Spark at its issue tracker website [13]. We reproduce this issue by running the SQL query in Figure 17(c) on Apache Spark 3.3.0. The hint `"/*+ BROADCAST(t) */"` of the SQL query means broadcasting the table `t` during the execution of the *Join* operator.

Identify the error in the optimization trace. We first observe the query plan (a1) in Figure 17(a), which is semantically equivalent to the user-input SQL query. In particular, the hint to broadcast *t* in the SQL query is transformed to the node with a red rectangle in it. However, the hint is ignored in the query plan (a2), which broadcasts the table `s`, see the red rectangle in it. Thus, it is interesting to discover where the hint is removed during the query optimization. Fortunately, the transformation logic we find via the plan transformation problem can be used to highlight the changes of the hint in query plan (a1) among all query plans in the optimization trace. We find the hint is lost during the *Analysis* step as it does not have corresponding nodes in its plan after this step.

Locate the root cause of the error. To investigate the root cause of the hint elimination in the *Analysis* step, we explore the trace hierarchy step-by-step. As shown in Figure 17(b), the query plan (b2) has the hint node but the query plan (b3) does not have it. However, the *ResolveJoinStrategyHints* step transforms query plan (b2) to (b3). Hence, it is safe to conclude that the *ResolveJoinStrategyHints* in the new version of Spark does not resolve the hint in the SQL query correctly. With the above conclusion, the expert users probably can analyze the raw log to find the reason why the *ResolveJoinStrategyHints* step does not resolve the hint successfully.

B Data Preprocessing Layer of QOVIS

Exploring and analyzing the raw optimization log of the system is laborious and time-consuming. Moreover, the optimizers of different systems might differ in optimization workflow and optimization strategies, resulting in challenges while analyzing. In this section, we introduce the *data preprocessing layer* in the QOVIS, as shown at the bottom of Figure 4. It processes the raw log of different systems into unified data (i.e., plan sequence and step sequence) and facilitates further analysis. *Data preprocessing layer* consists of four procedures. (i) Extracting step and plan data from system logs; (ii) Handling sub-expressions (iii) Handling unrelated records; and (iv) Constructing optimization trace hierarchy. We detailed each procedure as follows.

B.1 Extracting step and plan data from logs. For generalization, QOVIS only requires the step sequence and plan sequence in the optimization trace log, which are generic and common in many optimizers. In this section, we first take Apache Spark as an example to show the data extraction process, then highlight the techniques used to generalize QOVIS on different systems.

Figure 18 shows a raw log example, which is collected from Apache Spark during query processing by the *data preprocessing layer* in QOVIS. The raw log is a sequence of records, and each record includes a timestamp and the content. Specifically, the gray rectangles in Figure 18 illustrate the records of two query plans, i.e.,

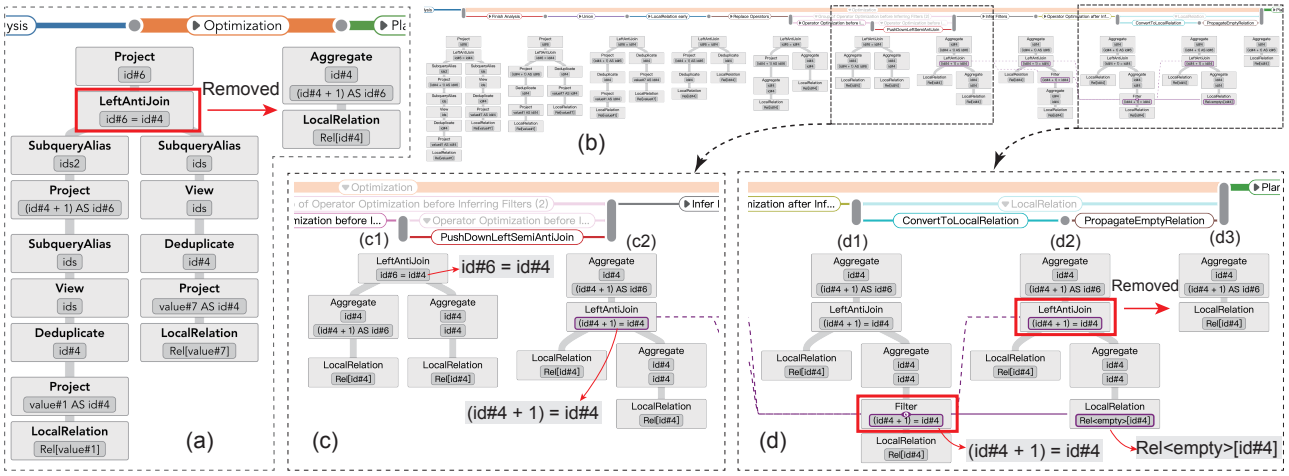

Figure 15: The diagnosis process for a *transformation error* (see Figure 2) of Apache Spark using QOVIS

