Qr-Hint: Actionable Hints and Edits for Query Debugging

ABSTRACT

Understanding the semantic difference between two SQL queries is a non-trivial task, especially when there may be errors in one or multiple '*clauses*' of the query like SELECT, FROM, WHERE, GROUP BY, or HAVING, and the correctness of one may affect the correctness of another clause. While there are tools that aim to show semantic differences between queries by providing users with sample concrete or abstract database instances, to our knowledge, less attention has been given to explaining how the way a query is written might cause the differences, and how a query can be '*fixed*'. This is particularly important in helping students or new programmers debug their SQL queries when extensive manual help may be difficult, e.g., in large database courses. In this paper, we describe a framework called Qr-Hint, which takes a (correct) reference query and a (wrong) user query as input (both single-block SELECT-FROM-WHERE-GROUP BY-HAVING queries under bag semantics, with no subqueries, outer joins, aggregates, NULLs, or database constraints), detects sources of errors in different clauses in the user query causing semantic difference with the reference query, and pinpoints what edits are needed in each clause in order for it to become equivalent to a reference query. We give algorithms to identify differences between corresponding SQL clauses of two queries and generate suggestions for fixes to convert the user query to the reference query, show their correctness, and evaluate them experimentally. We also show the effectiveness of our framework by presenting a user study in the settings of debugging SQL queries in educational setup.

1 INTRODUCTION

Identifying and understanding the semantic differences between two SQL queries is a key problem when learning, teaching, or debugging SQL queries. In particular, in a classroom setting, it is crucial to give students succinct but comprehensive information about the mistakes they have made, and also offer them advice or '*hints*' on how to resolve the mistakes without giving out the entire reference query. For this, instructors and teaching assistants in database courses often have to examine complex queries and locate mistakes in students' queries, which is a time consuming process especially for large classes. Further, in industry, developers or data science professionals might want to debug and compare updated SQL queries against their original counterparts when rewriting them for simplification or efficiency. Therefore, there is a need for recognizing the semantic differences between two SQL queries where the error may be in one or multiple SELECT, FROM, WHERE, GROUP BY, or HAVING clauses with mutual dependencies.

To this end, most prior work focuses on instance-based distinction between two queries [8, 13, 18, 28]. Specifically, these approaches generate concrete or abstract instances for which the queries give different result sets to illustrate the query difference. However, such approaches may incur high cognitive burden and may not be able to provide actionable suggestions to users who wish to detect the actual sources of errors that make their query different than the correct one and understand how to fix them.

Example 1. *Consider the following DBLP database schema [15] (keys are underlined):*

- `inproceedings: (pubkey, title, booktitle, year, area)`
- `authorship: (pubkey, author)`

The table `inproceedings` *stores conference papers, and* `authorship` *stores authorship information. The area in the inproceedings table can only be one of the following value: "AI-ML", "Database", "Systems", "Theory" and "UNKNOWN" (which can be extracted from some main conference names by pre-processing). Suppose we want to write a SQL query for the following problem:* **"For each author and each year, find the number of their co-authors from 'Database' papers except year 2018"** *(ignore the years they don't have any collaborators, so outer joins are not needed). Here is a correct query Q_1:*

```
SELECT t2.author, t1.year, COUNT(DISTINCT t3.author)
FROM inproceedings t1, authorship t2, authorship t3
WHERE t1.pubkey = t2.pubkey AND t3.pubkey = t1.pubkey
  AND t3.author <> t2.author AND t1.year <> 2018
  AND t1.area = 'Database'
GROUP BY t2.author, t1.year
```

Now consider the following (wrong) user query Q_2:

```
SELECT a2.author, year, COUNT(*)
FROM inproceedings i, authorship a1, authorship a2
WHERE (i.pubkey = a2.pubkey AND a1.pubkey = a2.pubkey
    AND a2.author <> a1.author AND i.year < 2018)
  OR (i.pubkey = a2.pubkey AND a1.pubkey = a2.pubkey
    AND a2.author <> a1.author AND i.year >= 2018)
GROUP BY a2.author, i.area, i.year, a1.author
HAVING i.area = 'Databases'
```

Here, our goal is to help the user identify mistakes in the query Q_2 and suggest fixes without revealing the correct query Q_1.

There are several standard approaches to detecting errors and debugging such wrong queries, that are frequently used in classroom settings. One can manually inspect the queries and evaluate its correctness, which is a time-consuming process. As a more efficient approach, one can execute the user queries on some carefully crafted input instances as test data [11, 28], and compare the answers. This approach does not check for query equivalence (which is undecidable in general [1]) and largely depends on the coverage of the test data to be able to detect different errors. On the other hand, when a user runs their query on a real input database, like the DBLP database that contains a large number of tuples, the user may lose track of the execution process when trying to debug their wrong query. Moreover, data is usually sensitive to the conditions as a subtle change in the query might cause massive differences in the output. For example, the wrong query in Example 1 will return an empty output simply due to the condition "`i.area = 'Databases'`", where the trailing "s" is redundant. There are other mistakes in the query, and it is hard to spot all of them at once by examining

the data because conditions and expressions in the query can potentially affect each other (e.g., the GROUP BY clause has an influence on the COUNT function in the SELECT clause). One obvious option to debug the query that does not need the data instance or manual inspection is if the user is given the correct reference query. Even then, understanding what is wrong in the user query is non-trivial. For instance, the aliases used for tables and attributes create extra cognitive burden, and such cognitive burden grows with the query size, preventing the user from promptly finding the errors in the query. Even when the reference query can be made available to the user, it may be important to help the user understand the errors in their queries (especially when points are deducted for wrong answers in a classroom setting). Hence, it is important to provide *succinct, accurate, and actionable 'hints' and 'edits' efficiently and automatically* about the parts of the wrong query that are causing semantic differences with a correct reference query.

Toward this goal, in this paper, we propose Qr-Hint, a framework that takes two non-equivalent SQL queries and pinpoints the parts of one query that cause semantic differences between the queries, and also provides hints for direct edits to convert one to the other. Instead of checking query equivalence, our approach relies on the structure of the given SQL queries, separating it into clauses. We consider the expressions/predicates appearing in the FROM, WHERE/HAVING, GROUP BY, and SELECT clauses. For each such SQL clause, we find the parts that need to be changed and the manner in which they should change for two queries to be equivalent. We show that Qr-Hint is able to detect query equivalence for select-project-join (SPJ) queries under some assumptions, i.e., simple SQL queries with single SELECT-FROM-WHERE with conjunction, disjunction, and negation under bag semantics (no outer joins, subqueries, aggregates, database constraints, NULLs), and it also covers GROUP-BY and HAVING clauses with limitations.

Example 2. *For the reference and user queries given in Example 1, our Qr-Hint framework returns the following suggested edits:*

- *"year >= 2018" → "year > 2018" (WHERE clause),*
- *"i.area = 'Databases' " → "i.area = 'Database' " (WHERE clause),*
- *Remove "a1.author" in the GROUP BY clause,*
- *"COUNT(*)" → "COUNT(DISTINCT a1.author)" (SELECT clause).*

Our framework considers the expression "i.area" in the GROUP BY clause as correct since it does not actually have any influence on the query output due to its constant value "i.area = 'Database' ".

Our Contributions. We make the following contributions.

- We develop a structural approach in the Qr-Hint framework for identifying which parts of a query cause semantic differences from a reference query, and propose edits to convert the query to the reference query.
- We formally define the concept of *repair sites* and *fixes* for each SQL clause, where replacing the repair sites with their corresponding fixes make the clause equivalent.
- We give algorithms to locate repair sites and determine fixes in each clause, and show how to use the repair sites and fixes for hints. We study the theoretical properties for repairs and provide guarantees for our algorithms. We show necessary conditions for the existence of repairs, which guide our algorithms in the search for repairs. In particular, we show how fixing one clause can make expressions/conditions in another clause correct, even if they appear different. Additionally, for WHERE/HAVING clauses, we show that our algorithms always find a repair. Moreover, we devise optimizations that generate smaller and more targeted fixes.
- We evaluate the performance and efficacy of Qr-Hint experimentally. We further perform a user study involving students from current/past database courses. Our findings indicate that the hints generated by Qr-Hint are close to the optimal repairs and are helpful for students.

2 RELATED WORK

Query debugging with reference queries. As for automated grading of SQL queries in classroom setting, XData [11] checks the correctness of a query by running the query on self-generated testing datasets based on a set of pre-defined common errors that usually occur in SQL queries. Furthermore, Chandra et. al. [10] takes a further step by developing a grading system that awards partial credits to wrong queries. The system first canonicalizes a wrong query and the reference query by applying a set of rewrite rules, and then decides partial credits based on a tree-edit distance between the logical plans. For debugging SQL queries, QueryVis [26] allows users to examine SQL queries by generating designated diagrams that capture the relationship among operators, helping users understand the difference between (non-monotone) queries with nested sub-queries through execution flows. QueryVis guarantees that distinct queries always yield distinct diagrams. Out of a large testing database instance, The RATest system [28] utilizes data provenance to generate a small illustrative instance to differentiate two queries from a large test data instance. Gilad et. al. [18] propose an approach using c-tables [22] to form small abstract conditional instances, which describe the general conditions in the input data that can differentiate two queries. SQLRepair [29] is an automated tool that is capable of fixing errors in a Select-Project-Join (SPJ) query, where it first repairs a set of common syntax error, and later relies on the Z3 SMT Solver [16] to synthesize/remove WHERE conditions until the wrong query produces the same output as the reference query over all testing database instances.

Testing query equivalence. There are several classical results on query equivalence. Chandra et. al. [9] show that equivalence testing of conjunctive queries is NP-complete, which has been later studied in other seminal works for different query classes [3, 23–25, 30]. While the query equivalence problem in general is known to be undecidable [1, 32], tools and algorithms are still developed to check the equivalence of various classes of queries with restrictions and assumptions. More recently, Cosette [13] transforms SQL queries to algebraic expressions called K-relations [14], and uses a decision procedure and rewrite rules to check if the resulting expressions of two queries are equivalent. As an extension of Cosette, UDP [12] covers a wider range of SQL queries under bag semantics by defining a new algebraic structure named U-semiring, which also takes database constraints into account when verifying the equivalence of two SQL queries. EQUITAS [35] develops a symbolic representation of SQL queries, which can be later transformed into first-order logic formulas, and uses satisfiability modulo theories (SMT) to check query equivalence. Finally, WeTune [34] also builds a query equivalence verifier by utilizing the U-semiring structure,

which is translated into first-order logic formulas whose equivalence can be verified by SMT solvers. There is rich literature on debugging, repairing, and giving feedback for general programs, e.g., [2, 7, 19–21, 31, 33], which we do not discuss in detail as we focus on declarative SQL queries. To the best of our knowledge, our work is the first in providing targeted hints and edits in a wrong SQL query with respect to a reference query to make the two queries equivalent without requiring a test database as input or output.

3 THE QR-HINT FRAMEWORK

Our Qr-Hint framework takes two queries as input: a reference query Q_1 (a correct query) and a user query Q_2 (a wrong query). It diagnoses the differences between two queries and proposes edits to the user query to make it equivalent to the reference query.

Supported SQL Queries. We consider (syntactically correct) single-block SQL queries that contain single SELECT and FROM clauses, and optionally single WHERE, GROUP BY, and HAVING clauses. We assume there are no subqueries or outer joins.

Schema and data. We consider a relational schema $\mathbf{R}$ that contains a set of relations (tables) $(R_1, \ldots, R_r)$; we use R_i to denote both the relation name as well as a relation instance. We consider bag semantics, i.e., in an instance of the database, an input relation R_i or the output can contain duplicates. Further, we assume the data instances do not have NULLs, and there are no database constraints (note that database constraints, e.g., foreign key constraints, can make two seemingly different queries equivalent).

3.1 FROM Clause Repair

Table mapping. Recall that in SQL, a FROM clause may reference a table T multiple times, and each reference is associated with a distinct alias (optional, defaults to the name of T). Assuming the query is valid, each column reference must resolve to exactly one of these aliases. Let $\mathit{Tables}(Q)$ denote the multiset of tables in the FROM clause of Q, and let $\mathit{Aliases}(Q)$ denote the set of aliases they are associated with in Q. With a slight abuse of notation, given table T, let $\mathit{Aliases}(Q, T)$ denote the subset of $\mathit{Aliases}(Q)$ associated with T (a non-singleton $\mathit{Aliases}(Q, T)$ would imply a self-join involving T). Given an alias $t \in \mathit{Aliases}(Q)$, let $\mathit{Table}(Q, t)$ denote the table that t is associated with in Q.

Definition 1. *Given queries Q_1 and Q_2 over the same schema where $\mathit{Tables}(Q_1) = \mathit{Tables}(Q_2)$ (multiset equality), a table mapping from Q_1 to Q_2 is a bijective function $\mathfrak{m} : \mathit{Aliases}(Q_1) \to \mathit{Aliases}(Q_2)$ with the property that two corresponding aliases are always associated with the same table, i.e., $\forall t \in \mathit{Aliases}(Q_1) : \mathit{Table}(Q_1, t) = \mathit{Table}(Q_2, \mathfrak{m}(t))$.*

With the help of such a table mapping $\mathfrak{m}$ from Q_1 to Q_2, we can "unify" Q_1 and Q_2 such that all table and attribute references use the same set of table aliases. Once we have the table mapping, a mapping for attributes follows naturally. The goal of the FROM clause repair problem is to make the multisets of two queries the same, and find a valid table mapping (see Section 3.5).

Example 3. *In Example 1, there are two possible table mappings $\mathfrak{m}_1$ and $\mathfrak{m}_2$ from Q_2 to Q_1, namely:*

(1) $\mathfrak{m}_1(i) = t1, \mathfrak{m}_1(a1) = t2, \mathfrak{m}_1(a2) = t3$.
(2) $\mathfrak{m}_2(i) = t1, \mathfrak{m}_2(a1) = t3, \mathfrak{m}_2(a2) = t2$.

Note that it is possible for two single-block SQL queries to be equivalent even if they do not have the same multi-set of tables in the FROM clause. For instance, consider two queries:

```
Q1: SELECT * FROM R, R WHERE FALSE
Q2: SELECT * FROM R, R, R, R WHERE FALSE
```

Both these queries return empty answers. However, we show that under the assumption that both the queries are not equivalent to FALSE, the assumption of same multiset is required for two single-block queries to be equivalent:

Proposition 1. *Two single-block SQL queries Q_1 and Q_2 cannot be equivalent under bag semantics if $\mathit{Tables}(Q_1) \neq \mathit{Tables}(Q_2)$ (multiset inequality) assuming no database constraints are present, and there exists some database instance for which either Q_1 or Q_2 returns a non-empty result.*

The proof takes one of the queries that returns non-empty result on some database instance D, and multiplies occurrence of tuples in each table by a factor that is a prime number, choosing different primes of different tables, and uses the Prime Factorization Theorem [17] to argue that with two different multi-sets of tables in the FROM clauses, Q_1, Q_2 can not be equivalent.

3.2 WHERE/HAVING Clause Repair

Repairing the predicate in WHERE/HAVING clauses is the most non-trivial component of Qr-Hint. We discuss repairs for the WHERE clause, HAVING is treated similarly, and along with WHERE clause for correctness (Section 4.4). We assume standard SQL logical formulas for the predicate in WHERE (involving $AND, OR, <, =, >$, $LIKE$ etc.), and denote the predicates in Q_1, Q_2 as P_1, P_2 respectively. Ideally we want to change small parts in P_2 so that the hints can be as close to the original user query Q_2, therefore, we give a tree model for the predicate P_1 or P_2.

Definition 2 (Syntax Tree for SQL Logical Formula). *Given a SQL logical formula P, the syntax tree of P is a binary tree satisfying the following conditions:*

- *All internal (non-leaf) nodes $\in \{\wedge, \vee, \neg\}$.*
- *All leaf nodes are atomic predicates (e.g., $X > 5, X > Y$, etc., i.e., they do not contain any of $\{\wedge, \vee, \neg\}$.)*
- *All internal nodes $\in \{\wedge, \vee\}$ have exactly two subtrees, and internal nodes $\in \{\neg\}$ have one subtree.*

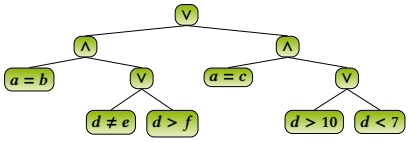
Given an SQL logical formula P, we can construct a unique syntax tree (also denoted by P) by following a fixed order while parsing associative logical connectives with appropriate parentheses (e.g., $p_1 \vee p_2 \vee p_3$ is always parsed as $(p_1 \vee p_2) \vee p_3$)).

