Qr-Hint: Actionable Hints for Guided SQL Query Debugging

ABSTRACT

We describe a system called Qr-Hint that, given a (correct) target query $Q^\star$ and a (wrong) working query Q, both expressed in SQL, provides actionable hints for the user to fix the working query so that it becomes semantically equivalent to the target. It is particularly useful in an educational setting, where novices can receive help from Qr-Hint without requiring extensive personal tutoring. Since there are many different ways to write a correct query, we do not want to base our hints completely on how $Q^\star$ is written; instead, starting with the user's own working query, Qr-Hint purposefully guides the user through a sequence of steps that provably lead to a correct query, which will be equivalent to $Q^\star$ but may still "look" quite different from it. Ideally, we would like Qr-Hint's hints to lead to the "smallest" possible corrections to Q. However, optimality is not always achievable in this case due to some foundational hurdles such as the undecidability of SQL query equivalence and the complexity of logic minimization. Nonetheless, by carefully decomposing and formulating the problems and developing principled solutions, we are able to provide provably correct and locally optimal hints through Qr-Hint. We show the effectiveness of Qr-Hint through quality and performance experiments as well as a user study in an educational setting.

1 INTRODUCTION

In an era of widespread database usage, SQL remains a fundamental skill for those working with data. Yet, SQL's rich features and declarative nature can make it challenging to learn and understand. When students encounter difficulties in debugging their SQL queries, they often turn to instructors and teaching assistants for guidance. However, this one-on-one approach is limited in scalability. Syntax errors are easy to fix, but many queries contain subtle semantic errors that may require careful and time-consuming debugging. To save time, the teaching staff is often tempted to give students based on how the reference solution query is written, ignoring what students have written themselves, but doing so misses opportunities for learning. After all, a SQL query can be written in many ways that are different in syntax but nonetheless equivalent semantically. Seasoned teaching staff knows how to guide students through a sequence of steps that, starting with their own queries, lead them to a corrected version that is equivalent to the solution query but without revealing the solution query itself. Our goal is to build a system to help provide this service to students in a more scalable manner.

Example 1. *Consider the following database (keys are underlined) about beer drinkers and bars:* Likes(drinker, beer), Frequents(drinker, bar), Serves(bar, beer, price). *Suppose we want to write a SQL query for the following problem:* For each beer b that Amy likes and each bar r frequented by Amy that serves b, show the rank of r among all bars serving b according to price (e.g., if r serves b at the highest price, r's rank should be 1). We assume that there are no ties.

The reference solution query $Q^\star$ is given as follows:

```
SELECT L.beer, S1.bar, COUNT(*)
FROM Likes L, Frequents F, Serves S1, Serves S2
WHERE L.drinker = F.drinker AND F.bar = S1.bar
  AND L.beer = S1.beer AND S1.beer = S2.beer AND S1.price <= S2.price
GROUP BY F.drinker, L.beer, S1.bar
HAVING F.drinker = 'Amy';
```

Now consider a wrong student query Q:

```
SELECT s2.beer, s2.bar, COUNT(*)
FROM Likes, Serves s1, Serves s2
WHERE drinker = 'Amy'
  AND Likes.beer = s1.beer AND Likes.beer = s2.beer AND s1.price > s2.price
GROUP BY s2.beer, s2.bar;
```

It is not easy to suggest good hints to help students fix Q. First, there are many ways to write a query that is equivalent to $Q^\star$, and queries that look very different syntactically might be semantically similar or equivalent, so relying solely on the syntactic difference between Q and $Q^\star$ to propose fixes is ineffective and potentially misleading. In Example 1, even though $Q^\star$ has a HAVING clause, it would be confusing to suggest add HAVING to Q, because the condition drinker='Amy' in Q's WHERE serves the same purpose. Also, even though Q has Likes.beer=s2.beer in WHERE while $Q^\star$ has S1.beer=S2.beer, the difference is non-consequential because of the transitivity of equality. Yet another example is s1.price>s2.price in Q versus S1.price≤S2.price in $Q^\star$. It would be wrong to suggest changing $>$ to $\leq$ in Q, because an examination of the entire Q would reveal that the student intends s2 (and s1) in Q to serve the role of S1 (and S2) in $Q^\star$. The correct fix is actually changing $>$ to $\geq$.[1]

Second, it is often impossible to declare a part of Q as "wrong" since one could instead fix the remainder of Q to compensate for it. For example, we could argue that s1.price>s2.price in Q is "wrong," but there exists a correct query containing precisely this condition, e.g., with (s1.price>s2.price OR s1.price=s2.price). Hence, it is difficult to formally define what "wrong" means. Instead of basing our approach heuristically on calling out "wrong" parts, we formulate the problem as finding the "smallest repairs" to Q that make it *correct*.

Third, hints are for human users, so for a query with multiple issues—which is often the case in practice—we must be aware of the cognitive burden on users and not overwhelm them by asking them to make multiple fixes simultaneously. This desideratum introduces the challenge of planning the sequence of hints and defining appropriate intermediate goals.

Finally, effective hinting faces several fundamental barriers. Realistically, we cannot hope to always provide "optimal" hints because doing so entails solving the query equivalence problem for SQL, which is undecidable [1, 20, 32, 43]; even for decidable query fragments, Boolean expression minimization is known to be Σ_2^p [11].

To address the above challenges, we propose Qr-Hint, a system that, given a target query $Q^\star$ and a working query Q, is able to produce step-by-step hints for the user to edit the working query with the eventual goal of making it equivalent to $Q^\star$. The sequence of steps is guaranteed to lead the user on a correct path to eventual correctness. The following example illustrates how Qr-Hint can be used to help fix the query in Example 1 above.

[1] Another wrong hint would be to suggest changing COUNT(*) to COUNT(*)+1 in Q's SELECT instead of changing the inequality because doing so misses the top-ranked bars. Qr-Hint will not make such a mistake.

Example 2. *Continuing with Example 1, Qr-Hint automatically generates the sequence of hints below. Currently built for the teaching staff, Qr-Hint only generates the "repairs" below; using these repairs, the teaching staff would then hint the user in natural language. With the recent advances in generative AI chatbots, it would not be difficult to automate the natural language hints as well; the advantage of using Qr-Hint in that setting would be to provide provable guarantees on the quality of hints, which otherwise would be difficult, if not impossible, for generative AI to achieve by itself.*

Stage	*Qr-Hint repair*	*Hint in natural language*
FROM	*Frequents needed*	Are you sure your FROM include all tables you need? It looks like you are missing one—read the problem carefully and see what other piece of information you need.
WHERE	*s1.price>s2.price* ↦ *s1.price≥s2.price*	Your WHERE has a small problem with s1.price>s2.price. Think through some concrete examples and see how you may fix it.