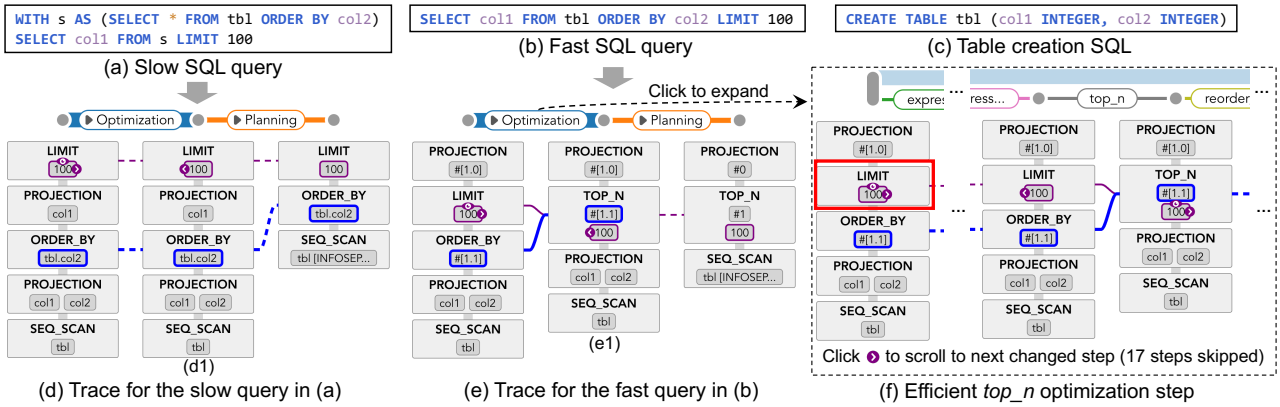

Figure 16: The diagnosis process for a *workflow error* of DuckDB using QOVIS, where a sub-optimal query plan is produced.

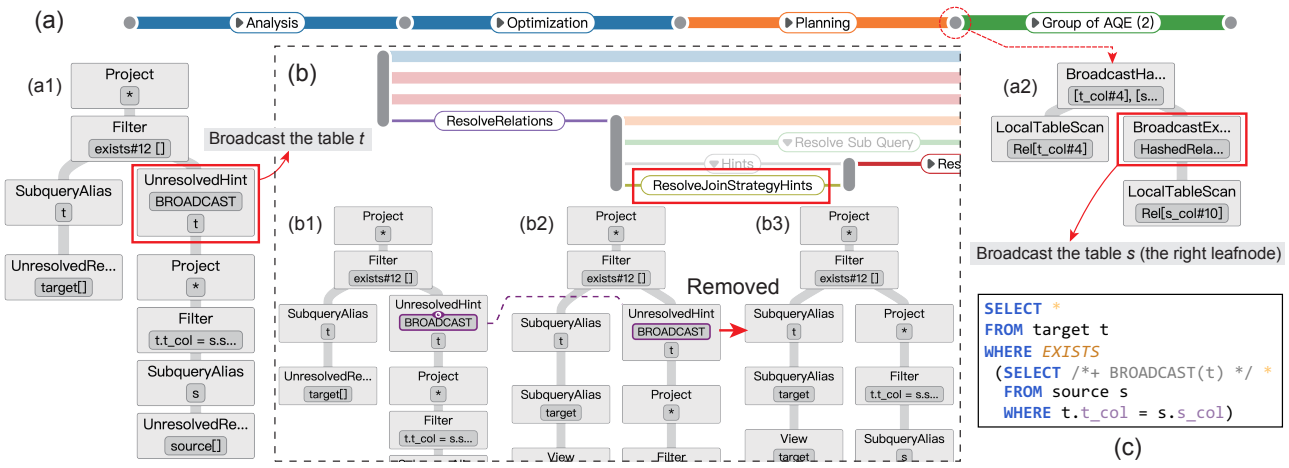

Figure 17: The diagnosis process for a transformation error in QOVIS, where the hint is eliminated