Example 4. *Consider the following SQL logical formulae P_1 and P_2 where a, b, c, d, e are integers:*

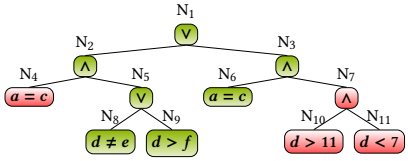
P_1: $(a = b \wedge (d \neq e \vee d > f)) \vee (a = c \wedge (d > 10 \vee d < 7))$
P_2: $(a = c \wedge (d \neq e \vee d > f)) \vee (a = c \wedge (d > 11 \wedge d < 7))$

Their respective syntax trees are shown in Figure 1.

Definition 3 (Repair Site and Fix for WHERE clause). *Given two non-equivalent SQL logical formulae P_1 and P_2, the* **repair sites for the WHERE clause of** *P_2 are n disjoint sub-trees in P_2, and the corresponding* **fixes** *are n syntax trees, such that replacing s_i with f_i $(1 \leq i \leq n)$ achieves $P_1 \equiv P_2$.*



(a) Syntax Tree of P_1



(b) Syntax Tree of P_2

Figure 1: Syntax tree of P_1 and P_2 in Example 5

The goal of the WHERE repair problem is to find repair sites and fixes such that $P_1 \equiv P_2$. Note that the entire formula P_2 can always be replaced with P_1 so that $P_2 \equiv P_1$. However, while proposing fixes to P_2, we want to return *small fixes* (e.g. typos, wrong logical operator, etc.), possibly identifying multiple concurrent fixes, so that more precise sources of errors can be identified (Section 4).

Example 5. *Consider the syntax trees shown in Figure 1 for P_1 and P_2. If the number of repair sites in P_2 is limited to at most 2, then the repair sites are $S = \{N_4, N_7\}$, and corresponding fixes F_S are $(a = b)$ and $(d > 10 \vee d < 7)$ respectively; replacing the sub-formulas at N_4 and N_7 with these fixes will make P_2 equivalent to P_1.*

When the number of allowed repair sites is 1, the only possible set of repair site in P_2 is N_1, and the fix is replacing it with the entire logical formula corresponding to P_1.

When the number of sites is 3, a set of repair sites in P_2 is $S = \{N_4, N_{10}, N_{11}\}$ and a set of corresponding fixes F_S is $(a = b)$, $(d > 10 \vee d < 7)$, and $(d > 10 \vee d < 7)$, which makes P_2 equivalent to P_1.

When the number of sites is 4, a set of repair sites in P_2 is $S = \{N_4, N_9, N_{10}, N_{11}\}$ and a set of corresponding fixes F_S is $(a = b)$, $(d > f)$, $(d > 10 \vee d < 7)$, $(d > 10 \vee d < 7)$. Here N_9 is redundant, i.e., our algorithm can return false negatives, which can be removed by post-processing from the proposed hints. Also note that fixes are always a sub-formula, so we do not change the $\wedge$ in N_7 with an $\vee$.

3.3 GROUP BY Clause Repair

We assume a GROUP BY clause can contain attributes as well as expressions on attributes (see Example 6 below). Two tuples (from the cross product of tables appearing in the FROM clause and then applying the predicate in the WHERE clause) belong to the same group if they share the same values for all attributes and expressions appearing in the GROUP BY clause.

Definition 4 (Equivalence of GROUP-BY Clauses). *Given two single-block SQL queries Q_1 and Q_2 whose FROM and WHERE clauses are equivalent, if any two tuples from the joined tables (from the FROM and WHERE clauses) belong to the same group in Q_1, they also belong to the same group in Q_2 and vice-versa.*

Example 6. *Consider the problem given in Example 1, its reference query Q_1, and a new user query Q_2 with the same aliases for corresponding tables for easier comparison.*

```
Q2:
SELECT t2.author, t1.year - 2018, COUNT(*)
FROM inproceedings t1, authorship t2, authorship t3
WHERE t1.pubkey = t2.pubkey AND t3.pubkey = t1.pubkey
  AND t2.author <> t3.author AND t1.year < 2018
  AND t1.area = 'Database'
GROUP BY t2.author, t1.area, t1.year - 2018, t3.author;
```

The user query Q_2 has the equivalent FROM and WHERE clauses as the reference query, but has a mistake in the GROUP BY clause (also in the SELECT clause). Although the GROUP BY clauses of Q_1, Q_2 look very different, not all attributes in the GROUP BY clause of Q_2 are problematic. For instance, $t1.area$ does not cause an error because it is set to a constant 'Database' by the WHERE clauses and thus has no effect. Similarly, $t1.year - 2018$ is also fine because for any two tuples fall into the same group in Q_1 (i.e. their value for $t1.year$ are equal), they must also fall into the same group in Q_2 (i.e. their value for $t1.year - 2018$ are equal). Hence if we had a version of Q_2 with the following GROUP BY clause it would be equivalent to that of Q_1.

```
GROUP BY t2.author, t1.area, t1.year - 2018;
```

The mistake in the GROUP BY clause of Q_2 is due to the expression $t3.author$ (which further splits the correct grouping). For instance, consider two tuples from the implicit joined tables in Q_1 and Q_2 before GROUP BY is applied:

t1.pubkey	t1.year	t1.area	t2.pubkey	t2.author	t3.pubkey	t3.author
BrinMUT97	*1997*	*Database*	*BrinMUT97*	*J. Ullman*	*BrinMUT97*	*S. Brin*
BrinMUT97	*1997*	*Database*	*BrinMUT97*	*J. Ullman*	*BrinMUT97*	*S. Tsur*

These two tuples fall into the same group in Q_1 but not Q_2. If we remove $t3.author$ from the GROUP BY clause in Q_2, the GROUP BY clauses of Q_1 and Q_2 become equivalent.

The goal of the GROUP BY repair problem is to find and replace the expressions in the GROUP-BY clause of Q_2 to make it equivalent to the GROUP BY clause of Q_1. We define the repair sites and fixes for GROUP BY clause in Section 5, and describe our solution.

3.4 SELECT Clause Repair

A valid SELECT clause in a SQL query can consist of variables (attribute names), constants, functions involving $+, -, *, /$, string concatenation function ||, and also aggregate functions like COUNT/SUM/AVG. We assume the output columns in the SELECT clause as an ordered set to define equivalence of two SELECT clauses[1]; we describe our solution in Section 6.

Definition 5 (Equivalence of SELECT Clauses). *Given two queries Q_1, Q_2, let $S_1 = (e_1^1, ..., e_n^1)$ and $S_2 = (e_1^2, ..., e_m^2)$ denote the ordered set of expressions (output columns) in their respective SELECT clauses. The SELECT clauses of Q_1 and Q_2 are equivalent if $m = n$ and $e_i^1 \equiv e_i^2$, $1 \le i \le n$ (given the equivalence of all other clauses).*

3.5 Overview and Steps of Qr-Hint

Qr-Hint compares the reference query Q_1 and user query Q_2 in the order of FROM → WHERE/HAVING → GROUP BY → SELECT. For each clause, Qr-Hint decides the edits on Q_2 so that it becomes equivalent to the one in Q_1, assuming the equivalence of the previous steps. In the end, all edits are reported to the user.

Step 1: FROM Clause Repair. Qr-Hint first checks if both queries share the same multiset of tables in their FROM clauses. Otherwise,

[1]We assume that the same order of output columns in the reference and user queries are required, although in some applications the order of the columns may not matter.

it suggests adding/removing necessary tables to make the FROM clause of Q_2 equivalent to that of Q_1.

Step 2: Decide Table Mapping. After ensuring both FROM clauses have the same multiset of tables, Qr-Hint decides a table mapping. In order to follow the user's intention as much as possible, it goes through each expression in the SELECT clause, tries to match the attributes in each expression from both queries, and maps their designated tables to each other. For instance, in Example 1, it detects a plausible mapping of "a2.author" to "t2.author" and "year" to "t1.year", and thus maps "inproceedings i" to "inproceedings t1" and "authorship a2" to "authorship t3" (i.e., it goes for $\mathfrak{m}_1$ in Example 3 rather than $\mathfrak{m}_2$). In the cases where it cannot find a match for the attributes in a SELECT expression (e.g. "COUNT(*)" and "COUNT(DISTINCT t3.author)" in Example 1), it simply ignores the current expression and moves to the next one. If such process for determining a mapping does not converge to a single mapping at the end, it randomly picks a table mapping left in the pool and proceeds to the next step[2]. It rewrites the reference and user queries with consistent aliases for the next steps from the chosen mapping.

Step 3. WHERE/HAVING Clause. Qr-Hint tests the WHERE and HAVING clauses together, since one may affect the correctness of the other (see Example 1). We model both WHERE and HAVING clauses as logical formulas and syntax trees, and test the equivalence between the formulas from both queries. If the equivalence holds, we declare the WHERE/HAVING clauses from Q_1, Q_2 as equivalent. If such equivalence does not hold, we report a set of sub-formulas in the WHERE/HAVING formulas of the user query (repair sites), and propose a corresponding set of formulas for replacement (fixes) so that two clauses become equivalent (see Sections 4).

Step 4. GROUP BY Clause. Qr-Hint checks the equivalence of GROUP BY clauses based on equivalent WHERE/HAVING clauses, and reports missing/incorrect expressions; e.g., consider a Q_2 modifying Q_2 in Example 6, where the WHERE condition includes `t2.author = t3.author`. Then grouping either by `t2.author` or by `t3.author` will return the same groups (see Section 5).

Step 5. SELECT Clause. Finally, Qr-Hint tests the equivalence of SELECT clauses assuming other equivalent clauses. Unlike GROUP BY clauses, however, the order of the expression matters in the SELECT clause. Thus, we can identify differences between two SELECT clause simply by checking the corresponding expressions from both queries at the same positions. It reports any mismatch of expressions in the SELECT clause (see Section 6).

4 HANDLING WHERE AND HAVING CLAUSE

Finding repair sites and fixes in the WHERE and HAVING clause is the most non-trivial component of the Qr-Hint framework; it aims to give edit suggestions for focused repair sites, as opposed to replacing the entire formula. Here we primarily discuss the WHERE clause; HAVING is briefly discussed in Section 4.4.

4.1 Tool Set for the Algorithms

We start by defining the concept of lower-bound and upper-bound of a SQL logical formula, which is used later in the algorithms to verify whether a set of disjoint sub-trees is a set of repair sites. For two Boolean formulas f_1, f_2, $f_1 \implies f_2$ denotes logical implication, i.e., if f_1 is true, f_2 must be true.

Definition 6 (Upper-bound and Lower-bound of SQL Logical Formula). *Given three SQL logical formulas $P_\perp$, P and $P_\top$, if $P_\perp \implies P$ and $P \implies P_\top$ hold, we say that $P_\top$ is an* **upper-bound** *of P and $P_\perp$ is a* **lower-bound** *of P. Equivalently, we write, $P \in [P_\perp, P_\top]$. In general, we use $\perp, \top$ as the subscripts to denote the lower-bound and upper-bound of an SQL logical formula.*

For this problem, we are given two SQL queries Q_1 and Q_2 that have already passed the FROM check, as well as a table mapping $\mathfrak{m}$ from Q_1 to Q_2; we need to detect whether evaluating their WHERE conditions can lead to different multisets of joined tuples (ignoring column ordering induced by the ordering of tables in FROM), and if yes, find a "*low-cost repair*" on Q_2's WHERE condition to make it equivalent to that of Q_1. Since we are given the mapping $\mathfrak{m}$, we can map each column reference in Q_2's WHERE condition to a corresponding column reference in Q_1, leading to a WHERE condition that reference the same set of columns as Q_1's WHERE condition. By treating each distinct column reference as a variable, the problem now reduces to the following:

- *Given two Boolean formulas P and $P^\star$ as syntax trees (Definition 2), find a "low-cost repair" on P that makes it $\equiv P^\star$.*

We now formalize the problem, together with the notions of repairs and costs. Recall from Definition 3 that a *repair* (S, F) on a Boolean formula P consists of a set S of *repair sites*, which are disjoint sub-trees of P (denoted by their root nodes), and the corresponding *fixes*, where for each sub-tree $s \in S$, $F(s)$ specifies a new Boolean formula to replace s with (denoted by $(s \to F(s))$). The *succinctness cost*, or simply the *cost*, of repair (S, F) is defined as:

$$\text{Cost}(S, F) = w_1 \cdot |S| + w_2 \cdot \sum_{s \in S} \texttt{dist}(s, F(s)) \tag{1}$$

where $w_1, w_2 \in \mathbb{R}^+$, and $\texttt{dist}(s, F(s)) \geq 0$ is function that measures the cost of replacing s with $F(s)$. The problem of finding the lowest-cost repair on a Boolean formula can be then stated as follows:

Given a Boolean formula P and target Boolean formula $P^\star$, find the repair (S, F) on P with minimum Cost(S, F) such that applying (S, F) to P results in a formula logically equivalent to $P^\star$.

Remark. A trivial repair is $(\{P\}, \{(P \to P^\star)\})$, which amounts to hinting the user that the entire WHERE condition is wrong and needs to be fixed. While valid, this repair may carry a high cost. Preferably, we want to find a single small sub-tree in P that can be replaced with a new small sub-tree to make P equivalent to $P^\star$. In situations where P contain multiple mistakes, a multi-site repair may be appropriate, but in practice, suggesting more than two sites tends to make the hints more difficult for users to reason with, because all sites must be fixed simultaneously to achieve equivalence to $P^\star$. Therefore, in the definition of Cost, the term $w_1 \cdot |S|$ penalizes a large number of repair sites. We define $\texttt{dist}(s, F(s))$ as the total number of nodes in s and $F(s)$ (other formulations are possible, but we found this one to work well in practice). In our experiments (Section 7), we set $w_1 = \frac{size(P)+size(P^*)}{2} \times \frac{1}{3}$, where $size(P)$ denotes the number of nodes of the syntax tree of P. The intuition is to add a penalty for the number of repair sites based on the query size – if the query size is large, more repair sites

[2] A more expensive alternative is to exhaustively check all possible valid mappings.

Algorithm 1 Find-Repair-Sites-and-Fixes

FIND-OPTIMAL-REPAIR-SITES-AND-FIXES(P_1, P_2, n)

Input: P_1, P_2 (as syntax trees), n: maximum number of repair sites
Output: Optimal repair sites and fixes

```
repair_sites = None; fixes = None; cost = ∞; S = all nodes in P2;
for i ∈ [1, n]:
    all_sets = List-All-Disjoint-Sets(S, i)
    Sort-on-Size(all_sets)
    for s ∈ all_sets:
        if Calculate-Cost(P1, P2, i, s, ∅) > cost:
            break
        L, U = Create-Bounds(P2, s)
        if P1 ∈ [L, U]:
            F = Derive-Fixes(L, U, s)
            cur_cost = Calculate-Cost(P1, P2, i, s, F)
            if cur_cost < cost:
                repair_sites = s; fixes = F; cost = cur_cost

return (repair_sites, fixes)
```

will significantly affect the cost of repair. We also set $w_2 = 1$ and (typically) set the number of repair sites to 2; in general, these numbers can be chosen by the users.

Example 7. *Consider the repair sites and fixes listed in Example 5. With $size(P_1) = 11$ and $size(P_2) = 11$ (Figure 1), the cost for each case is as followed, where $n = |S|$ is the number of repair sites.*

- *When $n = 1$, the cost is $\frac{22}{2} \times \frac{1}{3} \times 1 + (11 + 11) = 25.67$.*
- *When $n = 2$, the cost is $\frac{22}{2} \times \frac{1}{3} \times 2 + (1 + 1) + (3 + 3) = 15.33$.*
- *When $n = 3$, the cost is $\frac{22}{2} \times \frac{1}{3} \times 3 + (1+1) + (1+3) + (1+3) = 21$.*
- *When $n = 4$, the cost is $\frac{22}{2} \times \frac{1}{3} \times 4 + (1 + 1) + (1 + 1) + (1 + 3) + (1 + 3) = 26.67$*

For instance, for $n = 2$, the repair sites are N_4 (original size 1, fix size 1) and N_7 (original size 3, fix size 3). Among all cases listed, $n = 2$ has the minimum cost. For the cases where there exist redundant repair sites (e.g., for $n = 4$, the repair site N_9 is redundant as it is replaced with itself), its cost is much greater than cases carry fewer redundancies (e.g. $n = 2$ and $n = 3$). The case $n = 1$ has a high cost because it requires replacing the entire P_2 with P_1.