Note the sequential nature of the hints above; the working query constantly evolves. Qr-Hint first focuses on FROM *and will only proceed to* WHERE *after* FROM *is "viable." After adding Frequents to* FROM*, the user will also need to add appropriate join conditions in* WHERE*; if these were not added correctly, the second step above would suggest additional repairs. It turns out that for this example, only the above two hints are needed to fix the query. In particular, Qr-Hint knows* not *to suggest spurious hints such as adding to Frequents.drinker to* GROUP BY *or changing s2.beer to Likes.beer in* SELECT.

We make the following contributions:

- We develop a novel framework that allows Qr-Hint to provide step-by-step hints to fix a working SQL query with the goal of making it equivalent to a target query. This framework formalizes the notion of "correctness" for a sequence of hints, allowing Qr-Hint to guarantee that every hint is actionable and is on the right path to achieve eventual correctness. Further, by formulating the hinting problem in terms of finding repair sites in Q with viable fixes, we are able to quantify the quality of the hints.
- Since the optimality of hints, in general, is impossible to achieve due to the foundational hurdles discussed earlier, we aim to provide guarantees on the "local" optimality of Qr-Hint in each step. We design practical algorithms with sensible trade-offs between optimality and efficiency.
- We evaluate the performance and efficacy of Qr-Hint experimentally. We further perform a user study involving students from current/past database courses offered at the authors' institution. Our findings indicate that Qr-Hint finds repairs that are optimal or close to optimal in practice under reasonable time, and they lead to hints that are helpful for students.

2 RELATED WORK

Debugging with reference queries. As for previous work that directly relates to Qr-Hint, XData [14] checks the correctness of a query by running the query on self-generated testing datasets based on a set of pre-defined common errors. Chandra et. al. [13] developed a grading system which canonicalizes queries by applying rewrite rules, and then decides partial credits based on a tree-edit distance between logical plans. QueryVis [33] turns queries into intuitive diagrams, helping users understand the difference between (non-monotone) queries with nested sub-queries through execution flows. Using a large database instance, RATest [36] utilizes data provenance to generate a small illustrative instance to differentiate queries. Gilad et. al. [23] propose an approach using c-tables [28] to form small abstract conditional instances to differentiate two queries. Nonetheless, none of them pinpoints errors in the original query or suggests feasible edits to correct the semantics. On the other hand, SQLRepair [39] fixes simple errors in an SPJ query by using Z3 SMT Solver [19] to synthesize/remove WHERE conditions until the wrong query produces the same outputs as the reference query over all testing instances. This does not guarantee query equivalence due to its test-driven nature.

Testing query equivalence. There are several classical results on query equivalence. Chandra et. al.[12] show that equivalence testing of conjunctive queries is NP-complete. Aho et. al. [4] propose tableau to represent the value of a query, which is used to give algorithms for checking equivalence of SPJ queries. Sagiv et. al. [41] give a procedure for testing the equivalence of Select-Project-Join-Difference-Union (SPJDU) queries. Klug [31] presents algorithms for checking the equivalence of conjunctive queries with inequalities. The equivalence problem of some classes of queries under bag semantics has been proved to be undecidable [29, 30]. While the query equivalence problem in general is undecidable [1, 43], tools are developed to check the equivalence of various classes of queries with restrictions and assumptions. Cosette [16–18] transforms SQL queries to algebraic expressions and uses a decision procedure and rewrite rules to check if the resulting expressions of two queries are equivalent. EQUITAS [47] develops a symbolic representation of SQL queries in first-order logic, and uses satisfiability modulo theories (SMT) to check query equivalence. WeTune [45] builds a query equivalence verifier by utilizing the U-semiring structure [16]. These tools for query equivalence do not give hints on how to modify one query to become equivalent to another.

Program Repair and Feedback for GPL. In the domain of program repair for general-purpose programming language (GPL), several types of approaches have been developed but none of them can be directly applied or easily transferred to cover SQL. First, a wrong program is usually aligned with reference program(s) ([3, 24, 44]) and fixes are generated based on the selected reference program using various techniques. Such approach is similar to Qr-Hint, but SQL is essentially different from GPL as SQL is declarative and GPLs are usually procedural. While it is possible to write programs in GPL to simulate the execution of a specific SQL query, there is no well-defined mapping between the syntax of SQL and any GPL. As a result, it is impossible to apply such program repair technique to SQL in general. Another approach is to leverage test cases to synthesize "patches" for the wrong program so that it returns the same output as the reference program for all test cases ([27, 37, 40, 42, 46]). However, such approach heavily relies on the test cases to cover all possible errors and thus usually fails to guarantee the semantic equivalence. Besides the traditional approaches, recent work explores ML algorithms to provide feedback and correction ([7, 8, 15, 25, 26, 34, 38]). In addition, large language models such as GPT-3 [10] have shown an ability to explain the semantics of SQL queries, but does not guarantee the correctness of fixes.

3 THE QR-HINT FRAMEWORK

Queries. We consider SQL queries that are select-project-join queries with an optional single level of grouping and aggregation.

For simplicity of presentation, we assume these are single-block SQL queries with SELECT (without DISTINCT), FROM (without JOIN operators), and WHERE (with condition defaulting to TRUE if missing) clauses,[2] together with optional GROUP BY and HAVING clauses. We refer to such a query as an *SPJA* query if it contains grouping or aggregation or DISTINCT; otherwise, we will call it an *SPJ* query.

We assume the default bag (multiset) semantics of SQL. Given query Q, let $\mathtt{F}(Q)$ denote the cross product of Q's FROM tables (including multiple occurrences of the same table, if any); and let $\mathtt{FW}(Q)$ denote the query that further filters $\mathtt{F}(Q)$ by Q's WHERE condition (i.e., $\mathtt{FW}(Q)$ is a SELECT * query with the same FROM and WHERE clauses as Q). Furthermore, if Q is SPJA, let $\mathtt{FWG}(Q)$ denote the (non-relational) query[3] that further groups the result rows of $\mathtt{FW}(Q)$ according to Q's GROUP BY expressions (or $\emptyset$ if there are none but Q contains aggregation nonetheless, in which case all result rows belong to a single group). Finally, if Q is SPJA, let $\mathtt{FWGH}(Q)$ denote the (non-relational) query that filters the groups of $\mathtt{FWG}(Q)$ according to Q's HAVING conditions (which defaults to TRUE if missing). When discussing equivalence (denoted $\equiv$) among above queries, we require that they return the same bag of result rows (ignoring row and column ordering) for any underlying database instance, and additionally, for queries returning groups, they return the same partitioning of result rows (ignoring group ordering).