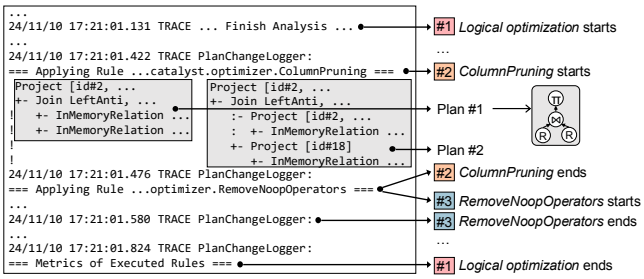

Figure 18: The raw optimization trace log in Apache Spark

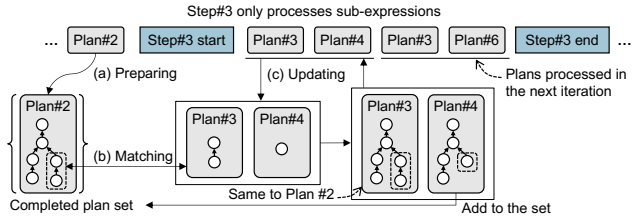

Figure 19: The procedure of sub-expressions processing

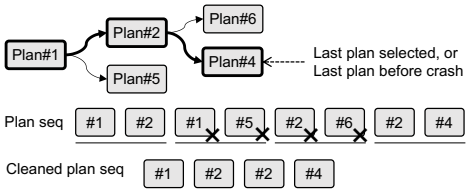

Figure 20: The procedure of unrelated record processing

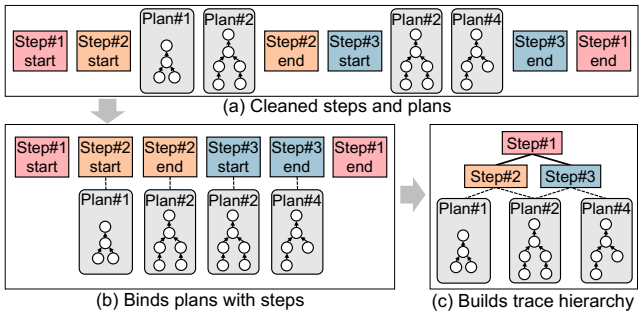

Figure 21: The procedure of trace hierarchy construction

Plan #1 and #2. The optimization step is defined by a pair of a start record and an end record. For example, the first line and the last line in Figure 18 show the starting and the ending of the *Logical optimization* step, respectively. Thus, all the contents that occurred between them belong to an optimization step: *Logical optimization*. In addition, an optimization step may include one or more other optimization steps. For example, the *Logical optimization* step includes optimization steps: *ColumnPrunning*, and *RemoveNoopOperators*, as shown in Figure 18.

In some systems, the log prints might not provide sufficient optimization data for analysis. However, the plans and steps can be obtained with a few modifications of the source code. For example, the optimizer of Apache Hive provided several hooks that are called before and after executing an optimization step. For the database systems that do not provide such hooks, such as PostgreSQL [10], patching and dumping these necessary data is laborious but still possible by adding log prints manually for each step. As long as the plan sequence and step sequence are collected, QOVIS can be employed for further analysis.

B.2 Handling sub-expressions. According to our observation, the collected plans might not always be a completed plan tree in some optimization steps, such as *Resolve Sub Query* step in Apache Spark and the steps using Cost Based Optimization (CBO) in Apache Hive. Instead, only the sub-expressions (i.e., sub-trees of the plans) are processed and recorded. For example, Plan #3 in the middle of Figure 19 is a sub-expression of the completed Plan #2 at the left. To obtain user-friendly and complete data, for each of these optimization steps we handle sub-expression data as follows.

- **Procedure 0: Preparing.** Before processing plans in the given step (e.g., Step #3 in Figure 19), we initialize a set of collected completed plans and add the initial plan (i.e., Plan #2) to the set, see Figure 19(a).
- **Procedure 1: Matching.** In each iteration, we process two adjacent sub-expressions sequentially. They are the input/output of a step (e.g., Plan #3 and Plan #4 in the middle of Figure 19). In the set, we find all matched plans that contain the input Plan #3. Here the result is Plan #2 (see Figure 19(b)).
- **Procedure 2: Updating.** As shown in Figure 19(c), we next convert two sub-expressions to completed plans using the matched plans (i.e., Plan #2). The result will be updated in the plan sequence, and the latter one, i.e., Plan #4 at the right, will be added to the plan set. Next, we go back to Procedure 1 if there are still unprocessed sub-expressions in this step.

After processing, all sub-expressions in the plan sequence are replaced to completed plans for further analysis.

B.3 Handling unrelated records. The optimizer might explore the plan space and print log records for the plans that are not selected as the final plan. QOVIS focus on the optimization trace of the final selected plans or the plans that result in crashes. Hence, we remove step and plan records that do not related to our target. As shown at the top of Figure 20, we build a tree for the plan exploration process, where each tree node is a unique plan and the edge represents a transformation. By traveling through the path from the root to the plan at the end of the sequence (i.e., Plan #4), we obtain the cleaned plan sequence (see the bottom of Figure 20) and the step sequence.

B.4 Constructing optimization trace hierarchy. The data produced in the last procedure (see Figure 21(a)) is still not user-friendly for analysis. Thus, we first bind the plans and steps, as shown in Figure 21(b). Then, according to the start time and the end time of steps, we build the hierarchy of steps. The resulting data structure is shown in Figure 21(c).