4.2 Finding Repair Sites and Fixes

First we give an outline of our approach:
(1) Algorithm 1 describes our overall procedure for finding repair sites, finding their upper and lower bounds and fixing them. Since we have no information about which parts of the formula causes the semantic differences, we iterate through all possible sets of disjoint sub-trees (up to n sub-trees as given as an input parameter) and determine if each is a valid set of repair sites. We show the correctness of our algorithm (Theorem 1). In the worst case, we return the entire reference formula as the fix.
(2) Next, we describe a subroutine of Algorithm 1 - the generation process of the bounds for each set of repair sites (Algorithm 2). We show how to determine if a set of disjoint sub-trees are valid repair sites. i.e., if it is possible to fix the formula by fixing a particular set of disjoint sub-trees (Proposition 4.1).
(3) Once a valid set of repair sites is found, we show how another subroutine of Algorithm 1 uses the synthesized lower-bound and upper-bound from the previous step to derive a fix for each repair site (Algorithm 3). We show that it is able to generate such fixes, given correct bounds (Proposition 4.2). The following theorem is the main theorem of this section, and shows the correctness of Algorithm 1; we defer the proofs to the full version [4].

Algorithm 2 Create-Bounds

CREATE-BOUNDS(P, S)

Input: P: syntax tree of a logical formula, S: a set of disjoint sub-trees in P
Output: lower-bound, upper-bound w.r.t. S

```
if P ∈ S:
    P.lower, P.upper = ⊥, ⊤
    return P.lower, P.upper
elseif P.children is empty:
    P.lower, P.upper = P, P
    return P.lower, P.upper
final_lower, final_upper = None, None
if P.op ∈ {∨, ∧}:
    left_lower, left_upper = Create-Bounds(P.left, S)
    right_lower, right_upper = Create-Bounds(P.right, S)
    final_lower = left_lower + P.op + right_lower
    final_upper = left_upper + P.op + right_upper
elseif P.op ∈ {¬}:
    child_lower, child_upper = Create-Bounds(P.child, S)
    final_lower, final_upper = ¬(child_lower), ¬(child_upper)
P.lower, P.upper = final_lower, final_upper
return P.lower, P.upper
```

Theorem 1 (Correctness of Algorithm 1). *Given two SQL logical formulas P_1 and P_2 s.t. $P_1 \not\equiv P_2$, Algorithm 1 returns a valid fix F for a set of repair sites S in P_2, i.e., if these repair sites are replaced with their fixes from F, then P_2 becomes equivalent to P_1.*

Creating the bounds for repair sites. Given a Boolean formula P and a set S of repair sites, suppose we are allowed to edit each site at will, how much effect can we achieve on the whole formula P? We can replace each sub-tree in S with something as tight as $\bot$ or as loose as $\top$, but in the end the effect of such edits on P may be limited by what lies outside S in P. The CREATE-BOUNDS function computes the range of possibilities that can be achieved (for each node in P_2 in a bottom-up fashion), when we are limited to editing S. The pseudo code is given in Algorithm 2.

Example 8. *Continuing the case of $n = 2$ in Example 5 with P_2 syntax tree in Figure 1, we can replace node N_4 with $\bot$ or $\top$ and obtain the bound $a = c \in [\bot, \top]$. On the other hand, both lower-bound and upper-bound for N_5 are $d \neq e \vee d > f$ since N_5 does not contain any incorrect sub-tree. Therefore, the lower-bound and upper-bound for N_2 are formed by combining the lower-bounds and upper-bounds of N_4 and N_5: $[\bot \wedge (d \neq e \vee d > f), \top \wedge (d \neq e \vee d > f)]$, and it can be checked that $a = c \wedge (d \neq e \vee d > f) \in [\bot \wedge (d \neq e \vee d > f), \top \wedge (d \neq e \vee d > f)]$. Similarly, the lower-bound and upper-bound for N_3 is $[a = c \wedge \bot, a = c \wedge \top]$. Subsequently, the lower-bound and upper-bound for N_1 is $[(\bot \wedge (d \neq e \vee d > f)) \vee (a = c \wedge \bot), (\top \wedge (d \neq e \vee d > f)) \vee (a = c \wedge \top)]$, and $P_1 \in [\bot, (d \neq e \vee d > f) \vee a = c]$ after simplification.*

The following lemma shows that the bounds computed by CREATE-BOUNDS are valid.

Algorithm 3 Derive-Fixes

DERIVE-FIX($S, P, T_\perp, T_\top$)

Input: S: a set of repair sites, P: current root node syntax tree of the incorrect SQL logical formula with pre-computed bounds w.r.t. S, $T_\perp$: current target lower-bound, $T_\top$: current target upper-bound,
Output: a set of pair of (repair site, fix)

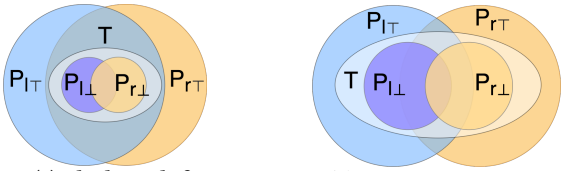
1 **if** $P \in S$:
2 T = Minimize-Fix($T_\perp, T_\top$)
3 **return** [(P, T)]
4 **elseif** $P.lower \equiv P.upper$:
5 **return** [ ]
6 F = [ ]
7 **if** P.op $\in \{\wedge, \vee\}$:
8 P_l, P_r = P.left, P.right
9 $P_{l\perp}, P_{l\top} = P_l$.lower, P_l.upper
10 $P_{r\perp}, P_{r\top} = P_r$.lower, P_r.upper
11 **if** P.op $\in \{\wedge\}$:
12 left_bounds = $[T_\perp \vee P_{l\perp},\ (T_\top \vee P_{l\perp} \vee \neg P_{r\top}) \wedge P_{l\top}]$
13 right_bounds = $[T_\perp \vee P_{r\perp},\ (T_\top \vee P_{r\perp} \vee \neg P_{l\top}) \wedge P_{r\top}]$
14 F += Derive-Fixes(S, P_l, left_bounds[0], left_bounds[1])
15 F += Derive-Fixes(S, P_r, right_bounds[0], right_bounds[1])
16 **elseif** P.op $\in \{\vee\}$:
17 left_bounds = $[(T_\perp \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp},\ T_\top \wedge P_{l\top}]$
18 right_bounds = $[(T_\perp \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp},\ T_\top \wedge P_{r\top}]$
19 F += Derive-Fixes(S, P_l, left_bounds[0], left_bounds[1])
20 F += Derive-Fixes(S, P_r, right_bounds[0], right_bounds[1])
21 **elseif** P.op $\in \{\neg\}$:
22 F += Derive-Fixes(S, P.child, $\neg P_\top, \neg P_\perp$)
23 **return** F

LEMMA 4.1 (VALIDITY OF BOUNDS COMPUTED BY ALGORITHM 2). *Given a Boolean formula P and a set S of repair sites, Algorithm 2 outputs two formulas $P_\perp$ and $P_\top$, such that applying any repair (S, F) (with the given S) will result in a formula $P' \in [P_\perp, P_\top]$.*

This lemma immediately yields a method that helps us decide whether a candidate set S of repair sites is viable: if the target Boolean predicate $P^\star$ falls outside the bounds computed given S, then there is no way to fix P through S so it becomes equivalent to $P^\star$. We will further show later that if $P^\star$ falls within this bound, then there indeed exists a way to fix P.

Deriving fixes for repair sites. Suppose the target formula $P^\star$ falls within the bounds computed by CREATE-BOUNDS(P, S). We now give a procedure, DERIVE-FIXES, to compute fixes F such that applying (S, F) to P yields a formula equivalent to $P^\star$. The procedure works by traversing P top-down, starting with a target bound of $[P^\star, P^\star]$ for the root and deriving a target bound for each sub-tree (with the help from the sub-tree bounds previously computed by CREATE-BOUNDS). Once the procedure reaches a sub-tree in S, it suffices to replace the sub-tree with a formula chosen from within the derived target bound for this sub-tree to achieve the overall repair goal.

EXAMPLE 9. *Considering the syntax tree of P_2 in Example 5, the target formula T at its root node N_1 is P_1 since we want to make P_2 equivalent to P_1 and T is composed by both the target formula at N_2 and N_3, we want to find the target formula bounds from which we can choose target formulas T_{N_2} and T_{N_3}, respectively so that $T_{N_2} \vee T_{N_3} \equiv T \equiv P_1$. In particular, if we replace N_4 with $a = b$ and N_7 with $d > 10 \vee d < 7$, we would get $T_{N_2} \vee T_{N_3} \equiv T \equiv P_1$.*



(a) The bounds for AND **(b) The bounds for OR**

Figure 2: Venn Diagrams for the derived target formula bounds in Algorithm 3

DERIVE-FIXES starts at the root of the P syntax tree with both $T_\perp$ and $T_\top$ being the reference formula, and ends at either a repair site in S where we use $P_\perp$ and $P_\top$ to derive a fix (lines 1–3), or a sub-tree whose pre-computed bounds are equivalent where P contains no repair site in S (lines 4–5). For a $\wedge$ node, the algorithm derives the bounds illustrated in Figure 2a (lines 11–15), while for a $\vee$ node it derives the bounds illustrated in Figure 2b (lines 16–20). At a $\neg$ node, the algorithm inverts the target formula bounds and move onto the child node (lines 21–22). In each of these cases, the algorithm recurses over the child nodes of the current node. We explain the subroutine MINIMIZE-FIX in the full version.

We now show that the target bounds computed by DERIVE-FIXES for S always lead us to some viable repair (S, F).

LEMMA 4.2 (CORRECTNESS OF REPAIR COMPUTED BY ALGORITHM 3). *Suppose $P^\star \in$ CREATE-BOUNDS(P, S). DERIVE-FIX($S, P, P^\star, P^\star$) returns F such that applying (S, F) to P yields a formula equivalent to $P^\star$.*

The heuristic bounds allow us to independently choose a fix for each repair site without considering how different choices of fixes might affect each other in the end. Note that when traversing the syntax tree in a top-down manner, we always further restrict the target lower bound by adding conjuncts and relaxing the upper bound by adding disjuncts (line 12, 13, 17, 18 in Algorithm 3) in order to maximize flexibility for deriving a fix in the end. When arriving at a repair site, we want to find the minimal formula (in terms of minterms) within the target formula bounds. We show that this problem is essentially the same as Boolean function minimization with don't-cares by introducing MINIMIZE-FIX.

EXAMPLE 10. *To demonstrate DERIVE-FIXES with MINIMIZE-FIX, consider the case of $n = 2$ in Example 5. We show how a set of fixes can be composed for the set of repair sites S. For convenience, we list the pre-computed lower-bounds/upper-bounds for each node from Example 8:*

N_1: $[\perp, (d \neq e \vee d > f) \vee a = c]$
N_2: $[\perp \wedge (d \neq e \vee d > f), \top \wedge (d \neq e \vee d > f)]$
N_3: $[a = c \wedge \perp, a = c \wedge \top]$
N_4: $[\perp, \top]$
N_5: $[d \neq e \vee d > f, d \neq e \vee d > f]$
N_6: $[a = c, a = c]$
N_7: $[\perp, \top]$

And the target formula bounds for each node are as followed:

$T_{N_2} \in [P_1 \wedge \top \wedge (d \neq e \vee d > f), P_1 \wedge \top \wedge (d \neq e \vee d > f)]$
$T_{N_4} \in [P_1 \wedge (d \neq e \vee d > f), (P_1 \wedge (d \neq e \vee d > f)) \vee \neg(d \neq e \vee d > f)]$
$T_{N_3} \in [P_1 \wedge a = c, P_1 \wedge a = c]$
$T_{N_7} \in [P_1 \wedge a = c, (P_1 \wedge a = c) \vee \neg a = c]$

As we invoke DERIVE-FIXES($\{N_4, N_7\}, P_2, P_1, P_1$), *the target formula computed for repair sites* N_4, N_7 *are as follows:*

$$T_{N_4} \equiv a = b \vee (a = c \wedge d > 10) \vee (a = c \wedge d < 7)$$
$$T_{N_7} \equiv d > 10 \vee d < 7 \vee (a = b \wedge d \neq e) \vee (a = b \vee d > f)$$

If we replace N_4 *with* T_{N_4} *and* N_7 *with* T_{N_7}, *the entire* P_2 *would be rewritten as:*

$$((a = b \vee (a = c \wedge d > 10) \vee (a = c \wedge d < 7)) \wedge (d \neq e \vee d > f)) \vee$$
$$(a = c \wedge (d > 10 \vee d < 7 \vee (a = b \wedge d \neq e) \vee (a = b \vee d > f))) \equiv$$
$$(a = b \wedge d \neq e) \vee (a = b \wedge d > f) \vee (a = c \wedge d > 10) \vee (a = c \wedge d < 7)$$
$$\equiv P_1$$

Thus, $F \equiv \{T_{N_4}, T_{N_7}\}$ *is a set of fixes for repair sites* $S \equiv \{N_4, N_7\}$.

Fixing parenthesization issues. We further have a procedure for handling inconsistencies in parenthesization. We perform a sanity check for logical precedence, as sometimes logical precedence causes drastic changes in the semantic of formula. We perform the check by inserting one pair of parentheses into the formula in all possible valid places and test if such addition of the parentheses make two queries equivalent. The purpose of such sanity check is partially to rule out that the wrong query has precedence issue since we aim for finding small fixes and a drastic change in logical precedence can be very influential over the formula structure. If two queries are still inequivalent after the sanity check, we will move forward with finding repair sites and fixes.

4.3 Dynamically Decreasing the Fix Sizes

While our algorithms from the previous section derive a set of valid fixes as shown in Example 10, those fixes are not optimal compared as we have seen for Example 5. In particular, such sub-optimality comes in when we independently derive the target formula bounds (Algorithm 3). Using the same notations as in the previous section, consider the target formula bounds for the direct subtrees of an "OR" node and their Venn Diagram in Figure 2b:

$$T_l \in [(T \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp},\ T \wedge P_{l\top}]$$
$$T_r \in [(T \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp},\ T \wedge P_{r\top}]$$

The region representing $(T \wedge P_{l\top} \wedge \neg P_{r\perp})$ and the region representing $(T \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp}$ overlap, though they match exactly the region of T, and this also applies for $T \wedge P_{l\top}$ and $T \wedge P_{r\top}$. Such fact implies that any combination of the target formulae from the two target formula bounds have semantic overlaps, which cause sub-optimality in the subsequent derivation of fixes.

EXAMPLE 11. *To illustrate this, reconsider Example 10. The predicates* $d > 10$ *and* $d < 7$ *appear in both* T_{N_4} *and* T_{N_7} *even if it is unnecessary for them to be in* T_{N_4}. *The same situation also occurs in the case of an "AND" node.*

In order to reduce such sub-optimality, we propose to use the following procedure **instead of Algorithm 3**. The optimized version starts with $T_\perp$ and $T_\top$ being the reference formula and update them in the same manner as it traverses down the syntax tree, but it stops at the lowest common ancestor of all repair sites. At this point, the algorithm calls a sub-process that does the following: **(1)** Replaces all repair sites in P with a unique identifier and constructs a truth table (we name it consistency table) with all semantically unique atomic predicates and identifiers. **(2)** Iterate over each row

$r1$	$r2$	$a=1$	$b=2$	$c=3$	$T_\perp$	$T_\top$	T	P
0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	1	1	1	1	1	0
0	0	1	0	0	1	1	1	0
0	0	1	0	1	1	1	1	0
0	0	1	1	0	1	1	1	0
0	0	1	1	1	1	1	1	0
0	1	0	0	0	0	0	0	1
0	1	0	0	1	0	0	0	1
0	1	0	1	0	0	0	0	1
0	1	0	1	1	1	1	1	1
0	1	1	0	0	1	1	1	1
0	1	1	0	1	1	1	1	1
0	1	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1	1

Figure 3: Part of the consistency table for Example 12

$a=1$	$b=2$	$c=3$	$r1, r2$	$r1$	$r2$
0	0	0	00,10	0	0
0	0	1	00,10	0	0
0	1	0	00	0	0
0	1	1	01,10,11	0	1
1	0	0	01,11	0	1
1	0	1	01,11	0	1
1	1	0	01,10,11	0	1
1	1	1	01,10,11	0	1

Figure 4: Constraint table for Example 12

in the consistency table and evaluate the truth value for $T_\perp$, $T_\top$, T (based on $T_\perp$, $T_\top$) and P.

Next, the algorithm proceeds to: **(1)** Constructs a truth table (we call it constraint table) with the same predicates in the consistency table. **(2)** Go through each row r in the constraint table and: **(a)** List all possible assignments of truth values for all identifiers. **(b)** For each truth value assignment of all identifier, concatenate it with the truth value assignment of the predicates and find the row r' that has the same assignment in the consistency table. **(c)** If the truth values of T and P in r' are not consistent (i.e., T and P have different truth value and T is not don't-care), eliminate this assignment of identifier truth value from r.