SMT Solvers. As with previous work [16, 36, 47], we leverage *satisfiability modulo theory* (*SMT*) solvers to implement various primitives used by our system. Such a solver can decide whether a formula, modulo the theories it references, is satisfiable, unsatisfiable, or unknown (beyond the solver's capabilities). Specifically, we use the popular SMT solver Z3 [19] to implement the following three primitives. Give two quantifier-free expressions, $\mathsf{IsEquiv}(e_1, e_2)$ tests whether $e_1 \Leftrightarrow e_2$ (for logic formulae such as those in WHERE) or $e_1 = e_2$ (for value experssions such as those in SELECT or GROUP BY). Given a logic formula p, $\mathsf{IsUnSatisfiable}(p)$ and $\mathsf{IsSatisfiable}(p)$ return, respectively, whether p is satisfiable or unsatisfiable, respectively. All above primitives may return "unknown" when Z3 is unsure about its answer. However, when they return true, Z3 guarantees that the answer is not a false positive. Our algorithms in subsequent sections act only on (true) positive answers from these primitives. For complex uses, it is often convenient to frame equivalence/satisfiability testing using a *context* C, or a set of logical assertions (e.g., types declaration, known constraints, and inference rules) under which testing is done. We use subscript to specify the context: e.g., $\mathsf{IsUnSatisfiable}_C(p)$ is a shorthand for $\mathsf{IsUnSatisfiable}\left((\wedge_{c \in C} c) \wedge p\right)$.

Example 3. *Consider a query with a* WHERE *condition stipulating that* $A > 100$ *for an* INT*-typed column A, as well as a* HAVING *condition* $\mathtt{MAX}(A) \geq 101$. *We might wonder whether the* HAVING *condition is unnecessary. To this end, we call* $\mathsf{IsUnSatisfiable}_C(p)$ *with*

$$C : \left\{ \begin{array}{r} \mathbf{A} \textit{ has type } \textit{Array}(\mathbb{N}) \\ \forall i \in \mathbb{N} : \mathbf{A}[i] > 100 \\ \mathtt{MAX} \textit{ has type } \textit{Array}(\mathbb{N}) \rightarrow \mathbb{N} \\ \forall i \in \mathbb{N}, \mathbf{X} \textit{ of type } \textit{Array}(\mathbf{N}) : \mathtt{MAX}(\mathbf{X}) \geq \mathbf{X}[i] \end{array} \right\}, \quad p : \neg(\mathtt{MAX}(\mathbf{A}) \geq 101).$$

The first two assertions in C *are derived from the type of* A *and the* WHERE *conditions; here the array-typed* $\mathbf{A}$ *refers to a collection of* A *values. The last two specify (some) general inference rules on the SQL aggregate function* MAX. *Z3 correctly returns true, meaning that* $\mathtt{MAX}(A) \geq 101$ *must be true under* C *and is therefore unnecessary.*

Our use of Z3 for reasoning with SQL aggregation, such as the example above, goes beyond the practice in previous work, where aggregation functions are mostly treated as uninterpreted functions. For example, to test equality of two aggreates, [47] conservatively checks whether input value sets or multisets for the aggregate function are equal. In contrast, we encode properties of SQL aggregation functions in a way that allows Z3 to reason with them. As formulae become more complicated, e.g., with quantifiers and arrays, Z3 no longer offers a complete decision procedure (as there exists no decision procedure for first-order logic) and may return "unknown" more often. Nonetheless, practical heuristics employed by Z3 allow it to handle many cases of practical uses to Qr-Hint.

3.1 Approach

Given a (*syntactically correct*) working query Q and a target query $Q^\star$, Qr-Hint provides hints in *stages* to help the user edit the working query incrementally until it becomes *semantically equivalent* to $Q^\star$. Each stage focuses on one specific syntactic fragment of the working query. Qr-Hint gives actionable hints for the user to edit this fragment with the aim of bringing Q a step "closer" to being equivalent to $Q^\star$. Qr-Hint strives to suggest the smallest edits possible and avoid suggesting unnecessary edits. Upon passing a *viability check*, the working query Q clears the current stage and moves on to the next stage. After clearing all stages, Qr-Hint guarantees that $Q \equiv Q^\star$ (even if syntactically they are still different).

We now briefly outline the concrete stages of Qr-Hint; the details will be presented in the subsequence sections.

For an SPJ query, there are three stages. (1) We start with Q's FROM clause (Section 4) and make sure that its list of tables can eventually lead to a correct query; following this stage, $\mathtt{F}(Q) \equiv \mathtt{F}(Q^\star)$. (2) Next, we provide hints to repair Q's WHERE clause (Section 5) such that $\mathtt{FW}(Q) \equiv \mathtt{FW}(Q^\star)$, i.e., the repaired query returns the same sub-multiset of rows as $Q^\star$ that satisfy the WHERE clause, ignoring SELECT. (3) Finally, we handle Q's SELECT clause and ensure the working query returns correct output column values. Importantly, we make inferences of equivalence under the premise that all rows before SELECT already satisfy WHERE; this use of WHERE allows us to infer more equivalent cases and avoid spurious hints.

For an SPJA query, there are five stages. (1) The *first stage* handles FROM as in the SPJ case. (2) The *second stage* handles WHERE, but with a twist. As we have seen from Example 1, some condition can be either WHERE or HAVING, and it would be misleading to hint its absence from WHERE to be wrong; hence, Qr-Hint will look "ahead" at the two queries' HAVING and GROUP BY clauses to avoid misleading the user. At the end of this stage, instead of insisting that $\mathtt{FW}(Q) \equiv \mathtt{FW}(Q^\star)$ for the original $Q^\star$, we may rewrite $Q^\star$ (by legally moving some conditions between WHERE and HAVING) as needed first. (3) The *third stage* is GROUP BY, where we provide hints to edit Q's GROUP BY expressions to achieve equivalent grouping, i.e., $\mathtt{FWG}(Q) \equiv \mathtt{FWG}(Q^\star)$. Here, we infer equivalence under the premise that the rows to be grouped all satisfy WHERE. (4) The *fourth stage*

[2] We can handle a query with common table expressions (WITH) and subqueries in FROM that are aggregation-free, as well as non-outer JOINs in FROM, by rewriting the query into single-block SQL. SELECT DISTINCT is treated as grouping by all output columns.
[3] This query is non-relational because it returns, besides the underlying bag of rows from $\mathtt{FW}(Q)$, a partitioning of them into groups.

is HAVING, where we provide hints to repair Q's HAVING condition in the same vein as WHERE; however, inferences in this stage would additionally consider both WHERE and GROUP BY, and they are more challenging because of aggregation functions. After this stage, we have $\texttt{FWGH}(Q) \equiv \texttt{FWGH}(Q^\star)$. (5) The *fifth and final stage* is SELECT, which is similar to the SPJ case, but with the challenge of handling aggregation functions while simultaneously considering WHERE, GROUP BY, and HAVING.

Progress and Correctness. Note that to clear a stage, the user only needs to come up with a fix to pass the viability checks up to this stage. Even though QR-HINT may examine the queries in their entirety, the user does not have to think ahead about how to make the entire query correct.[4] Moreover, once a stage is cleared, QR-HINT never requires the user to come back to fix the same fragment again. This stage-by-stage design with "localized" hints helps limit the cognitive burden on the user.