After obtaining all possible truth value assignments for each identifier, the algorithm takes a greedy approach to assign truth values which does the following: **(1)** Minimize a formula for each identifier in the order from left to right. **(2)** For each identifier in each row, choose a truth value that maximize the flexibility (i.e. when the truth value could be either 0 or 1, assign a don't-care). If the current identifier depends on the previous one, only consider the truth values that do not have any conflict (e.g. in the last row of the constraint table, $r1$ has been assigned 0, and thus $r2$ can only be 1). **(3)** After finishing assigning truth value for an identifier, run Quine-McCluskey method to derive a minimal formula for the identifier. This is the fix will be applied to the corresponding repair site. **(4)** Given the minimal formula, update the truth value of the identifier so that there is no don't-care left.

EXAMPLE 12. *Consider the following formulae:* $P_1 \equiv a = 1 \vee (b = 2 \wedge c = 3)$, $P_2 \equiv c = 3 \vee (b = 2 \wedge a = 1)$. *Let the repair sites in* P_2 *be* $c = 3$ *and* $a = 1$ *with identifier* $r1, r2$, *and their lowest common ancestor be the top-level* $\vee$. *We can obtain* $T_\perp \equiv T_\top \equiv P_1$. *Therefore, the consistency table can be constructed as partially shown in Figure 3. Part of the constraint table is shown in Figure 4. The* ($r1$,$r2$) *column indicates the possible truth values assignment for identifier* $r1$ *and* $r2$ *simultaneously. In the highlighted row, only three assignments are possible. When* $a = 1$, $b = 2$ *and* $c = 3$ *are all true,* $r1$ *and* $r2$ *cannot be both false as their corresponding row in the consistency table contains inconsistency between* T *and* P *as highlighted in red.* $r1$ *is minimized*

first to False and r2 is minimized later to $(b = 2 \wedge c = 3) \vee a = 1$, *yielding* $P_2 \equiv \bot \vee (b = 2 \wedge ((b = 2 \wedge c = 3) \vee a = 1)) \equiv P_1$. *While the ideal fixes are* $r1 \equiv a = 1$ *and* $r2 \equiv c = 3$, *we can actually achieve the ideal fixes by minimize r2 before r1. For our running example (Example 5), the fixes obtained by this approach are* $T_{N_4} \equiv a = c$ *and* $T_{N_7} \equiv d > 10 \vee d < 7$, *which are the optimal ones.*

While there is no guarantee the optimized algorithm will always find the ideal fixes for a set of repair sites at first due to the order it derives fixes (shown in Example 12), it can always permute such order to find the fixes that are smallest if the number of repair sites is small (as the number of permutations is exponential). We denote the version of QR-HINT that uses the optimized version of Algorithm 3 as QR-HINT-optimized.

4.4 HAVING Clause

Similar to WHERE, HAVING clauses can be treated as SQL logical formulas, possibly with extra function symbols with aggregates and DISTINCT. While the purpose of HAVING clause is to apply extra constraints on the groups, it is also possible that HAVING clause contain predicates that can appear in the WHERE clause.

EXAMPLE 13. *Consider the following two queries:*

```
Q1:
SELECT t2.author, t1.year
FROM inproceedings t1, authorship t2, inproceedings t3,
     authorship t4
WHERE t1.pubkey = t2.pubkey AND t3.pubkey = t4.pubkey
  AND t2.author = t4.author AND t1.area = t2.area
  AND t1.area = 'Database' AND t1.year <= t3.year
GROUP BY t2.author, t1.year
HAVING COUNT(DISTINCT t3.year) = 1;
Q2:
SELECT t4.author, t1.year
FROM inproceedings t1, authorship t2, inproceedings t3,
     authorship t4
WHERE t1.pubkey = t2.pubkey AND t3.pubkey = t4.pubkey
  AND t2.author = t4.author AND t1.year <= t3.year
GROUP BY t2.author, t4.author, t1.year, t1.area
HAVING COUNT(t3.year) = 1 AND t1.area = t2.area AND
     t1.area = 'Database';
```

Although the GROUP BY and HAVING clauses of Q_1, Q_2 *look very different, it can be noted that the only error in* Q_2 *is the* `COUNT(t3.year) = 1` *in the HAVING clause (it misses* `DISTINCT`*). The predicates* `t1.area = t2.area` *and* `t1.area = 'Database'` *in the HAVING clause of* Q_2 *appeared in the WHERE clause of* Q_1, *so simply testing the equivalence of WHERE clause would introduce false negatives due to missing two predicates in the WHERE clause. Therefore, let* P_1 *and* P_2 *denote the SQL logical formulas for WHERE clauses and* H_1 *and* H_2 *denote the formulas for HAVING clause in* Q_1 *and* Q_2, *we need to test the equivalence of the conjunction of WHERE and HAVING clauses (i.e.* $P_1 \wedge H_1 \equiv P_2 \wedge H_2$*). Given the two conjunctions of SQL logical formulas, we can invoke our previously defined procedure to find repair sites and fixes.*

5 HANDLING GROUP-BY CLAUSE

As described in Section 3.5, we find fixes for the FROM and WHERE clauses of the user query before the GROUP-BY clause. Therefore, we check the GROUP-BY equivalence with the assumption that both the reference and user Q_1, Q_2 have equivalent FROM and WHERE clauses. We first show how the problem can be reduced to the problem of checking equivalence between two logical formulas:

(1) Make a copy of all tables in Q_1 by giving new aliases. Suppose the original predicate in the WHERE clause of Q_1 was P_1, compute the copy of the predicate P_1' using the new aliases.
(2) Let $G_1 = \{g_1^1, ..., g_n^1\}$ be the set of GROUP-BY expressions in Q_1, and $G_1' = \{g_1^{1'}, ..., g_n^{1'}\}$ be their respective copies using the table aliases created above. Create a new logical formula
$$P_1'' = (P_1 \wedge P_1' \wedge (g_1^1 = g_1^{1'} \wedge ... \wedge g_n^1 = g_n^{1'})) \quad (2)$$
(3) Follow steps 1-2 also for query Q_2 with group by expressions $G_2 = \{g_1^2, ..., g_m^2\}$ to create P_2' and $G_2' = \{g_1^{2'}, ..., g_m^{2'}\}$, and
$$P_2'' = (P_2 \wedge P_2' \wedge (g_1^2 = g_1^{2'} \wedge ... \wedge g_n^2 = g_n^{2'})) \quad (3)$$

We assume that Q_2 has the same old and new table aliases as in Q_1.

EXAMPLE 14. *Consider the queries in Example 6. Here* P_1''*:*

```
(t1.pubkey = t2.pubkey AND t3.pubkey = t1.pubkey
       AND t3.author <> t2.author AND t1.year < 2018
       AND t1.area = 'Database') -- original WHERE
AND (t4.pubkey = t5.pubkey AND t6.pubkey = t4.pubkey
       AND t6.author <> t5.author AND t4.year < 2018
       AND t4.area = 'Database') -- WHERE predicate with
            new aliases t4 (for t1), t5 (t2), and t6 (t3)
AND (t2.author = t5.author AND t1.year = t4.year); --
     equality conditions for two group by attributes
```

Similarly, P_2''*:*

```
(t1.pubkey = t2.pubkey AND t3.pubkey = t1.pubkey
       AND t2.author <> t3.author AND t1.year < 2018
       AND t1.area = 'Database' AND t1.year < 2018)
AND (t4.pubkey = t5.pubkey AND t6.pubkey = t4.pubkey
       AND t5.author <> t6.author AND t4.year < 2018
       AND t4.area = 'Database' AND t4.year < 2018)
AND (t2.author = t5.author AND t1.area = t4.area
       AND t1.year - 2018 = t4.year - 2018
       AND t3.author = t6.author);
```

The following proposition shows that for any two tuples t_1, t_2 from the cross product of tables appearing in the FROM clause that satisfy the WHERE clause (for both Q_1, Q_2), if t_1, t_2 belong to the same groups in Q_1, Q_2, they must satisfy P_1'', P_2'' when t_1 is assigned to old table aliases and t_2 to new table aliases, and vice versa.

PROPOSITION 2. *The GROUP-BY clauses of* Q_1 *and* Q_2 *are equivalent if and only if* $P_1'' \equiv P_2''$.

EXAMPLE 15. *In Example 14,* P_1'' *and* P_2'' *are not equivalent, only due to the condition* `t3.author = t6.author`, *which is present in* P_2'' *but not in* P_1'', *hence the GROUP BY clauses in the queries in Example 6 are not equivalent either as we explained in that example.*

On the other hand, if Q_2 *did not have the* `t3.author` *in its GROUP BY clause, then the GROUP BY clauses will be equivalent despite*

seemingly different GROUP BY expressions `t1.year - 2018` *and* `t1.area` *in the GROUP BY clause of* Q_2*, since (modified)* P_1'' *and* P_2'' *in Example 14 will still be equivalent.*

Based on Proposition 2, we now give definitions of the repair sites and fixes for the GROUP-BY clause using the notations from the construction process. Note that if $P_1'' \equiv P_2''$, P_1'' must imply P_2'', in particular, $P_1'' \Rightarrow (P_2 \wedge P_2' \wedge (g_i^2 = g_i^{2'}))$, for each $1 \leq i \leq m$.

Definition 7 (Repair Sites and Fixes for GROUP-BY Clause). *The* **repair sites of the GROUP BY clause** *of user query* Q_2 *are a set of GROUP-BY expressions* $S \equiv \{s_1, ..., s_k\} \subseteq G_2$ *where for each expression* $s_i \in S$ *and its counterpart expression* $s_i' \in G_2'$, $P_1'' \not\Rightarrow$ *(i.e., does not imply)* $P_2 \wedge P_2' \wedge (s_i = s_i')$*. The* **fixes** *consist of* S *(GROUP BY expressions to remove from* Q_2*) and a set of GROUP-BY expressions* $F = \{f_1, ..., f_s\} \subseteq G_1$ *to add to the GROUP-BY clause of* Q_2*, such that it becomes equivalent to the GROUP-BY clause of* Q_1*.*

The algorithm for finding repair sites for GROUP BY clause of Q_2 is divided into two stages. The first stage finds all repair sites and the second stage determine the fixes. Since the GROUP-BY clauses are equivalent if and only if $P_1'' \equiv P_2''$, we verify if a GROUP-BY expression in G_2, say g_i^2, is correct by checking whether $P_1'' \Rightarrow P_2 \wedge P_2' \wedge (g_i^2 = g_i^{2'})$ holds (if not, it is a repair site). After removing all the repair sites from G_2, we add all group by expressions from G_1 to G_2, and remove the newly added expressions in an arbitrary order until removing any expression causes $P_1'' \equiv P_2''$ not to hold any more. The remaining expressions from G_1 forms a set of fixes.

6 HANDLING SELECT CLAUSE

Now we describe repairing SELECT clauses and aggregates.

6.1 Algorithm for SELECT Repair

Since repairing SELECT clause is the last step of Qr-Hint, we assume that the other clauses have been made equivalent by the proposed fixes in the earlier steps, i.e., FROM, WHERE, GROUP BY, and HAVING clauses are equivalent in both Q_1, Q_2. Let S_1, S_2 denote the ordered set of expressions in the SELECT clauses of Q_1, Q_2 respectively, and we assume that the output columns should be equivalent and also in the same order (Section 3.4). The algorithm iterates through each expression $s^2 \in S_2$ and checks whether the expression is equivalent to the one with the same index, say $s^1 \in S_1$ given the logical formula P for the WHERE clause of Q_1 and Q_2. For any tuple in the output, this formula must be true. Hence, we check for equivalence assuming P is true, i.e., whether $P \Rightarrow (s^1 \equiv s^2)$. After iterating over all expressions in S_2, if there are more expressions left in S_1, more expressions are needed in S_2 so we consider them as fixes. If more expressions are left in S_2, we consider them as repair sites which are to be removed.

Example 16. *Consider the following queries:*

```
Q1:
SELECT t2.author, t2.pubkey, t1.year
FROM articles t1, authorship t2
WHERE t1.pubkey = t2.pubkey;
Q2:
SELECT t2.author, t1.pubkey, t1.title, t1.year
FROM articles t1, authorship t2
WHERE t1.pubkey = t2.pubkey;
```

Here S_1 = (`t2.author, t2.pubkey, t1.year`) *and* S_2 = (`t2.author, t1.pubkey, t1.title, t1.year`)*. Both* S_1, S_2 *have* `t2.author` *in position 1, which does not need a change. For the second position, the WHERE predicate* (`t1.pubkey = t2.pubkey`) *makes the two output columns equivalent, hence it does not need a change. Hence the repair sites for the SELECT clause of the user query* Q_2 *is* {`t1.title, t1.year`} *and the corresponding fixes are* {`t1.year`}*, since replacing* `t1.title` *with* `t1.year` *and removing* `t1.year` *at the end makes the SELECT clauses of the two queries equivalent.*

6.2 Aggregates/DISTINCT and Limitations

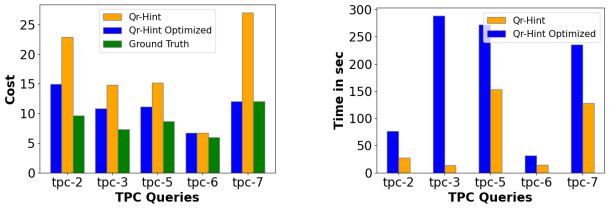
The DISTINCT keyword and aggregates (MIN, MAX, COUNT, AVG, SUM) can appear in the SELECT and/or HAVING clauses. Qr-Hint treats them as unary function symbols in the expressions and logical formulas. Qr-Hint uses Z3 SMT Solver [16] to check implications between two formulas. Therefore, the operations and functions we support are limited by the Z3 solver. While most of the comparison and arithmetic operators, string concatenations, and all logical connectives are supported by the Z3 solver, it does not support the *LIKE* operator while it does support checking the matching of suffix, prefix and string containment. In addition, Z3 Solver does not support aggregation functions (SUM, AVG, COUNT, MAX, MIN), hence Qr-Hint can not recognize $AVG(x) \equiv \frac{SUM(x)}{COUNT(x)}$.

7 EXPERIMENTS

In this section, we describe experimental results to evaluate the accuracy and performance of Qr-Hint. In Section 8 we describe a user study for further qualitative evaluation.

Implementation details. We implemented Qr-Hint and Qr-Hint-optimized in Python 3.8 using Apache Calcite [5] to parse SQL queries and Z3 SMT Solver [16] to test constraint satisfactions. We run the experiments locally on a 64-bit Ubuntu 20.04 LTS server with 3.20GHz Intel Core i7-8700 CPU and 32GB 2666MHz DDR4.

Schema and queries. We tested Qr-Hint with the TPC-H [6] schema and queries, to have insights on how Qr-Hint might perform during applications on realistic queries (the user study is performed on the DBLP dataset). TPC-H contains 8 tables (part, partsupp, lineitem, orders, supplier, customer, nation, region) and each table has an average of 7 attributes. Out of the 22 queries in the dataset, we picked tpc-2, tpc-3, tpc-5, tpc-6 and tpc-7 as the testing queries (this is a representative set of the WHERE clause of all queries that we can handle). For each query, we make the following necessary changes: 1) Remove subqueries, 2) Remove predicates that have keyword INTERVAL, 3) Change "x BETWEEN a and b" with two separate predicates using ">" and "<", 4) For tpc-7 whose WHERE clause is in a giant subquery in FROM clause, we remove the outer query block and use the subquery as the test query. After applying the above changes, each selected query contains 6, 5, 7, 4, 8 syntactically unique predicates. We then manually changed the queries to add two errors in their WHERE clause (we recommend that Qr-Hint will be used with up to 2 errors in WHERE clause) and, therefore, we also have the ground truth edits for each query. We will employ this knowledge in our accuracy experiments. In



(a) Cost (b) Runtime

Figure 5: Qr-Hint vs. Qr-Hint-optimized

summary, we changed two conditions to be wrong in tpc-2, tpc-3, tpc-5 and tpc-7; we changed one condition and removed one condition in tpc-6. In addition, tpc-2, tpc-3, tpc-5 and tpc-6 are all conjunctive queries while tpc-7 contains a mix of AND/OR logical operator.

Examined approaches. We have performed our automatic experiments with Qr-Hint and Qr-Hint-optimized (Section 4.3). We measure both the accuracy and performance. In our accuracy experiments we have compared the fixes found by Qr-Hint and Qr-Hint-optimized to the ground truth.

Measures used in the experiments. We have performed both accuracy and performance experiments. Performance is measured by runtime and accuracy is measured by the succinctness cost function defined in equation (1) in Section 4.1.

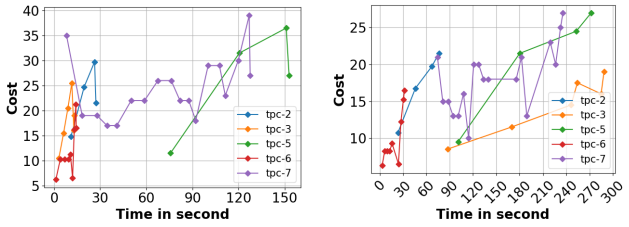
7.1 Results

We describe the results of our accuracy and runtime experiments.

Accuracy. Figure 5a shows the results of the accuracy experiments in terms of cost for TPC-H queries. We see that the ground truth edits always have the smallest cost. Qr-Hint-optimized is able to detect fixes close to the optimal ones, whereas Qr-Hint often finds fixes that require more significant edits. As shown in the figure, Qr-Hint has a cost that is close to that of Qr-Hint-optimized when the query size is small (e.g. tpc-6 only has 4 predicates). However, when there are more complicated logical structures (e.g., nested AND/OR), Qr-Hint produces higher cost (27 for tpc-7) while Qr-Hint-optimized maintains low cost (12 for tpc-7). In particular, for all queries, the cost of the ground truth was 7.83, 7.66, 8.33, 6.5 and 12 respectively, compared to the cost of Qr-Hint-optimized that was 16.75, 12.5, 13.5, 8.25 and 12 respectively. This suggests that Qr-Hint-optimized has the ability to derive close to optimal fixes.

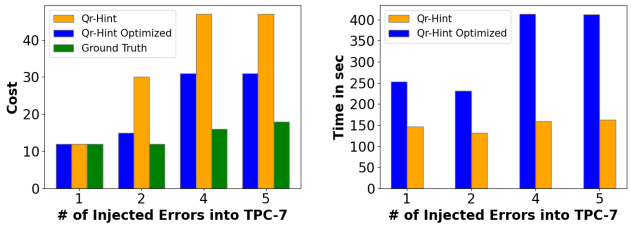
Computation time. Figure 5b depicts the runtimes of TPC-H queries for Qr-Hint and Qr-Hint-optimized. In general, the runtime increases with the number of unique predicates in the query. We observe the optimized version incurs higher runtimes for all queries, which can be explained by the need to generate both the consistency and constraint tables (Section 4.3). For example, for queries tpc-6 and tpc-7 (with 4 and 8 predicates) the runtimes of Qr-Hint were 13 and 130 seconds, respectively, while the runtimes of Qr-Hint-optimized were 31 and 224 seconds, respectively.

Cost over time. Figure 6 demonstrates how the costs of both Qr-Hint and Qr-Hint-optimized change over time when testing on the TPC-H queries. We observe that the minimum cost usually occurs early in the iteration when the query structure is simple (e.g. conjunctive queries). However, when a query involves multiple level of nesting logical operator (e.g. tpc-7), it takes more iterations



(a) Qr-Hint (b) Qr-Hint-optimized

Figure 6: Cost of each iteration of repair sites and fixes



(a) Cost (b) Runtime

Figure 7: Cost and runtime with number of injected errors

for our algorithm to determine the repair sites and fixes with the minimum cost since the cost does not consistently grow over time.

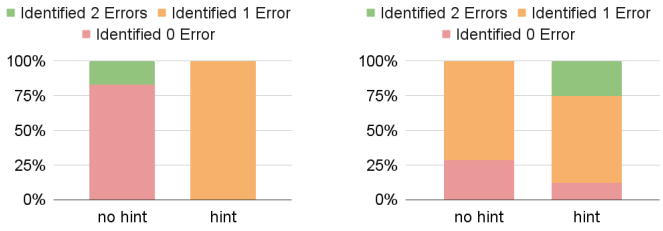
Relationship between number of errors and cost/runtime. Figure 7 shows the change in cost and runtime against the number of errors in a query. We chose tpc-7 as it has the most number of predicates as well as a complex structure. In general, the cost and runtime both increase with the number of errors in the query because we prioritize small repair sites and get to large repair sites toward the end. As always, Qr-Hint-optimized outperforms Qr-Hint in cost but runs longer than Qr-Hint in general.

8 USER STUDY

We conducted a user study to evaluate the Qr-Hint framework: (1) whether students can recognize and understand what is wrong with a query with the proposed hints/fixes, and (2) how the hints generated by Qr-Hint-optimized are comparable to the ones provided by '*expert users*' (teaching assistants in our study).

Preparation. We designed four SQL questions Q_1, Q_2, Q_3, Q_4 using the DBLP database schema (details in full version [4]). For each question, we prepared a wrong solution that contains one or more mistakes: two WHERE errors in Q_1, one each GROUP BY and SELECT errors in Q_2, a parenthesization error in WHERE of Q_3, and one each WHERE and HAVING errors in Q_4. In addition, we ran Qr-Hint on all wrong queries to obtain repair sites and fixes. We removed fixes and only showed repair sites to the participants (not to the experts). Finally, in order to prevent participants from recognizing the generated hints whether from experts or from Qr-Hint by observing the hint format, we paraphrased all hints as"In [SQL Clause], [Hints]". Then we conducted the user study in two steps:

Step 1: generation of hints by human experts. First, a smaller study was performed with four graduate teaching assistants (TAs) to generate hints given by expert users. For each wrong query, each TA was asked to pinpoint and list all mistakes in a query as if they are giving hints to help students debug wrong queries. In order to simulate the environment of office hours, we asked TAs to finish all four questions in one sitting. At the end of the questionnaire, we



(a) Q1: No Hints vs. QR-HINT (b) Q2: No Hints vs. QR-HINT

Figure 8: User performance identifying errors with and without QR-HINT(one student found 2 errors in Q_1 after 20 mins.)

collected the hints provided by the TAs. These TA-generated hints are later used for comparison with the hints given by QR-HINT.

Step 2: study with students as participants. Then, we conducted a user study as described below to evaluate the hints by QR-HINT.

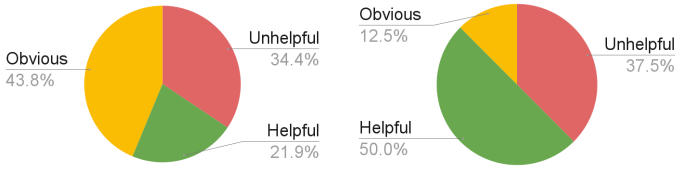
Participants. We recruited 15 students who have taken/are taking a graduate or undergraduate database course[3]. All participants have finished training for writing and debugging SQL queries.

Tasks. Using the four queries, each participant saw and completed three questions. Students were required to complete questions on Q1 and Q2, and they completed one of Q3 and Q4 at random. For each question, students were given the database schema, problem statement in English, and the wrong SQL query, and were asked to explain what is wrong with the query. For creating *treatment* and *control* groups, students received hints from QR-HINT for either Q1 or Q2 (not both) at random, and for the other one they were asked to detect errors without any hints provided; the order of the two questions with and without hints was also chosen at random. For the last question, participants received Q3 or Q4 at random, and we showed the union of hints (mixed together) generated by the TAs as well as by QR-HINT, and asked participants to categorize each hint as one of the following: "*Unhelpful or incorrect*" (for hints that did not assist them in detecting errors), "*Helpful but require thinking*" (for hints that were helpful but did not entirely reveal the errors), and "*Obvious and giving away the answer*" (for hints that reveal the errors immediately). We used these ranking to evaluate the quality of the hints given by QR-HINT and compare them to the TAs' hints. All participants were asked to finish all questions in one sitting. We recorded the time a participant spent on each question[4]. In our study, for the question on Q1, 8 students answered it with no hints and 7 with hints from QR-HINT. For the question on Q2, 7 students answered it with no hints and 8 with hints from QR-HINT. For the third question, 7 students received Q3 and 8 received Q4.

Result and Analysis. Our results for Q1 and Q2 show that participants were better at identifying at least one error in the query given the hints provided by QR-HINT compared to no hints. As shown in Figure 8a and Figure 8b, 100% and 87.3% of the participants were able to identify at least one of the two errors in the wrong query in Q1 and Q2 respectively after receiving hints from QR-HINT, as

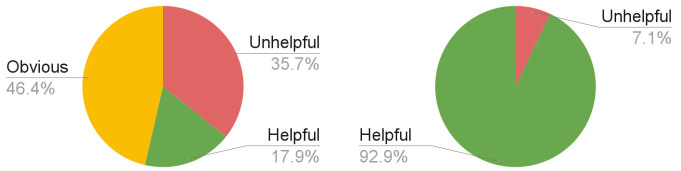
[3]Except an incentive of receiving a small gift card, the participation of students were voluntary, and we were able to get 15 complete responses. We plan to do a larger-scale user study in the future.

[4]We note that Q_1 without hints took 704s on average, with hints took 460s; Q_2 without hints took 756s on average, and with hints took 658s. However, the students were able to answer our survey asynchronously without supervision, so the time recorded may not accurately reflect the time they actually took.



(a) Q3: Vote Cast for TA Hints (b) Q3: Vote Cast for QR-HINT

Figure 9: Hint categorization from participants for Q3



(a) Q4: Vote Cast for TA Hints (b) Q4: Vote Cast for QR-HINT

Figure 10: Hint categorization from participants for Q4

opposed to 14.3% and 71.4% who were able to do so without a hint. While there is a single participant who correctly identified both errors without any hint for Q1, this participant spent more than 20 minutes to do so, while most participants spent no more than 10 minutes on the same question without hints.

Q3 and Q4 are used to evaluate whether QR-HINT provided hints that are comparable to the ones given by teaching assistants in terms of their quality. For Q3, there are four TA hints and one hint from QR-HINT; and there are four TA hints and two hints generated by QR-HINT for Q4. For all responses, we sum up the number of times participant vote for each of the three categories of hint ranks: "Obvious", "Unhepful", and "Helpful". The results are shown in Figures 9a, 9b, 10a and 10b. In summary, the quality of TAs' hints vary greatly as perceived by participants. On the other hand, hints generated by QR-HINT are consistently perceived by participants "helpful but require thinking", which might be best suited for classroom settings where we may not want to directly give out the solutions to students. Also automated generation of hints tailored to queries produced by students is useful for larger classes when personalized help from TAs may be difficult to scale.

9 CONCLUSION AND FUTURE WORK

In this paper, we presented QR-HINT, a framework for automatically generating hints and suggestions for fixes for a wrong SQL query with respect to a reference query. We developed techniques to produce fixes for all of SELECT, FROM, WHERE/HAVING, and GROUP BY clauses and gave theoretical guarantees. There are multiple intriguing directions of future work, including the support of more complex SQL queries, such as ones that include subqueries, negations, outerjoins, and can handle nulls and constraints in the data, and finally can go beyond standard SQL and can support recursions and Datalog. There are many steps where the framework evaluates all possible options (e.g., repair sites), hence improving the scalability of the system is also a future work. We are implementing a graphical user interface for QR-HINT so that it will be used as a learning tool for students in database courses to help students and TAs. Conducting a larger-scale user study to understand the effectiveness of this tool in helping students learn to debug SQL queries and write correct queries is also an important future work.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc.

[2] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. 31, 4, Article 74 (jul 2022), 31 pages.

[3] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. 1979. Equivalences among relational expressions. *SIAM J. Comput.* 8, 2 (1979), 218–246.

[4] Anonymized git and the full version of this submission. [n.d.]. https://anonymous.4open.science/r/sigmod2023-5358/. ([n. d.]).

[5] 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 *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[6] TPC Benchmark. [n.d.]. http://www.tpc.org/tpch.

[7] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 60–70.

[8] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-based why-not provenance with nedexplain. In *Extending database technology (EDBT)*.

[9] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC '77)*. Association for Computing Machinery, 77–90.

[10] Bikash Chandra, Ananyo Banerjee, Udbhas Hazra, Mathew Joseph, and S Sudarshan. 2021. Edit Based Grading of SQL Queries. In *8th ACM IKDD CODS and 26th COMAD*. 56–64.

[11] Bikash Chandra, Bhupesh Chawda, Biplab Kar, KV Reddy, Shetal Shah, and S Sudarshan. 2015. Data generation for testing and grading SQL queries. *The VLDB Journal* 24, 6 (2015), 731–755.

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

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

[14] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524.

[15] DBLP data. [n.d.]. https://dblp.org/. ([n. d.]).

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[17] Carl Friedrich Gauss. 1966. *Disquisitiones arithmeticae*. Yale University Press.

[18] Amir Gilad, Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2022. Understanding Queries by Conditional Instances. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, 355–368.

[19] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, 465–480.

[20] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 930–937.

[21] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural attribution for semantic bug-localization in student programs. *Advances in Neural Information Processing Systems* 32 (2019).

[22] Tomasz Imieliński and Witold Lipski Jr. 1989. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*. Elsevier, 342–360.

[23] Yannis E Ioannidis and Raghu Ramakrishnan. 1995. Containment of conjunctive queries: Beyond relations as sets. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 288–324.

[24] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. 2006. The Containment Problem for Real Conjunctive Queries with Inequalities. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '06)*. Association for Computing Machinery, 80–89.

[25] Anthony Klug. 1988. On conjunctive queries containing inequalities. *Journal of the ACM (JACM)* 35, 1 (1988), 146–160.

[26] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2303–2318.

[27] Edward J McCluskey. 1956. Minimization of Boolean functions. *The Bell System Technical Journal* 35, 6 (1956), 1417–1444.

[28] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining wrong queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data*. 503–520.

[29] Kai Presler-Marshall, Sarah Heckman, and Kathryn Stolee. 2021. SQLRepair: identifying and repairing mistakes in student-authored SQL queries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 199–210.

[30] Yehoshua Sagiv and Mihalis Yannakakis. 1980. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)* 27, 4 (1980), 633–655.

[31] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, 15–26.

[32] Boris Trahtenbrot. 1950. The impossibility of an algorithm for the decision problem for finite domains. In *Doklady Academii Nauk SSSR*, Vol. 70. 569–572.

[33] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, 481–495.

[34] 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 *(SIGMOD '22)*. Association for Computing Machinery, 94–107.

[35] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.

Algorithm 4 Minimize-Fix

MINIMIZE-BOUNDS($P_\perp, P_\top$)

Input: $P_\perp$: lowerbound of an SQL logical formula
$P_\top$: upperboundof an SQL logical formula
Output: A formula that contains minimum number of minterms

1 $t_\perp, t_\top$ = Construct-Expanded-Truth-Tables($P_\perp$, $P_\top$)
2 final_truth_table = $t_\top$
3 **for** row i in $t_\perp, t_\top$:
4 **if** $t_\top[i][P_\top] = t_\perp[i][P_\perp] = c$ or
($t_\top[i][P_\top] = true$ and $t_\perp[i][P_\perp] = false$):
5 final_truth_table[i][P] = x
6 **else** :
7 final_truth_table[i][P] = $Combine(t_\perp[i][P\perp], t_\top[i][P_\top])$
8 minimal_formula = Quine-McCluskey-Method(final_truth_table)
9 **return** minimal_formula

A MISSING PSEUDO CODES

In this section, we present the missing Pseudo Codes for handling various clauses.

A.1 Minimizing the fix sizes

MINIMIZE-FIX is a subroutine of Algorithm 3 (line 2). The pseudo code appears in Algorithm 4. It takes a lower bound $P_\perp$ and an upper bound $P_\top$ as inputs and utilizes the Quine-McCluskey's method [27] to find a formula P in disjunctive normal form, having the minimum number of minterms among all formulas that fall into the bounds. It first scans through both $P_\perp$ and $P_\top$ and extracts all semantically unique atomic predicates (e.g. $a = b$ is semantically equivalent to $a+1 = b+1$, thus one of them would be dropped in the truth table), then based on which it constructs a truth table with columns for the formulas $P_\perp, P_\top$ and the final formula P (line 1) and fills out the truth values for $P_\perp$ and $P_\top$ separately following the below rules:

- If the conjunction of all atomic predicates in a row is not satisfiable (e.g. there exists a contradiction like $a = b$ and $a > b$ are both true), we mark both truth values for $P_\perp$ and $P_\top$ by "c" for contradiction.
- If the conjunction of all atomic predicates in a row is satisfiable, we evaluate the truth value for $P_\perp$ and $P_\top$ based solely on the atomic predicates contained in the formula.