The following theorem formalizes the intuition that this stage-based approach leads to steady, forward progress toward the goal of fixing the working query. It follows from the observation that our solution for each stage ensures the properties asserted below, which we will show stage by stage in the subsequent sections.

THEOREM 3.1. *Let $Q_0 = Q$ denote the initial working query and $Q^\star$ denote the target query. Let V_i denote the viability check for stage i, and Q_i denote the working query upon clearing stage i, where Q_i satisfies $V_1, V_2, \ldots, V_i$. We say that two queries are* stage-i consistent *if they are identical syntactically except in the fragments that stage $i+1$ and beyond focus on. For each stage i, the following hold:*

(Hint leads to fix) *If Q_{i-1} fails to satisfy V_i, there exists a query $\hat{Q}_i$ such that $\hat{Q}_i$ satisfies $V_1, V_2, \ldots, V_i$, $\hat{Q}_i$ is stage-$(i-1)$ consistent with Q_{i-1}, and $\hat{Q}_i$ follows the stage-i hint provided by QR-HINT.*

(Fix leads to eventual correctness) *There exists a query $\hat{Q}$ such that $\hat{Q} \equiv Q^\star$ and $\hat{Q}$ is stage-i consistent with Q_i.*

Optimality. Ideally, we would like QR-HINT to suggest the "best possible" hints, e.g., those leading to minimum edits to the working query. Unfortunately, it is impossible for any system to provide such a guarantee in general, because doing so entails being able to determine the equivalence of SQL queries: if $Q \equiv Q^\star$ to begin with, the system should not suggest any fix. It is well-known that the equivalence of first-order queries with only equality comparisons is undecidable [1]. Under bag semantics, even the decidability of equivalence of conjunctive queries has not been completely resolved [32]. Once we open up to the full power of SQL, which can express integer arithmetic, even equivalence of selection predicates becomes undecidable via a simple reduction to the satisfiability of Diophantine equations [20].

Given the foundational hurdles above, QR-HINT seeks a pragmatic solution. Instead of offering any global guarantee on the optimality of its hints, which is impossible, QR-HINT establishes, for each stage, guarantees on the necessity or minimality of its hints under certain assumptions. For example, for the FROM stage, QR-HINT guarantees its suggested fixes are optimal for SPJ queries, but for some SPJA queries, it may suggest a fix that turns out to be unnecessary. As another example, for the WHERE stage, the optimality of QR-HINT depends on, among other things, Z3-based primitives offering *complete* decision procedures. In each subsequence section, we will state any such assumption explicitly.

Finally, it is important to note that QR-HINT's progress and correctness properties (Theorem 3.1) do *not* rely on these assumptions. In the worst case, the user may be hinted to make some fixes that are unnecessary or unnecessarily big, but QR-HINT will still ensure that the user gets a correct working query in the end.

Features Not Currently Handled. This paper assumes that all database columns are NOT NULL. We believe that with some additional effort and complexity, QR-HINT can be extended to handle NULL using the technique in [47] of encoding each column with a pair of variables in Z3 (one for its value and the other a Boolean representing whether it is NULL). The same applies to OUTER JOIN. Also, QR-HINT does not currently incorporate database constraints (such as keys and foreign keys). While we can, in theory, encode some constraints as logical assertions and include them as part of the context when calling Z3, these assertions (with quantifiers) can significantly hamper Z3's performance. Future work is needed to develop more robust algorithms for incorporating constraints.

4 FROM STAGE

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

The viability check (Theorem 3.1, stage 1) for FROM is simple:

$$V_1 : \text{Check if } \mathsf{Tables}(Q) \stackrel{\square}{=} \mathsf{Tables}(Q^\star)$$

where $\stackrel{\square}{=}$ denotes multiset equality. If the working query Q fails the viability check, QR-HINT simply hints, for each table T whose counts in $\mathsf{Tables}(Q)$ and $\mathsf{Tables}Q^\star$ differ (including cases where T is used in one query but not the other), that the user should consider using T more or less to make the counts the same. It is straightforward to see that this hint leads to a fix that makes $\mathsf{Tables}(Q) \stackrel{\square}{=} \mathsf{Tables}(Q^\star)$, which enables the user to further edit Q into some $\hat{Q} \equiv Q^\star$ without retouching FROM: at the very least, one can make $\hat{Q}$ isomorphic to $Q^\star$ up to the substitution of table references with those in $\mathsf{Aliases}(Q)$. This observation establishes the progress and correctness properties (see Theorem 3.1) of FROM-stage hints, which we state below along with the remark that $\mathsf{F}(Q) \equiv \mathsf{F}(Q^\star)$ after this stage.

LEMMA 4.1. *QR-HINT's* FROM*-stage hint leads to a fixed working query Q_1 that (1) passes the viability check V_1 $\mathsf{Tables}(Q_1) \stackrel{\square}{=} \mathsf{Tables}(Q^\star)$; (2) satisfies $\mathsf{F}(Q_1) \equiv \mathsf{F}(Q^\star)$; and (3) leads to eventual correctness.*

While the correctness of the FROM-stage hint is straightforward, its optimality is surprisingly strong and has interesting proof. The

[4] In some cases, just to maintain syntactic correctness, a fix may necessitate trivial edits to fragments handled in future stages: e.g., if we remove a table from FROM, we will need to remove references to this table in the rest of the query. However, the user never needs to worry about making those edits semantically correct—that responsibility falls on future stages.

following lemma states that the viability check is, in fact, necessary—regardless of what could be done in WHERE and SELECT—under reasonable assumptions.

Lemma 4.2. *Two SPJ queries $Q^\star$ and Q cannot be equivalent under bag semantics if $Tables(Q^\star) \stackrel{\square}{\neq} Tables(Q)$ assuming no database constraints are present, and there exists some database instance for which either $Q^\star$ or Q returns a non-empty result.*[5]

The proof is in full version [21], which starts with a database instance D where one of $Q^\star, Q$ returns non-empty results, creates another instance D' by duplicating the tuples in D number of times equal to a unique prime for some tables, and then arguing using the Prime Factorization Theorem [22] that the queries must have the same results on D', they must have the same multi-set of tables.

Table Mappings. To facilitate analysis in subsequent stages, Qr-Hint needs a way to "unify" table and column references in Q and $Q^\star$ so that all of them use the same set of table aliases.

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

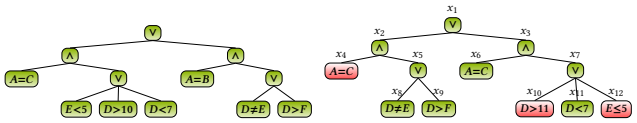
If the queries have no self-joins, it is straightforward to establish this mapping by table names. With self-joins, however, it can be tricky because we must match multiple roles played by the same table across queries. The information contained in FROM alone would be insufficient for matching. One approach is to explore every possible table mapping and select the one that leads to the minimum fix. Doing so would blow up complexity by a factor exponential in the number of self-joined tables. Qr-Hint instead opts for a heuristic that picks the single most promising table mapping. Details are in [21], but briefly, for each alias, we build a "signature" that captures how its columns are used by various parts of the query in a canonical fashion. We define a distance (cost) metric for the signatures. Then, for each table involved in self-joins, to determine the mapping between its aliases in Q and $Q^\star$, we construct a bipartite graph consisting of these aliases and solve the minimum-cost bipartite matching problem. We illustrate the high-level idea using the example below.