Such truth table construction yields two truth tables for $P_\perp$ and $P_\top$, respectively. Due to $P_\perp \implies P_\top$, we can infer that for all rows in the truth tables where $P_\perp$ evaluated to true, the corresponding row in $P_\top$ must also evaluate to true; $P_\perp$ is evaluated to false, the corresponding row in $P_\top$ can be either true or false. Therefore, we construct each row in the truth table of P by the following rules:

(1) If the truth value of $P_\perp$ and $P_\top$ are both false, we mark the truth value of P as false (line 7).
(2) If the truth value of $P_\perp$ and $P_\top$ are both true, we mark the truth value of P as true (line 7).
(3) If the truth value of $P_\perp$ and $P_\top$ is a "c", we mark the truth value of P as a don't-care "x" (line 5).
(4) If the truth value of $P_\perp$ is false but true for $P_\top$, we mark the truth value of P as a don't-care "x" (line 5).

While items (1) and (2) are hard requirements needed to guarantee $P \in [P_\perp, P_\top]$, items (3) and (4) allow more flexibility when Quine-McCluskey's method minimizes P based on the truth table of P (line 8). We mark the truth value of P as a don't-care when the conjunction of atomic predicates in a row is not satisfiable because such situation can never happen (e.g. $a = b$ and $a > b$ can never be true at the same time). We also mark the truth value of P as a don't-care when $P_\perp$ is evaluated to false while $P_\top$ is evaluated to true because P can either be true or false due to $P \in [P_\perp, P_\top]$.

In the end, the formula P represented by its truth table is guaranteed to fall into $[P_\perp, P_\top]$. As a result, the problem becomes Boolean function minimization with don't-cares which can be handled by the well-known Quine-McCluskey heuristic [27].

EXAMPLE 17. *Consider the following bounds for some SQL logical formula P: $P_\perp \equiv (a \geq b \wedge f = e) \vee a = b$, and $P_\top \equiv a = b \vee e = f \vee a > b$. Thus, we can construct a truth table for each formula shown in Figure 11. Based on the truth tables, it is clear that contradiction occurs when*

- *$a = b$ and $a > b$ are both true.*
- *Either $a = b$ or $a > b$ is true but $a \geq b$ is false.*
- *$a \geq b$ is true but both $a = b$ and $a > b$ are false.*

Running the Quine-McCluskey method on the final truth table for P yields $P \equiv a \geq b$, and $P \in [P_\perp, P_\top]$.

$a \geq b$	$f = e$	$a = b$	$a > b$	$P_\perp$	$P_\top$	P
0	0	0	0	0	0	0
0	0	0	1	c	c	x
0	0	1	0	c	c	x
0	0	1	1	c	c	x
0	1	0	0	0	1	x
0	1	0	1	0	1	x
0	1	1	0	c	c	x
0	1	1	1	c	c	x
1	0	0	0	c	c	x
1	0	0	1	0	1	x
1	0	1	0	1	1	1
1	0	1	1	c	c	x
1	1	0	0	c	c	x
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	c	c	x

Figure 11: Compact Truth tables for $P_\perp$, $P_\top$ and P in Example 17.

B ALGORITHM FOR SECTION 4.3

The overall routine of QR-HINT-optimized is shown in Algorithm 5. In general, it operates the same as Algorithm 3 except for the base case where it reaches the lowest common ancestor of all repair sites and start deriving fixes by constructing the consistency table and constraint table (line 1-2). At the lowest common ancestor of all repair sites, Algorithm 5 first creates the consistency table (line 2) and use the consistency table to construct the constraint table (line 3) which is later used to greedily derive fixes (line 4).

Algorithm 6 takes the same inputs as Algorithm 5 and outputs a consistency table. It first assigns identifiers for all repair sites (line 1), and extract all syntactic unique predicates from P, $T_\perp$ and $T_\top$ (line 2). It then creates the truth table whose variables are the union of all identifiers and predicates, and the formulas we want to evaluate are $T_\perp$, $T_\top$, T and P (line 3). Here $T \in [T_\perp, T_\top]$. While evaluating the formulas at each row in the truth table, there are two possible scenarios:

- The truth assignment of the row is not satisfiable under conjunction (e.g. $a > b$ and $a = b$ cannot be true at the same time). In this case, we mark all truth value of the four formulas as don't-care because such scenario can never happen (line 6-7).

Algorithm 6 Construct-Consistency-Table

CONSTRUCT-CONSISTENCY-TABLE($s, P, T_\perp, T_\top$)

Input:
s: A set of repair sites
P: Current root node syntax tree of the incorrect SQL logical formula with pre-computed bounds w.r.t. s
$T_\perp$: Current target lowerbound
$T_\top$: Current target upperbound
Output: A consistency table

```
1  idn = Create-Identifiers(s)
2  all_preds = Extract-All-Predicates([P, T⊥, T⊤])
3  T_consistency = Create-Truth-Table(idn + all_preds, [T⊥, T⊤, T, P])
4
5  for each row r in T_consistency:
6      if r is not satisfiable:
7          r[T⊥], r[T⊤], r[T], r[P] = c, c, x, x
8      else
9          r[T⊥] = Evaluate-Row(r, T⊥)
10         r[T⊤] = Evaluate-Row(r, T⊤)
11         if r[T⊥] = r[T⊤]:
12             r[T] = r[T⊤]
13         else
14             r[T] = x
15         r[P] = Evaluate-Row(r, P)
16 return T_consistency
```

Algorithm 7 Construct-Constraint-Table

CONSTRUCT-CONSTRAINT-TABLE($T_{consistency}$)

Input:
$T_{consistency}$: Consistency table
Output: A constraint table

```
1  idn = Extract-identifiers(T_consistency)
2  all_preds = Extract-Predicates-From-Table(T_consistency)
3  T_constraint = Create-Truth-Table(all_preds, ["assignments"] + idn])
4  T_constraint[assignments] = Generate-All-Assignments(idn)
5  for each row r in T_constraint:
6      for each row r' in T_consistency:
7          if r and r' have the same truth value assignment for all_preds
               AND r'[T] != r'[P] AND r'[T] != x:
8              r[assignment].remove(r'[idn])
9  return T_constraint
```

Algorithm 8 Greedy-Minimize

GREEDY-MINIMIZE($T_{constraint}$)

Input:
idn: list of repair sites identifier
$T_{constraint}$: Constraint table
Output: A list of fixes

```
1  fixes = []
2  prev_id = None
3  for each id in idn:
4      for each row r in T_constraint:
5          r[id] = Get-Flexible-Assignment(r[assignments], r[prev_id])
6      P_min = Quine-MacCluskey(T_constraint[id])
7      for each row r in T_constraint:
8          r[id] = Evaluate-Row(r, P_min)
9      fixes.add(P_min)
10     prev_id = id
11 return fixes
```

Algorithm 5 Derive-Optimized-Fixes

DERIVE-OPTIMIZED-FIX($s, P, T_\perp, T_\top$)

Input:
s: A set of repair sites
P: Current root node syntax tree of the incorrect SQL logical formula with pre-computed bounds w.r.t. s
$T_\perp$: Current target lowerbound
$T_\top$: Current target upperbound
Output: A set of pair of (repair site, fix)

```
1  if P is the lowest common ancestor of s:
2      T_consistency = Construct-Consistency-Table(s, P, T⊥, T⊤)
3      T_constraint = Construct-Constraint-Table(T_consistency)
4      fixes = Greedy-Derive-Fixes(T_constraint)
5      result = []
6      for i ∈ [0, size(s)]:
7          result.add((s[i], fixes[i]))
8      return result
9  elseif P.lower ≡ P.upper:
10     return [ ]
11
12 F = [ ]
13 if P.op ∈ {∧}:
14     P_l, P_r = P.left, P.right
15     P_l⊥, P_l⊤ = P_l.lower, P_l.upper
16     P_r⊥, P_r⊤ = P_r.lower, P_r.upper
17     left_bounds = [T⊥ ∨ P_l⊥,  (T⊤ ∨ P_l⊥ ∨ ¬P_r⊤) ∧ P_l⊤]
18     right_bounds = [T⊥ ∨ P_r⊥,  (T⊤ ∨ P_r⊥ ∨ ¬P_l⊤) ∧ P_r⊤]
19     F += Derive-Fixes(s, P_l, left_bounds[0], left_bounds[1])
20     F += Derive-Fixes(s, P_r, right_bounds[0], right_bounds[1])
21 elseif P.op ∈ {∨}:
22     P_l, P_r = P.left, P.right
23     P_l⊥, P_l⊤ = P_l.lower, P_l.upper
24     P_r⊥, P_r⊤ = P_r.lower, P_r.upper
25     left_bounds = [(T⊥ ∧ P_l⊤ ∧ ¬P_r⊥) ∨ P_l⊥,  T⊤ ∧ P_l⊤]
26     right_bounds = [(T⊥ ∧ P_r⊤ ∧ ¬P_l⊥) ∨ P_r⊥,  T⊤ ∧ P_r⊤]
27     F += Derive-Fixes(s, P_l, left_bounds[0], left_bounds[1])
28     F += Derive-Fixes(s, P_r, right_bounds[0], right_bounds[1])
29 elseif P.op ∈ {¬}:
30     F += Derive-Fixes(s, P.child, ¬P⊤, ¬P⊥)
31
32 return F
```

- The truth assignment of the row is satisfiable under conjunction. Then we evaluate $T_\perp$ and $T_\top$ as they are, and derive the truth value of T based on both $T_\perp$ and $T_\top$ (line 9-14). Note that the scenario where $T_\perp$ is true but $T_\top$ is false can never happen as we have shown how we will always guarantee $T_\perp \implies T_\top$ previously. Finally, P is evaluated using the truth value of identifiers (line 15).

Algorithm 7 takes a consistency table as the input, and outputs a constraint table. It uses the same set of syntactically unique predicates as variables, but treats identifiers as the formulas for evaluation (line 3). Since each identifier represents a repair site, the general intuition is to construct fixes using all given predicates. However, there could be multiple possible truth assignments for a list of identifiers, so we insert all possible assignments into the table (line 4) and later eliminate the ones that are infeasible. In order to eliminate the infeasible assignments of identifiers, we go through

each row r in the constraint table, find its corresponding rows (i.e. the rows with the same truth assignment for predicates) in the consistency table and verify that T and P are indeed inconsistent (e.g. $T \neq P$ and T is not don't-care, line 7). If T and P are inconsistent, it means the truth assignment of identifiers and the truth assignment predicates for that particular row in the consistency table cannot be transferred to the constraint table. Thus we remove the assignments of identifiers from all possible assignments in r (line 8). At the end of Construct-Constraint-Table, we return a consistency table that contains valid truth assignments for identifiers in each row. However, the truth value of each identifier remains empty and will be computed by Greedy-Minimize.

Algorithm 8 takes a constraint table as the input, and returns a list of fixes corresponding to the repair sites in s in Algorithm 5. It optimizes the formula one at a time for each identifier (line 3). For each identifier, Algorithm 8 iterates through each row, checks what the possible truth values are and decides the truth value of the identifier in each row to be true, false or don't-care (line 5). It then calls Quine-McCluskey method to get a minimal formula P_{min} for the identifier (line 6). After P_{min} is obtained, the truth assignment of the current identifier in each row now has a concrete truth value (i.e. 0 or 1) based on P_{min}, so we iterates through each row in the constraint table again to change the truth value of the identifier to a concrete value so that more flexibility will be given to the optimization of next identifier (line 7-8).

Complexity of Algorithm 5. Algorithm 5 requires exponential complexity in the number of repair sites (denoted by s) and the number of syntactically unique predicates (denoted by n) as they control the size of the truth table generated both in Algorithm 6 and Algorithm 7. Since the number of truth table created is constant and the number of operations performed for each row in the truth tables is also constant, the overall complexity of Algorithm 5 is $O(2^{s+n})$.

$r1$	$r2$	$a=1$	$b=2$	$c=3$	$T_\perp$	$T_\top$	T	P
0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	1	1	1	1	1	0
0	0	1	0	0	1	1	1	0
0	0	1	0	1	1	1	1	0
0	0	1	1	0	1	1	1	0
0	0	1	1	1	1	1	1	0
0	1	0	0	0	0	0	0	1
0	1	0	0	1	0	0	0	1
0	1	0	1	0	0	0	0	1
0	1	0	1	1	1	1	1	1
0	1	1	0	0	1	1	1	1
0	1	1	0	1	1	1	1	1
0	1	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0	1
1	0	0	1	1	1	1	1	1
1	0	1	0	0	1	1	1	0
1	0	1	0	1	1	1	1	0
1	0	1	1	0	1	1	1	1
1	1	0	0	0	0	0	0	1
1	1	0	0	1	0	0	0	1
1	1	0	1	0	0	0	0	1
1	1	0	1	1	1	1	1	1
1	1	1	0	0	1	1	1	1
1	1	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	1

Figure 12: The full consistency table for Example 12

B.1 Algorithm for Section 5

Algorithm 9 Repair-Group-By

Repair-Group-By(Q_1, Q_2)

Input:
Q_1: Reference query
Q_2: Query to be fixed
Output: missing GROUP-BY expression, redundant GROUP-BY expression

1 P_1, P_1' = Create-Formulas(Q_1)
2 $P = P_1 \wedge P_1'$
3 G_1, G_1' = Create-GROUP-Expressions(Q_1)
4 G_2, G_2' = Create-GROUP-Expressions(Q_2)
5 $P_1'' = P \wedge G_1[1] = G_1'[1] \wedge G_1[2] = G_1'[2]...$
6 $P_2'' = P \wedge G_2[1] = G_2'[1] \wedge G_2[2] = G_2'[2]...$
7 missing, incorrect = [], []
8 **for** $i \in size(G_2)$:
9 **if** not $P_1'' \implies P \wedge G_2[i] = G_2'[i]$
10 incorrect.add($G_2[i]$)
11 **for** $i \in size(G_1)$:
12 **if** not $P_2'' \implies P \wedge G_1[i] = G_1'[i]$
13 missing.add($G_1[i]$)
14 **return** incorrect, missing

The pseudo code for the algorithm for finding repair sites and fixes for the GROUP BY clause is shown in Algorithm 9. It takes two queries as inputs, and output a set of repair sites and fixes. It first creates the formula P_1 representing the conjunction of WHERE and HAVING clauses in the reference query, and the counterpart P_1' which is the same as P_1 but with different aliases as demonstrated in Example 14 (line 1). It then creates P by taking the conjunction of P_1 and P_1' (line 2). It extracts and creates the GROUP BY expressions also shown in Example 14 (line 3-4), after which it creates P_1'' and P_2'' as shown in Example 14. Note that we do not need to create P_2 and P_2' separately because we start the process with the necessary assumption that Q_1 and Q_2 have equivalent WHERE and HAVING clauses. If the WHERE and HAVING clauses are not equivalent, the two constructed logical formulas will always be inequivalent because of $P_1 \not\equiv P_2$ in the first place. After all necessary components have been created, we first examine if each GROUP BY expression is correct by checking the implication between P_1'' and the conjunction of P and the predicate corresponds to the expression, and recognize the expression as incorrect if the implication does not hold (line 8-10). After identifying all incorrect expressions, we examine if each GROUP BY expression is missing by checking the implication between P_2'' and the conjunction of P and the predicate corresponds to the expression, recognize the expression as missing if the implication does not hold (line 11-13).

B.2 Algorithm for Section 6.1

The algorithms for finding the repair sites and fixes are shown in Algorithm 10. It takes the SQL logical formula P of the WHERE and HAVING clause of Q_1 (or Q_2, assuming they are equivalent), and two lists of SELECT expressions as inputs. Given P, it then tests if P implies the pair-wise equality between each expression with the same index in S_1 and S_2. Unlike the GROUP BY clause,

Algorithm 10 SELECT-Repair-Sites-and-Fixes

SELECT-Repair-Sites-and-Fixes(P_1, P_2, n)

Input:
S_1: Ordered set of SELECT expressions in Q_1
S_2: Ordered set of SELECT expressions in Q_2
P: SQL logical formula for Q_1 WHERE clause
Output: A set of repair sites and a set of fixes

```
1  for k ∈ [1, size(S2)]
2      if k ≤ size(S1) and not P ⟹ S1[k] = S2[k]
3          repair_sites.append(S2[k])
4          fixes.append(S1[k])
5      else
6          break
7  if k > size(S1):
8      repair_sites += S2[k :]
9  elseif k < size(S1):
10     fixes += S1[k :]
11 return repair_sites, fixes
```

the order of the expressions matters in SELECT. Therefore, the algorithm checks the equality in order and record the expression needs to be replaced in S_2 (line 1-6). If S_2 has more expressions than S_1, we list the redundant expressions as repair sites that are to be deleted (line 7-8); If S_1 has more expressions than S_2, the algorithm adds the extra expressions to the end of fixes (line 9-10). In the end, the user will be suggested to replace each expression in repair sites with each expression in fixes. If there are more expressions in repair sites/fixes, the user will be suggested to remove/add those expressions accordingly.

C LIMITATIONS DUE TO IMPLEMENTATION

In this section, we discuss the limitation of our approach due to its implementation, expanding on Section 6.2.

C.1 Limitation on Operators

The implementation of algorithms in Section 4 rely on the Z3 SMT Solver [16] to check whether an implication is valid between two formulas. Therefore, the operations and functions we support are limited by the Z3 solver. While most of the comparison and arithmetic operators, string concatenation and all logical connectives are supported by the Z3 solver, there are limitations for the "LIKE" operator. The Z3 solver does not support the *LIKE* operator while it does support checking the matching of suffix, prefix and string containment. Therefore, our framework supports the following:

(1) Check if a string expression is the prefix of another string expression (e.g. *a LIKE '%xxx'*).
(2) Check if a string expression is the suffix of another string expression (e.g. *a LIKE 'xxx%'*).
(3) Check if a string expression is contained in another string expression (e.g. *a LIKE '%xxx%'*).

C.2 Limitations on Aggregation Functions

In addition, Z3 Solver does not support aggregation functions (SUM, AVG, COUNT, MAX, MIN), so we define aggregation functions as function symbols in Z3 Solver and test input equivalence instead of the semantic equivalence of the aggregation functions. For example, we do not consider $AVG(x)$ to be equivalent to $\frac{SUM(x)}{COUNT(x)}$.

C.3 Limitaions on HAVING

While the predicates contain no aggregation function are not affected much by the Z3 limitations, the semantics of HAVING predicates involving aggregation function changes drastically as aggregation functions are declared as function symbols in Z3. Since Z3 does not consider group, aggregation functions are now semantically applied to each single tuple in the group instead of the entire group of tuples as we use Z3 to check equivalence of HAVING predicates. Therefore, Qr-Hint must guarantee two queries have the exact identical groups and each group has the exact same member tuples (i.e. equivalent GROUP BY clauses) before reporting any error in the HAVING clause (this is also a key reason why the workflow is determined as shown in Appendix D).

D WORKFLOWS AND GUARANTEES

In this section, we further elaborate the workflow of Qr-Hint and the guarantees Qr-Hint provides.

D.1 Qr-Hint Workflows

As mentioned in Section 3.5, Qr-Hint progresses in the order of FROM → WHERE/HAVING → GROUP BY → SELECT, and reports errors to users in the end. Given such progression, Qr-Hint can run in two different modes, namely interactive mode and static mode. In interactive mode, errors are reported to users clause by clause in the order of FROM → WHERE → GROUP BY → HAVING → SELECT. In static mode, all errors are reported to users at once following the same order.

D.1.1 Interactive Mode. When running in interactive mode, Qr-Hint starts by checking the equivalence of FROM clause and stop at the first clause where Qr-Hint detects inequivalence between two clauses. It then reports the errors to the user who can later invoke Qr-Hint again to check a modified query.

While Qr-Hint tests WHERE and HAVING clauses in conjunction, the order in which Qr-Hint reports errors differs from the aforementioned progressions. Specifically,

(1) If the formulas of WHERE and HAVING are equivalent, no error will be reported and Qr-Hint moves to GROUP BY clause.
(2) If the formulas of WHERE and HAVING are not equivalent, Qr-Hint proceeds differently depending on the clauses that contain the repair sites.
   - If repair sites are only in the WHERE clause, Qr-Hint reports errors in WHERE and stops.
   - If the repair sites are only in HAVING, Qr-Hint moves on to check GROUP BY clauses and reports errors in HAVING if GROUP BY clauses are equivalent. Otherwise it reports the errors in GROUP BY and stops. For checking GROUP BY, Qr-Hint assumes that the user will fix WHERE/HAVING eventually, and thus using the reference WHERE and HAVING formulas (i.e., P_1 and P_1' in Section 5) to construct the formula for the wrong query.
   - If repair sites exist in both WHERE and HAVING, Qr-Hint reports error in WHERE and stops.

Example 18. *Reconsidering Q_1 and Q_2 from Example 13, where the only error in Q_2 is* `COUNT(t3.year) = 1`. *Since this repair site*

resides in HAVING, Qr-Hint declares the WHERE clause as correct and move to GROUP BY and report the error in HAVING after detecting the GROUP BY clauses are equivalent.

We report errors in HAVING after WHERE and GROUP BY because that is the natural flow of query execution. Though we treat all aggregation functions as function symbols due to the limitation of the solver that we employ in our implementation (see Appendix C), we use this procedure to ensure that it is easy for users to follow. We will discuss the limitations of HAVING in details later in this appendix.

D.1.2 Automatic Mode. In the automatic mode, Qr-Hint runs through each clause regardless the existence of errors within the clause, and report all errors at once in the end. However, since errors from a former clause might affect testing a later one, we fix each clause in the following order:

- FROM
  - If a table is missing from the wrong query, we simply add the missing table to the wrong query and assign it an alias so that a mapping can be established between two queries. Such addition of table does not affect the subsequent clause because its attributes never appear in the subsequent clauses.
  - If a table is redundant in the wrong query, we add such table to the reference query so that a mapping can be established. This also does not affect the subsequent clauses because the attributes of this extra table do not appear in the subsequent clauses in the reference query.
- WHERE/HAVING will be tested as usual.
- GROUP BY will be tested with the assumption that the WHERE/HAVING clause of the wrong query is correct (i.e., when it comes to formula construction discussed in Section 5, we use P_1 and P_1' in place of P_2 and P_2').
- SELECT will also be tested with the same assumption as GROUP BY.

Example 19. *Consider the following two queries:*

```
Q1:
SELECT t2.author, t1.year
FROM inproceedings t1, authorship t2, inproceedings t3,
     authorship t4
WHERE t1.pubkey = t2.pubkey AND t3.pubkey = t4.pubkey
  AND t2.author = t4.author AND t1.area = t2.area
  AND t1.area = 'Database' AND t1.year <= t3.year
GROUP BY t2.author, t1.year
HAVING COUNT(DISTINCT t3.year) = 1;
Q2:
SELECT t4.author, t1.year
FROM inproceedings t1, authorship t2, inproceedings t3
WHERE t1.pubkey = t2.pubkey AND t1.year <= t3.year
GROUP BY t2.author, t1.year, t1.area
HAVING COUNT(t3.year) = 1 AND t1.area = t2.area AND
     t1.area = 'Database';
```

In automatic mode, Qr-Hint first detects a missing table `authorship t4` *in the FROM clause. It then add* `authorship t4` *to the FROM clause of* Q_2 *and proceed to WHERE/HAVING clause.*

Depending on the cost function and the number of repair sites, one combination of repair sites and fixes is to replace `COUNT(t3.year) = 1` *with* `COUNT(DISTINCT t3.year) = 1 AND t3.pubkey = t4.pubkey AND t2.author = t4.author` *when the number of repair sites is set to 1. Another combination is to replace* `t1.year <= t3.year` *and* `COUNT(t3.year)` *with* `t1.year <= t3.year AND t3.pubkey = t4.pubkey AND t2.author = t4.author` *and* `COUNT(DISTINCT t3.year) = 1` *respectively. In the former case, Qr-Hint reports errors in the WHERE clause while it declares WHERE clause as correct and move to GROUP BY in the later one.*

After finishing WHERE/HAVING clauses, Qr-Hint will recognize the GROUP BY clauses as equivalent and finally report the error in the HAVING clause found in the previous step.

D.2 Guarantees

In summary, Qr-Hint does not guarantee query equivalence detection test (which is an undecidable problem). Instead, we have the following guarantee.

Theorem D.1. *Given two inequivalent queries Q_1 and Q_2, Qr-Hint guarantees that $Q_1 \equiv Q_2$ after applying all fixes to the repair sites in all clauses of Q_2, according the flow mentioned in Appendix D.1.*

Proof. We prove the theorem by contradiction. By Qr-Hint all clauses of Q_1 and Q_2 become equivalent after fixes are applied to Q_2. We then first assume that $Q_1 \not\equiv Q_2$ while all of their corresponding clauses are equivalent. Such assumption implies that there exist a database instance D such that $Q_1(D)$ and $Q_2(D)$ differ by some tuple t (i.e. t_1 is either in $Q_1(D)$ or $Q_2(D)$ but not both).

We then go through each clause to examine how t might be produced.

Since FROM clauses are equivalent, they have the same multiset of tables and thus same cross-join results. Therefore, up to the execution of FROM clause, t cannot exist and Q_1 and Q_2 must be equivalent.

As stated previously in the limitations on HAVING, all aggregation functions are treated as function symbols and thus being evaluated on a per-tuple basis, so we discuss WHERE and HAVING together. Applying WHERE and HAVING conditions on the previous cross-join results and producing t implies that the WHERE and HAVING clauses are not equivalent, which contradicts the fact that they are equivalent after fixes are applied.

Since the execution result is the same up to WHERE/HAVING, the existence of t implies Q_1 and Q_2 group tuples differently. However, the GROUP BY clause of Q_1 and Q_2 are equivalent, so t also cannot be produced by the GROUP BY clause.

Finally, with all previous intermediate outputs being the same, the existence of t implies that there exists at least one SELECT expression e_i^1 and e_i^2 such that $e_i^1 \not\equiv e_i^2$ given the equivalence of all other clauses, and this contradicts the fact that SELECT clause are equivalent.

In summary, a tuple t that differs $Q_1(D)$ and $Q_2(D)$ cannot exist when evaluating both queries on some database instances. Therefore, by making all clauses equivalent, Qr-Hint guarantee query equivalence after fixes are applied. □

The fixed order for examining queries in QR-HINT does not consider all possible dependencies among different clauses, which usually have great influence among each other.

EXAMPLE 20. *Consider the following Q_1 and Q_2:*

```
Q1:
SELECT booktitle FROM inproceedings
WHERE year > 2000 AND area = 'Database'
AND title LIKE '%Query'
GROUP BY booktitle
HAVING MAX(year) > 2010

Q2:
SELECT booktitle FROM inproceedings
WHERE area = 'Database' AND title LIKE 'Query'
GROUP BY booktitle
HAVING MAX(year) > 2010
```

The only mistake in Q_2 is the predicate `title LIKE 'Query'` *where a "%" is missing.* `year > 2000` *is not necessary to be in Q_1 as* `MAX(year) > 2010` *will eventually filter out the* `booktitle` *that do not have year larger than 2010, so omitting it in Q_2 does not change the semantics of the query. However, QR-HINT will suggest replacing* `title LIKE 'Query'` *with* `title LIKE '%Query' AND year > 2000` *in order to match the WHERE clause of Q_1. While it is not required to have* `year > 2000` *as part of the fixes, QR-HINT does guarantee the equivalence of query after fixes are applied.*

E MISSING PROOFS

PROOF OF PROPOSITION 1. Without loss of generality, suppose that Q_1 returns a non-empty result over some database instance D. We construct a new database instance D' as follows: 1) For each unique table $T \in \text{Tables}(Q_1)$, duplicate each row in T a number of times equal to a unique prime p_T (such that $p_{T_1} \neq p_{T_2}$ for any $T_1 \neq T_2$). 2) For each table $T \notin \text{Tables}(Q_1)$, make T empty. We have

$$|Q_1(D')|/|Q_1(D)| = \prod_{\text{unique } T \in \text{Tables}(Q_1)} p_T^{|\text{Aliases}(Q_1,T)|}.$$

For any Q_2 equivalent to Q_1, it must be the case that $\text{Tables}(Q_2) \subseteq \text{Tables}(Q_1)$; otherwise $Q_2(D')$ would be empty by construction of D'. Furthermore, note that for Q_2 to be equivalent to Q_1, we must have $|Q_2(D')|/|Q_2(D)| = |Q_1(D')|/|Q_1(D)|$, so

$$\prod_{\text{unique } T \in \text{Tables}(Q_2)} p_T^{|\text{Aliases}(Q_2,T)|} = \prod_{\text{unique } T \in \text{Tables}(Q_1)} p_T^{|\text{Aliases}(Q_1,T)|}.$$

It then follows from the Prime Factorization Theorem that $\text{Tables}(Q_2)$ and $\text{Tables}(Q_2)$ contain the same *set* of tables, and that for each unique $T \in \text{Tables}(Q_1)$ (or $\text{Tables}(Q_2)$), $|\text{Aliases}(Q_2, T)| = |\text{Aliases}(Q_1, T)|$. In other words, $\text{Tables}(Q_2) = \text{Tables}(Q_1)$ (multiset equality).

□

PROOF OF THEOREM 1. **(If).** Assume that there exists a fix F for a set of repair sites S in P_2. By Lemma 4.1, we can create a lower bound $P_{2\perp}$ and an upper bound $P_{2\top}$ such that $P_2 \in [P_{2\perp}, P_{2\top}]$ with respect to S. Meanwhile, replacing each subtree in S with its counterpart in F creates a new formula P_2' such that $P_2' \in [P_{2\perp}, P_{2\top}]$ because there is an one-to-one correspondence between elements in S and F. Since $P_2' \equiv P_1$ by the definitions of S and F, $P_1 \in [P_{2\perp}, P_{2\top}]$ must hold.

(Only If). Assume that there exists a lower bound $P_{2\perp}$ and upper bound $P_{2\top}$ for P_2 such that $P_1 \in [P_{2\perp}, P_{2\top}]$ with respect to S. By Lemma 4.2, if $P_1 \in [P_{2\perp}, P_{2\top}]$, there exists a fix F for the set of repair sites S. □

PROOF OF LEMMA 4.1. We use induction over the size of the syntax tree of P.

Base case. We first discuss the case where the size of P is 1 (i.e. P is a single atomic predicate). If such single predicate is a repair site, the lowerbound and the upperbound are created by replacing P with $\perp$ and $\top$ respectively, and it is obvious that $P \in [\perp, \top]$. If the single predicate is not a repair site, then both lowerbound and upperbound stays as P, and $P \in [P, P]$.

Induction step. Now we assume that CREATE-BOUNDS is correct given a P syntax tree of size up to n. We complete the proof of correctness by proving CREATE-BOUNDS is correct when P has a size of $n+1$. There are three possible cases.

In the case of P rooted at $\wedge$, the algorithm is guaranteed to find lowerbounds and upperbounds for both left and right subtrees since they have a size smaller or equal to n. Let P_l denote the original left subtree, $P_{l\perp}$, $P_{l\top}$ denote the bounds for the original left subtree. Similarly, we use notation P_r, $P_{r\perp}$, $P_{r\top}$ for the right subtree. Since $P \equiv P_l \wedge P_r$ and $P_l \wedge P_r \in [P_{l\perp} \wedge P_{r\perp}, P_{l\top} \wedge P_{r\top}]$, $P \in [P_{l\perp} \wedge P_{r\perp}, P_{l\top} \wedge P_{r\top}]$ must hold.

In the case of P rooted at $\vee$, the algorithm is guaranteed to find lowerbounds and upperbounds for both left and right subtrees since they have a size smaller or equal to n. Following the same notation in case $\wedge$, $P \in [P_{l\perp} \vee P_{r\perp}, P_{l\top} \vee P_{r\top}]$ must hold due to $P \equiv P_l \vee P_r$ and $P_l \vee P_r \in [P_{l\perp} \vee P_{r\perp}, P_{l\top} \vee P_{r\top}]$.

In the case of P rooted at $\neg$, the algorithm is guaranteed to find lowerbounds and upperbounds for the only subtrees since it has a size of n. Let P' denote the subtree and $P'_\perp$, $P'_\top$ denote the lowerbound and upperbound. Since $P' \in [P'_\perp, P'_\top]$, $\neg P' \in [\neg P'_\top, \neg P'_\perp]$ and $\neg P' \equiv P$, $P \in [\neg P'_\top, \neg P'_\perp]$ must hold.

Since CREATE-BOUNDS always correctly creates a lowerbound and an upperbound for an arbitrary SQL logical formula P with respect to a set of its repair sites, Lemma 4.1 holds. □

PROOF OF LEMMA 4.2. The proof is by induction over the size of the syntax tree of the incorrect formula.

Base case. We first discuss the case where the size of P is 1 (i.e. P is a single atomic predicate). In this case, P must be a repair site and $T \in [T_\perp, T_\top]$ and $T_\perp \equiv T_\top \equiv P^\star$. Therefore, $T \equiv P^\star$ after MINIMIZE-FIX($T_\perp$, $T_\top$) and replacing P with T trivially makes P equivalent to $P^\star$.

Induction step. Now we assume that DERIVE-FIXES is correct given a P syntax tree of size up to n. We complete the proof of correctness by proving DERIVE-FIXES is correct when P has a size of $n+1$. There are three possible cases.