Example 4. *Continuing with Example 1, the following are signatures (one per column) for s1 and s2 in $Q^\star$ and s1 and s2 in Q.*

		s1 in $Q^\star$	*s2 in $Q^\star$*	*s1 in Q*	*s2 in Q*
WHERE & HAVING	*bar:*	$=\{F.bar\}$	$=\{F.bar\}$	*None*	*None*
	beer:	$=\{L.beer, S2.beer\}$	$=\{L.beer, S2.beer\}$	$=\{Likes.beer, s2.beer\}$	$=\{Likes.beer, s2.beer\}$
	price:	$\leq\{S2.price\}$	$\geq\{S2.price\}$	$>\{s2.price\}$	$<\{s1.price\}$
GROUP BY		$\{bar, beer\}$	$\{beer\}$	$\{beer\}$	$\{beer\}$
SELECT	*bar:*	$\{2\}$	$\emptyset$	$\emptyset$	$\{2\}$
	beer:	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$
	price:	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

For example, s1.beer's WHERE/HAVING *signature says that it is involved in an equality comparison with both L.beer and S2.beer; the latter is inferred—Qr-Hint automatically adds column references and constants that obviously belong to the same equivalence class. Likewise, s1's* GROUP BY *signature includes both bar and beer, with the latter added because of its equivalence to the* GROUP BY *column L.beer. When comparing signatures, all aliases are replaced by table names (which is a heuristic simplification); therefore, all four* WHERE/HAVING *signatures above for beer are considered the same. In this case, what makes the difference in bipartite matching turns out to be the* SELECT *signatures for bar, which clearly favors the mapping with s1 ↦ s2 and S2 ↦ s1.*

[5]The assumption of a not-always-empty result may seem out of the blue but is necessary. For example, queries SELECT 1 FROM R WHERE FALSE and SELECT 1 FROM R, R WHERE FALSE are equivalent—both always return empty results. However, if at least one of $Q^\star$ and Q can return non-empty results, $Tables(Q^\star) \stackrel{\square}{=} Tables(Q)$ becomes necessary for equivalence. Our proof of Lemma 4.2, in fact, builds on such a non-empty result.



(a) $P^\star$ (b) P

Figure 1: Syntax trees for $P^\star$ and P in Example 5.

Once we have selected the table mapping $\mathfrak{m}$, we can then "unify" $Q^\star$ and Q. For convenience, we simply rename each alias a in $Q^\star$ to $\mathfrak{m}(a)$; in subsequent sections, we shall assume that $Q^\star$ and Q have consistent column references.

5 WHERE STAGE

WHERE is the most involved stage of Qr-Hint. Recall from Section 3.1 that following WHERE stage, we should achieve $\mathsf{FW}(Q^\star) \equiv \mathsf{FW}(Q)$. Let P and $P^\star$ denote the WHERE predicates in Q and $Q^\star$, respectively. We assume that they have already been unified by the selected table mapping to have the same set of column references, as discussed in Section 4. The viability check for the WHERE stage (Theorem 3.1, stage 2) is simply that P is logically equivalent to $P^\star$:

$$V_2 : \text{Check if } P \Leftrightarrow P^\star$$

As discussed in Section 1, if $P \not\Leftrightarrow P^\star$, there are many different ways to modify P so that becomes logically equivalent to $P^\star$, and it is impossible to declare any part of P as definitively "wrong." Therefore, we suggest the smallest possible edits on P to reduce the cognitive burden on the user. We formalize the notion of "small edits" below. We represent P and $P^\star$ using syntax trees, where:

- Internal (non-leaf) nodes represent logical operators $\wedge$, $\vee$, and $\neg$. Let $\mathsf{op}(x)$ denote the operator associated with node x, and $\mathsf{Children}(x)$ denote the x's child nodes. If $\mathsf{op}(x)$ is $\neg$, $|\mathsf{Children}(x)| = 1$. If $\mathsf{op}(x) \in \{\wedge, \vee\}$, $|\mathsf{Children}(x)| \geq 2$.
- Leaf nodes are atomic predicates involving column references and/or literals. We treat each unique column reference as a free variable over the domain of the referenced column. We support basic SQL types and as well as standard comparison, arithmetic, and string operators to the extent supported by Z3, e.g.: $A{>}5$, $B{\leq}2C{-}10$, D LIKES 'Eve%'.

Example 5. *Consider the following logical formulae $P^\star$ and P where A, B, C, D, E are integers:*

$P^\star$: $(A{=}C \wedge (E{<}5 \vee D{>}10 \vee D{<}7) \vee (A{=}B \wedge (D{\neq}E \vee D{>}F))$
P: $(A{=}C \wedge (D{\neq}E \vee D{>}F)) \vee (A{=}C \wedge (D{>}11 \vee D{<}7 \vee E{\leq}5))$

Their respective syntax trees are shown in Figure 1.

Definition 2 (Repair for SQL Predicate). *Given a quantifier-free logical formulae P represented as a tree, a* repair *of P is a pair $(\mathcal{S}, \mathcal{F})$ where $\mathcal{S}$ is a set of disjoint subtrees of P called the* repair sites*, and $\mathcal{F}$ is function that maps each site $x \in \mathcal{S}$ to a new formulae $\mathcal{F}(x)$ called the* fix *for x. Given a target predicate $P^\star$, a repair $(\mathcal{S}, \mathcal{F})$*

Algorithm 1: RepairWhere$(x, x^\star, n)$

```
Input  : a wrong predicate x, a correct predicate x*, and a cap n on the number of
         repair sites
Output : a repair (S, F) with minimum cost
let S_o = ∅, F_o = ∅;
let c_o denote the minimum cost so far, and ∞ initially;
foreach set S of ≤n disjoint subtrees in x, in ascending |S| order do
    if Cost(S, ·) ≥ c_o then // cost due to # sites alone is already too big
        return (S_o, F_o);                                   // safe to stop now
    if x* ∈ CreateBounds(x, S) then
        let _, F = DeriveFixes(x, S, x*, x*);
        if c_o > Cost(S, F) then
            let S_o = S, F_o = F;
return (S_o, F_o);
```