In the case of P rooted at $\wedge$, the algorithm is guaranteed to find fixes for both the left and right subtrees, and the resulting formulas rooted at P_l and P_r (we denote them by T_l and T_r) after fixes applied are guaranteed to fall in to the bounds $[T_\perp \vee P_{l\perp}, (T_\top \vee P_{l\perp} \vee \neg P_{r\top}) \wedge P_{l\top}]$, $T_r \in [T_\perp \vee P_{r\perp}, (T_\top \vee P_{r\perp} \vee \neg P_{l\top}) \wedge P_{r\top}]$ respectively because the left and right subtrees have a size smaller than n.

Due to Lemma 4.1, the following holds at P:

$$P_{l\perp} \wedge P_{r\perp} \implies T_\perp$$
$$T_\top \implies P_{l\top} \wedge P_{r\top}$$
$$T_\perp \in [P_{l\perp} \wedge P_{r\perp}, P_{l\top} \wedge P_{r\top}]$$
$$T_\top \in [P_{l\perp} \wedge P_{r\perp}, P_{l\top} \wedge P_{r\top}]$$

We thus can further infer that

$$T_\perp \vee P_{l\perp} \implies (T_\top \vee P_{l\perp} \vee \neg P_{r\top}) \wedge P_{l\top}$$
$$T_\perp \vee P_{r\perp} \implies (T_\top \vee P_{r\perp} \vee \neg P_{l\top}) \wedge P_{r\top}$$
$$P_{l\perp} \implies T_\perp \vee P_{l\perp}$$
$$(T_\top \vee P_{l\perp} \vee \neg P_{r\top}) \wedge P_{l\top} \implies P_{l\top}$$
$$P_{r\perp} \implies T_\perp \vee P_{r\perp}$$
$$(T_\top \vee P_{r\perp} \vee \neg P_{l\top}) \wedge P_{r\top} \implies P_{r\top}$$

We then prove that $T_l \wedge T_r \in [T_\perp, T_\top]$. First, we know:

$$T_l \wedge T_r \in [(T_\perp \vee P_{l\perp}) \wedge (T_\perp \vee P_{r\perp}), \\ ((T_\top \vee P_{l\perp} \vee \neg P_{r\top}) \wedge P_{l\top}) \wedge ((T_\top \vee P_{r\perp} \vee \neg P_{l\top}) \wedge P_{r\top})]$$

due to:

$$T_l \in [T_\perp \vee P_{l\perp}, (T_\top \vee P_{l\perp} \vee \neg P_{r\top}) \wedge P_{l\top}]$$
$$T_r \in [T_\perp \vee P_{r\perp}, (T_\top \vee P_{r\perp} \vee \neg P_{l\top}) \wedge P_{r\top}]$$

Combining the lowerbounds of T_l and T_r, we get:

$$\begin{aligned} & (T_\perp \vee P_{l\perp}) \wedge (T_\perp \vee P_{r\perp}) \\ \equiv\ & T_\perp \vee (P_{l\perp} \wedge P_{r\perp}) \\ \equiv\ & T_\perp \qquad\qquad [P_{l\perp} \wedge P_{r\perp} \implies T_\perp] \end{aligned}$$

Combining the upperbounds of T_l and T_r, we get:

$$\begin{aligned} & ((T_\top \vee P_{l\perp} \vee \neg P_{r\top}) \wedge P_{l\top}) \wedge ((T_\top \vee P_{r\perp} \vee \neg P_{l\top}) \wedge P_{r\top}) \\ \equiv\ & (T_\top \wedge P_{l\top} \wedge P_{r\top}) \vee \\ & ((\neg P_{r\top} \vee P_{l\perp}) \wedge (\neg P_{l\top} \vee P_{r\perp}) \wedge P_{l\top} \wedge P_{r\top}) \\ \equiv\ & (T_\top \wedge P_{l\top} \wedge P_{r\top}) \vee (P_{l\top} \wedge P_{r\perp} \wedge P_{l\perp} \wedge P_{r\top}) \\ \equiv\ & T_\top \vee (P_{l\perp} \wedge P_{r\perp}) \ [T_\top \implies P_{l\top} \wedge P_{r\top},\ P_{l\perp} \implies P_{l\top},\ P_{r\perp} \implies P_{r\top}] \\ \equiv\ & T_\top \qquad\qquad [P_{l\perp} \wedge P_{r\perp} \implies T_\top] \end{aligned}$$

Therefore, the resulting $T_l \wedge T_r \in [T_\perp, T_\top]$. Since $T_\perp \equiv T_\top \equiv P^\star$ at P, $T_l \wedge T_r \equiv P^\star$.

In the case of P rooted at $\vee$, the algorithm is guaranteed to find fixes for both the left and right subtrees, and the resulting formulas rooted at P_l and P_r (we denote them by T_l and T_r) after fixes applied are guaranteed to fall in to the bounds $[(T_\perp \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp},\ T_\top \wedge P_{l\top}]$, $[(T_\perp \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp},\ T_\top \wedge P_{r\top}]$ respectively because the left and right subtrees have a size smaller than n.

Due to Lemma 4.1, the following holds at P:

$$P_{l\perp} \vee P_{r\perp} \implies T_\perp$$
$$T_\top \implies P_{l\top} \vee P_{r\top}$$
$$T_\perp \in [P_{l\perp} \vee P_{r\perp}, P_{l\top} \vee P_{r\top}]$$
$$T_\top \in [P_{l\perp} \vee P_{r\perp}, P_{l\top} \vee P_{r\top}]$$

We thus can further infer that

$$(T_\perp \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp} \implies T_\top \wedge P_{l\top}$$
$$(T_\perp \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp} \implies T_\top \wedge P_{r\top}$$
$$P_{l\perp} \implies (T_\perp \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp}$$
$$T_\top \wedge P_{l\top} \implies P_{l\top}$$
$$P_{r\perp} \implies (T_\perp \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp}$$
$$T_\top \wedge P_{r\top} \implies P_{r\top}$$

We then prove that $T_l \wedge T_r \in [T_\perp, T_\top]$. First, we know:

$$T_l \vee T_r \in [((T_\perp \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp}) \vee ((T_\perp \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp}), \\ (T_\top \wedge P_{l\top}) \vee (T_\top \wedge P_{r\top})]$$

due to:

$$T_l \in [(T_\perp \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp},\ T_\top \wedge P_{l\top}]$$
$$T_r \in [(T_\perp \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp},\ T_\top \wedge P_{r\top}]$$

Combining the lowerbounds of T_l and T_r, we get:

$$\begin{aligned} & ((T_\perp \wedge P_{l\top} \wedge \neg P_{r\perp}) \vee P_{l\perp}) \vee ((T_\perp \wedge P_{r\top} \wedge \neg P_{l\perp}) \vee P_{r\perp}) \\ \equiv\ & (T_\perp \vee P_{l\perp} \vee P_{r\perp}) \wedge ((P_{l\top} \wedge \neg P_{r\perp}) \vee (\neg P_{l\perp} \wedge P_{r\top}) \vee P_{l\perp} \vee P_{r\perp}) \\ \equiv\ & (T_\perp \vee P_{l\perp} \vee P_{r\perp}) \wedge (P_{l\top} \vee P_{r\perp} \vee P_{l\perp} \vee P_{r\top}) \\ \equiv\ & T_\perp \wedge (P_{l\top} \vee P_{r\top}) \ [P_{l\perp} \wedge P_{r\perp} \implies T_\perp,\ P_{l\perp} \implies P_{l\top},\ P_{r\perp} \implies P_{r\top}] \\ \equiv\ & T_\perp \qquad\qquad [T_\perp \implies P_{l\top} \vee P_{r\top}] \end{aligned}$$

Combining the upperbounds of T_l and T_r, we get:

$$\begin{aligned} & (T_\top \wedge P_{l\top}) \vee (T_\top \wedge P_{r\top}) \\ \equiv\ & T_\top \wedge (P_{l\top} \vee P_{r\top}) \\ \equiv\ & T_\top \qquad\qquad [T_\top \implies P_{l\top} \vee P_{r\top}] \end{aligned}$$

Therefore, the resulting $T_l \vee T_r \in [T_\perp, T_\top]$. Since $T_\perp \equiv T_\top \equiv P^\star$ at P, $T_l \vee T_r \equiv P^\star$.

In the case of P rooted at $\neg$, the algorithm is guaranteed to find fixes for its only subtree, and the resulting formula T_{child} rooted at P_{child} after fixes applied is guaranteed to fall into the bounds $[\neg T_\top, \neg T_\perp]$ because the left and right subtrees have a size smaller than n. Therefore, after applying the negation, we get $\neg T_{child} \in [T_\perp, T_\top]$. Since $T_\perp \equiv T_\top \equiv P^\star$ at P, it is clear that $\neg T_{child} \equiv P^\star$.

Since Algorithm 3 always returns a formula equivalent to $P^\star$ in all three possible cases, Lemma 4.2 holds. □

Proof of Proposition 2. Note that the logical formulas P_1'' and P_2'' are defined on the same set of table aliases, both old and new. So these two logical formulas are defined on the same set of tuple variables, although some tuples/attributes may be used in one of them and not in the other. We say that two tuples t_1, t_2 from the cross product of the tables in the FROM clause of Q_1 (respectively, Q_2) **join over** P_1'' (respectively, P_2'') if we assign the old table aliases of Q_1 to t_1 and new table aliases of Q_1 to t_2, then the attribute values of t_1, t_2 satisfy the predicate P_1'' (respectively, P_2'').

(Only If). Assume the contradiction, that the GROUP-BY expressions of Q_1 and Q_2 are equivalent but $P_1' \not\equiv P_2'$. Then, without loss of generality, there exist two tuples t_1, t_2 such that they belong to the same group in Q_1 and Q_2, and join over P_1'', but do not join over P_2''. Since t_1, t_2 both satisfy the WHERE conditions of Q_1, Q_2, hence (i) $P_1 = P_2$ evaluates to true on t_1, and (ii) $P_1' = P_2'$ evaluates to true on t_2. Further, since t_1, t_2 belong to the same group in Q_2, all the GROUP BY expressions evaluate to the same value for both tuples, hence (iii) $g_1^2 = g_1^{2'} \wedge \ldots \wedge g_m^2 = g_m^{2'}$. Combining these, we get t_1, t_2 also join over P_2'' (see (3)), which is a contradiction. Hence $P_1'' \equiv P_2''$.

(If). Assume the contradiction, i.e., $P_1'' \equiv P_2''$ but the GROUP-BY clauses of Q_1 and Q_2 are not equivalent. Hence, without loss of generality, there must exist two tuples t_1, t_2 such that they join over both P_1'' and P_2'', and belong to the same group in Q_1 but do not belong to the same group in Q_2. However, joining over P_2'' implies that t_1, t_2 together satisfy P_2'', in particular satisfy that $g_1^2 = g_1^{2'} \wedge \ldots \wedge g_m^2 = g_m^{2'}$ (see (3)). Therefore, they have the same

values of all GROUP BY expressions in Q_2, and must belong to the same group in Q_2, which is a contradiction. Hence the GROUP BY clauses of Q_1, Q_2 must be equivalent. □

F USER STUDY

The following DBLP database schemas were used for the user study, we change the table name (`inproceedings` → `conference_paper`, `article` → `journal_paper`) in order to make them more intuitive for participants:

- `conference_paper: (pubkey, title, conference_name, year, area)`
- `journal_paper: (pubkey, title, journal_name, year)`
- `authorship: (pubkey, author)`

The area attribute in the conference_paper table can only be one of the following: "ML-AI", "Theory", "Database", "Systems" or "UNKNOWN".

The questions and the correct queries for the user study are shown in Table 1. The wrong queries and hints are shown in Table 2 (Hints from teaching assistants are in black, and hints from Qr-Hint are in blue). All hints are shown in the same order as they were shown to the participants. We run Qr-Hint in automatic mode to generate all hints at once.

Question Statement	Correct Query
Q_1: Find names of the authors, such that among the years when he/she published both conference paper and journal paper, 2 of the published papers are at least 20 years apart.	SELECT i1.author FROM conference_paper i1, conference_paper i2, journal_paper a1, journal_paper a2, authorship au1, authorship au2, authorship au3, authorship au4 WHERE i1.pubkey = au1.pubkey AND i2.pubkey = au2.pubkey AND a1.pubkey = au3.pubkey AND a2.pubkey = au4.pubkey AND au1.author = au2.author AND au2.author = au3.author AND au3.author = au4.author AND i1.year + 20 >= i2.year AND i1.year = a1.year AND i2.year = a2.year GROUP BY i1.author
Q_2: For each author who has published conference papers in the database area, find the number of their conference paper collaborators in the database area by years before 2018 (ignore the years when they have 0 collaborators). Your output should be in the format of (author, year, number of collaborators in that year).	SELECT t2.author, t1.year, COUNT(DISTINCT t3.author) FROM conference_paper t1, authorship t2, authorship t3 WHERE t1.pubkey = t2.pubkey AND t3.pubkey = t1.pubkey AND t3.author <> t2.author AND t1.year < 2018 AND t1.area = 'Database' GROUP BY t2.author, t1.year
Q_3: Excluding publications in the year of 2015, find authors who publish conference papers in at least 2 areas.	SELECT t1.author FROM conference_paper t1, authorship t2, conference_paper t3, authorship t4 WHERE t1.pubkey = t2.pubkey AND t2.author = t4.author AND t3.pubkey = t4.pubkey AND t1.year = t3.year AND t1.area <> t3.area AND t1.year <> 2015 AND t1.area <> 'UNKNOWN' AND t3.area <> 'UNKNOWN' GROUP BY t1.author
Q_4: Among the authors who publish in the Systems-area conferences, find the ones that have no co-authors on such publications (i.e. the author does not have any collaborator for any conference paper in systems area).	SELECT t2.author FROM conference_paper t1, authorship t2, authorship t3 WHERE t1.pubkey = t2.pubkey AND t2.pubkey = t3.pubkey AND t1.area = 'Systems' GROUP BY t2.author HAVING COUNT(DISTINCT t3.author) <= 1

Table 1: Question statements and correct queries in the user study.

Wrong Queries	Hints
Q_1: SELECT e.author FROM conference_paper a, authorship e, conference_paper b, authorship f, journal_paper c, authorship g, journal_paper d, authorship h WHERE a.pubkey = e.pubkey AND b.pubkey = g.pubkey AND c.pubkey = f.pubkey AND e.author = h.author AND d.pubkey = h.pubkey AND e.author = g.author AND f.author = h.author AND a.year + 20 > d.year GROUP BY e.author	1. In WHERE: You should change "a.year + 20 > d.year" to some other conditions.
Q_2: SELECT a.author, year, COUNT(*) FROM conference_paper, authorship, authorship a WHERE conference_paper.pubkey = a.pubkey AND authorship.pubkey = a.pubkey AND a.author <> authorship.author AND year < 2018 GROUP BY a.author, area, year, authorship.author HAVING area = 'Database' AND conference_paper.year < 2018	1. In GROUP BY: authorship.author is incorrect. 2. In SELECT: COUNT(*) is incorrect.
Q_3: SELECT b.author FROM conference_paper, authorship b, conference_paper a, authorship WHERE conference_paper.pubkey = authorship.pubkey AND a.year < 2015 OR a.year > 2015 AND b.author = authorship.author AND a.pubkey = b.pubkey AND conference_paper.year = a.year AND a.area <> conference_paper.area AND a.area <> 'UNKNOWN' AND conference_paper.area <> 'UNKNOWN' GROUP BY b.author	1. In WHERE, try to fix the whole condition by adding a pair of parentheses - in SQL AND takes higher precedence than OR (this fix alone should make the query correct) 2. In WHERE, you are missing a pair of parentheses around a.year < 2015 OR a.year > 2015. 3. GROUP BY is incorrect. 4. GROUP BY is incorrect without an aggregate function.
Q_4: SELECT a.author FROM authorship, conference_paper, authorship a WHERE conference_paper.pubkey = a.pubkey AND a.pubkey = authorship.pubkey GROUP BY a.author, conference_paper.area HAVING conference_paper.area = 'System' AND COUNT(DISTINCT a.author) <= 1	1. GROUP BY should not include t1.area. 2. In HAVING, conference_paper.area = 'System' should not appear. 3. In HAVING, try to fix conference_paper.area = 'System' (this plus another fix in HAVING will make the query right). 4. In HAVING, conference_paper.area = 'System' should be = 'Systems'. 5. In HAVING, try to fix COUNT(DISTINCT a.author) <= 1 (this plus another fix in HAVING will make the query right). 6. In HAVING, COUNT(DISTINCT a.author) <= 1 is referring to the same author attribute as the GROUP BY.

Table 2: Wrong queries and the hints provided.