Algorithm 2: CreateBounds$(x, \mathcal{S})$

```
Input  : a predicate x, and a set S of disjoint subtrees (repair sites) of x
Output : lower and bound bounds for x achievable by fixing S
if x ∈ S then return [false, true] ;
else if x is atomic then return [x, x] ;
else if op(x) ∈ {∧, ∨} then
    foreach c ∈ Children(x) do
        let [l_c, u_c] = CreateBounds(c, S[c]);[6]
    return [Θ_{c∈Children(x)} l_c, Θ_{c∈Children(x)} u_c] where Θ = op(x);
else // op(x) is ¬
    let c = Children(x)[0];                                  // the only child of x
    let [l_c, u_c] = CreateBounds(c, S[c]);
    return [¬u_c, ¬l_c];
```

for P is correct *if applying it to P—i.e., replacing each $x \in \mathcal{S}$ with $\mathcal{F}(x)$—results in a formulae P' such that $P' \Leftrightarrow P^\star$.*

Definition 3 (Cost of a repair). *Given target predicate $P^\star$, the* cost *of a repair $(\mathcal{S}, \mathcal{F})$ for P is:*

$$Cost(\mathcal{S}, \mathcal{F}) = w \cdot |\mathcal{S}| + \sum_{s \in \mathcal{S}} \frac{dist(s, \mathcal{F}(s))}{|P| + |P^\star|}, \text{ where} \quad (1)$$

$$dist(s, \mathcal{F}(s)) = |s| + |\mathcal{F}(s)|, \text{ and} \quad (2)$$

$w \in \mathbb{R}^+$ controls the relative weights of the cost components.

Here, we simply define $\mathsf{dist}(\cdot, \cdot)$ to be the number of nodes deleted and inserted by the repair; other notions of edit distance could be used too. The denominator under $\mathsf{dist}(\cdot, \cdot)$ serves to normalize the measure relative to the sizes of the queries. Also, note that the $w \cdot |\mathcal{S}|$ term adds a fixed penalty for each additional repair site. Intuitively, Qr-Hint will present all repair sites (without the associated fixes) to the user as a hint. Even a moderate number of repair sites will pose a significant cognitive challenge—if there were so many issues with P, we might as well ask the user to rethink the whole predicate (which would be a single repair site at the root). In our experiments (Section 9), we set $w = 1/6$, and the number of repair sites per WHERE rarely goes above two or three.

Example 6. *Consider Figure 1. One correct repair for P consists of three sites (x_4, x_{10}, x_{12}) and the corresponding fixes (A=B, D>10, E<5). The cost for this repair is $3w + \frac{3\times(1+1)}{12+12} = \frac{1}{2} + \frac{1}{4} = 0.75$.*

Another correct repair for P consists of two sites (x_5, x_3) and the corresponding fixes $E<5 \vee D>10 \vee D<7$ and $A=B \wedge (D\neq E \vee D>F)$. The cost for this repair is $2w + \frac{(4+3)+(5+6)}{12+12} = \frac{1}{3} + \frac{3}{4} \approx 1.08$.

A trivial single-site repair that replaces the entire P with $P^\star$ would have cost $1w + \frac{(12+12)}{12+12} \approx 1.16$.

Algorithm 1 is our overall procedure for computing a minimum-cost repair for a predicate. It considers all possible sets of repair sites, prioritizing smaller ones because the number of repair sites heavily influences the repair cost, and stopping once the lowest cost found so far is no greater than a conservative lower bound on the cost of the repairs to be considered. In the worst case, the number of repairs to be considered is exponential in the size of P, but in practice, the early stopping condition usually kicks in when the number of repair sites is 2 or 3, so the number of repairs considered is usually quadratic or cubic in $|P|$.

The two key building blocks of Algorithm 1 are CreateBounds and DeriveFixes, which we describe in more detail in the remainder of this section. Intuitively, CreateBounds (Section 5.1) provides a quick and "exact" test to determine whether a given set of repair sites could ever lead to a correct repair. If yes, DeriveFixes (Section 5.2) then finds the "optimal" fixes for these repair sites. Our algorithms use Z3, so their exactness and optimality depend on Z3's completeness for the types of predicates they are given. DeriveFixes's optimality future hinges on a Boolean minimization procedure (MinBoolExp) that it also uses. On the other hand, since Z3 inferences are sound, progress and correctness (Section 3.1) is guaranteed.

Lemma 5.1. *Qr-Hint's* WHERE*-stage hint leads to a fixed working query Q_2 with* WHERE *condition Q_2 that 1) passes the viability check $P \Leftrightarrow P^\star$; 2) satisfies $FW(Q_2) \equiv FW(Q^\star)$; and 3) leads to eventual correctness.*

Lemma 5.2. *Given P and $P^\star$, assuming that Z3 inference is complete with respect to the logic exercised by P and $P^\star$, and that MinBoolExp finds a minimum-size Boolean formula equivalent to its given input, the repair returned by RepairWhere$(P, P^\star, |P|)$ is optimal (i.e., has the lowest possible cost) if there exists an optimal repair that either contains a single site or has all its sites sharing the same parent in P.*

Proofs of the above and other lemmas are in the full version [21]. Note that Lemma 5.2 provides optimality for two important cases that commonly arise in practice: 1) P makes a single (presumably small) mistake; 2) P is either conjunctive or disjunctive (because all atomic-predicate nodes share the same $\wedge$ or $\vee$ parent node).

5.1 Viability of Repair Sites

The key idea is that, given a set of repair sites in P, we can quickly compute a "bound" that precisely defines what can be accomplished by *any* fixes at these sites (and only at these sites). We first give the definition of bounds and introduce some notations. Give quantifier-free logical formulae $P_\perp$, P, and $P_\top$ such that $P_\perp \Rightarrow P \Rightarrow P_\top$, we say that $[P_\perp, P_\top]$ is a *bound* for P, denoted $P \in [P_\perp, P_\top]$. We call $P_\top$ an *upper bound* of P and $P_\perp$ a *lower bound* of P.

CreateBounds$(P, \mathcal{S})$ (Algorithm 2) computes a precise bound for any predicate that can be obtained by fixing P at the given set $\mathcal{S}$ of repair sites. It works by computing a bound for each node in P in a bottom-up fashion, starting from the repair sites or leaves of P. We call these bounds *repair bounds*. Intuitively, the repair bound at a repair site would be [false, true], because a fix can change it to any logical formula. If a subtree contains no repair sites underneath, it would have a very tight repair bound of $[p, p]$, where p denotes the formulae corresponding to the subtree, which is unchangeable by the given repair. The internal logical nodes combine and transform these bounds in expected ways in Algorithm 2.

[6] For node x in P, $\mathcal{S}[x]$ denotes the subset of $\mathcal{S}$ that belong to the subtree rooted at x.

Algorithm 3: DeriveFixes$(x, \mathcal{S}, l^\star, u^\star)$

```
Input  : a predicate x, a set S of disjoint subtrees (repair sites) of x, and a target
         bound [l*, u*] for x to achieve by fixes
Output : a repair represented as a set of (s, f) pairs, one for each s ∈ S
1  if x ∈ S then return {(x, MinFix(l*, u*))} ;
2  else if x is atomic then return ∅ ;
3  else if op(x) is ¬ then
4      let c = Children(x)[0];                                  // the only child of x
5      return DeriveFixes(c, S[c], ¬u0*, ¬l0*);
6  let Θ = op(x) ;                                  // either ∧ or ∨ at this point
7  foreach c ∈ Children(x) do
8      let [l_c, u_c] = CreateBounds(c, S[c]);
9  let R = Children(x) ∩ S ;                        // children of x being repaired
10 if R = ∅ then let C = Children(x);
11 else // treat all children being repaired as one
12     let r = Θ_{c∈R} c and [l_r, u_r] = [false, true];
13     let C = Children(x) \ R ∪ {r};
14 let F = ∅ ;                                      // result set of (s, f) pairs to be computed
15 foreach c ∈ C do
       // Combine bounds from all other children:
16     let [l', u'] = [Θ_{c'∈C\{c}} l_{c'}, Θ_{c'∈C\{c}} u_{c'}];
17     if Θ is ∧ then
18         let l_c* = l*; let u_c* = u_c ∧ (u* ∨ ¬u');
19     else // Θ is ∨
20         let l_c* = l_c ∨ (l* ∧ ¬l'); let u_c* = u*;
21     if c is not r then let F = F ∪ DeriveFixes(c, S[c], l_c*, u_c*);
22     else let F = F ∪ DistributeFixes(MinFix(l_c*, u_c*), C);
23 return F;
```

Example 7. *Given repair sites $\{x_4, x_{10}, x_{12}\}$ for P in Figure 1, CreateBounds computes the repairs bounds as shown below.*

Node(s)	*repair lower bound*	*repair upper bound*
x_4	*false*	*true*
x_8, x_9, x_5	*same as in original predicate*	
x_2	*false*	$D{\neq}E \vee D{>}F$
x_6	*same as in original predicate*	
x_{10}	*false*	*true*
x_{11}	*same as in original predicate*	
x_{12}	*false*	*true*
x_7	$D{<}7$	*true*
x_3	$A{=}C \wedge D{<}7$	$A{=}C$
x_1 *(P)*	$A{=}C \wedge D{<}7$	$D{\neq}E \vee D{>}F \vee A{=}C$

The following shows that repair bounds computed by CreateBounds are valid. The proof uses an induction on the structure of P.

Lemma 5.3 (Validity of Repair Bounds). *Given a predicate P and a set $\mathcal{S}$ of repair sites, CreateBounds$(P, \mathcal{S})$ outputs two predicates $P_\perp$ and $P_\top$, such that applying any repair $(\mathcal{S}, \mathcal{F})$ (with the given $\mathcal{S}$) will result in a predicate $P' \in [P_\perp, P_\top]$.*

Lemma 5.3 immediately yields a method for deciding whether a candidate set $\mathcal{S}$ of repair sites is viable: if the target formula $P^\star \notin [P_\perp, P_\top]$ given $\mathcal{S}$, then there does not exist a set of correct fixes $\mathcal{F}$ for $\mathcal{S}$. The next natural question to ask is: if the target formula $P^\star \in [P_\perp, P_\top]$ given $\mathcal{S}$, is it always possible to find some correct fixes? The answer to this question is yes—and Section 5.2 will provide a constructive proof. Hence, repair bounds provide a *precise* test of whether a set $\mathcal{S}$ of repair sites is viable.

For example, continuing from Example 7, using Z3, it is easy to verify that $P^\star \in [A{=}C \wedge D{<}7,\ D{\neq}E \vee D{>}F \vee A{=}C]$; therefore, $\{x_4, x_{10}, x_{12}\}$ is a viable set of repair sites for P with respect to $P^\star$.

5.2 Derivation of Fixes

Suppose the target formula $P^\star$ falls within the repair bound $[P_\perp, P_\top]$ computed by CreateBounds$(P, \mathcal{S})$. We now introduce DeriveFixes (Algorithm 3) that computes correct fixes $\mathcal{F}$ for $\mathcal{S}$. The idea is to traverse

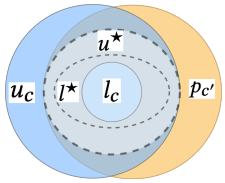


Figure 2: Relationships among $l_c, u_c, l^\star, u^\star, p_{c'}$ in a Venn diagram.

P's syntax tree top-down and derive a *target bound* for each node x. As long as we repair subtrees rooted at x's children such that the resulting predicates fall within their respective target bounds, we will have a repair for x that makes its result predicate fall within x's target bound. We start from P's root with the desire target bound $[P^\star, P^\star]$ and "push it down"; whenever we reach a repair site, its fix would simply be the smallest formula (found by MinFix) that falls within the target bound we have derived for the repair site.

The intuition behind how to "push down" the target bound at node x to its children is as follows. First, the repair bound on a child c of x dictates what repairs are possible—the target bound we set for c must be bound by its repair bound. However, we want to tighten the repair bound as little as possible because a looser target bound gives MinFix more freedom in finding a small formula.

Lines 15–22 of Algorithm 3 spells out our strategy, but we will illustrate the key ideas with some special cases (the full version [21] contains more detailed explanations). First, consider a simple case of a $\wedge$ node x with only two children c and c'. Suppose c has some repair site(s) in its subtree, but c' does not. The repair bound for c is $[l_c, u_c]$, but c' is fixed at $p_{c'}$. The relationships among these formulae are depicted in a Venn diagram in Figure 2. Given a target bound $[l^\star, u^\star]$ for x, how do we tighten $[l_c, u_c]$ a little as possible to create the target bound $[l_c^\star, u_c^\star]$ for c, such that if we pick any predicate p_c within $[l_c^\star, u_c^\star]$, $p_c \wedge p_{c'}$ is guaranteed to be in $[l^\star, u^\star]$?

- To derive $l_c^\star$, we have to expand l_c to at least $l_c \vee l^\star \Leftrightarrow l^\star$. This expansion is necessary because if p_c does not cover $l^\star$ in the Venn diagram (i.e., $p_c \not\Leftarrow l^\star$), $p_c \wedge p_{c'}$ will not either.
- To derive $u_c^\star$, as a first cut, we could tighten u_c to $u_c \wedge u^\star \Leftrightarrow u^\star$, which ensures $p_c \Rightarrow u^\star$ and therefore $p_c \wedge p_{c'} \Rightarrow u^\star$. But this is an overkill because of the help from $p_{c'}$: it is fine for p_c to include a region outside $u^\star$ in the Venn diagram, provided that it will be excluded by $p_{c'}$ when we take the conjunction. Therefore, instead of tightening to $u_c \wedge u^\star$, we can just set $u_c^\star$ to be $u_c \wedge (u^\star \vee \neg p_{c'})$.

Now, consider the more complex case where both c and c' contain repair sites; the situation is murkier. Suppose their repair bounds are $[l_c, u_c]$ and $[l_{c'}, u_{c'}]$, respectively. We can tighten the bounds for both c and c' to guarantee the overall target bound, but how we tighten one would affect the other. While making a globally optimal decision is expensive (which we will discuss later), making a correct, conservative decision is not difficult. DeriveFixes treats c and c' symmetrically: we would set $u_c^\star$ to $u_c \wedge (u^\star \vee \neg l_{c'})$ and set $u_{c'}^\star$ to $u_{c'} \wedge (u^\star \vee \neg l_c)$, essentially assuming that they receive the least amount of help possible from the other.

Example 8. *Continuing from Example 7, we show the target bounds derived by DeriveFixes for P given repair sites $\{x_4, x_{10}, x_{12}\}$ in Table 1.*

Another aspect of DeriveFixes worth mentioning is its handling of the case when multiple repair sites have the same $\wedge$ or $\vee$ parent

Node(s)	target lower bound	target upper bound
x_1 (P)	$P^\star$	$P^\star$
x_2	$P^\star \wedge \neg(A{=}C \wedge D{<}7)$	$P^\star$
x_4	$P^\star \wedge \neg(A{=}C \wedge D{<}7)$	$P^\star \vee \neg(D{\neq}E \vee D{>}F)$
x_5, x_8, x_9	same as in original predicate	
x_3	$P^\star \vee (A{=}C \wedge D{<}7)$	$P^\star$
x_6	same as in original predicate	
x_7	$P^\star \vee (A{=}C \wedge D{<}7)$	$P^\star \vee \neg(A{=}C)$
$x_{10} \wedge x_{12}$	$P^\star \wedge \neg(D{<}7)$	$P^\star \vee \neg(A{=}C)$
x_{11}	same as in original predicate	

Table 1: Target lower and upper bounds in Example 8

(which is common because many queries in practice are conjunctive, and therefore, their trees have only two levels—the root and the leaves). Since $\wedge$ and $\vee$ are commutative, all such sites can be combined into effectively one site (r in Algorithm 3) to be fixed. In Example 8 above, x_{10} and x_{12} are handled in this manner. Once we obtain a fix for r using MinFix, it can then be distributed among the constituent sites (details in [21]).

The following is the main result of this section, which affirms that so long as a candidate set $\mathcal{S}$ of repair sets passes the repair bound check in Section 5.1, there must exist a correct repair for $\mathcal{F}$ and DeriveFixes will find it. This lemma and Lemma 5.3 together imply that our repair bound check is *exact*.

Lemma 5.4 (Existence of Correct Repair). *Suppose $P^\star \in$ CreateBounds$(P, \mathcal{S})$. DeriveFixes$(P, \mathcal{S}, P^\star, P^\star)$ returns $\mathcal{F}$ such that applying $(\mathcal{S}, \mathcal{F})$ to P yields a formula equivalent to $P^\star$.*

In the remainder of this section, we first focus on MinFix, which DeriveFixes uses to find the smallest formula within a target bound. We end with a discussion of complexity, optimality, and, when we cannot guarantee optimality, techniques to mitigate suboptimality.

Finding Smallest Formula with a Bound. Given a target bound $[l^\star, u^\star]$ for a repair site, MinFix needs to find a formula g with the smallest size possible such that $g \in [l^\star, u^\star]$. This goal is intimately related to the *Boolean minimization* problem, which has been well studied and known to be hard (Σ_2^p shown by [11]). Many practically effective tools have been developed over the years, so our strategy is to leverage these tools for Qr-Hint. There are two technical challenges: 1) Boolean minimization is formulated in terms of expressions involving independent Boolean variables, while our formulae involve atomic predicates whose truth values are not independent. 2) Our minimization problem is given a bound as opposed to a single expression that Boolean minimization typically expects.

To address (1), we run a heuristic procedure using Z3 to identify a set $\mathcal{A}$ of "unique" atomic predicates that appear in $l^\star$ and $u^\star$; those that are logically equivalent to others or can be expressed easily in terms of others (e.g., with a negation) are excluded. This procedure does not need to detect or remove intricate dependencies (such that $A{>}C$ follows from $A{>}B$ and $C{\leq}B$); any such dependencies will still be caught later. Then, we map each predicate in $\mathcal{A}$ to a unique Boolean variable and convert $l^\star$ and $u^\star$ into Boolean expressions involving these variables.

To address (2), we note that many practical Boolean minimization tools accept the specification of Boolean expressions as truth tables with possible *don't-care* output entries. Our idea is to use *don't-cares* to encode the constraint implied by the target bound. Specifically, we generate a truth table whose rows correspond to truth assignments of the Boolean variables for $\mathcal{A}$. If a particular assignment is not feasible (which is testable in Z3) due to interacting atomic predicates, we mark the output for the row as *don't-care*. For each feasible assignment, if $l^\star$ and $u^\star$ evaluates to the same truth value, we designate the output for that row to be this value. If $l^\star$ evaluates to false and $u^\star$ evaluates to true, we mark the output to be *don't-care*—reflecting the flexibility offered by the bound. (Note that because $l^\star \Rightarrow u^\star$, the case where $l^\star$ and $u^\star$ evaluate to true and false respectively cannot occur.)

The current implementation of Qr-Hint uses Quine-McCluskey's method [35] as the primitive MinBoolExp for finding a minimum-size Boolean expression given a truth table with *don't-cares*. Better alternatives such as *ESPRESSO* [9] can be used instead, although our current implementation already performs well in practice.

Complexity and Optimality. In our analysis below, let κ denote the combined size of formulae P and $P^\star$. DeriveFixes's main cost comes from calls to MinFix and Z3. The number of times that MinFix is invoked is $|\mathcal{S}|$, which is $O(\kappa)$ but is usually a small constant in practice. MinFix runs in time exponential in the number of Boolean variables, which is capped at κ. To construct the input truth table for MinBoolExp, MinFix will also call Z3 $O(2^\kappa)$ times. Each Z3 call may take time exponential in the length of its input, though in practice, we time out with an inconclusive answer. Finally, as discussed at the beginning of Section 5, the number of calls to DeriveFixes by RepairWhere can be worst-case exponential in κ, but in practice it will be $O(\kappa^3)$. Regardless, the overall complexity of RepairWhere is exponential in the complexity of the WHERE predicates. Although this worst-case complexity seems daunting, we have found that Qr-Hint delivers acceptable performance in practice: thankfully, κ is often small, and the structures of P and $P^\star$ and the interdependencies among their atomic predicates tend to be much simpler than, e.g., our Example 5.

The optimality result is presented earlier as Lemma 5.2; see [21] for its proof. Intuitively, the guarantees (which still depend on the primitives Z3 and MinBoolExp) stem from two observations: 1) if repair is limited to a single site, the target bound computed by DeriveFixes is indeed the best one can do; and 2) if all sites share the same parent, DeriveFixes would effectively process them as a single site. However, as discussed earlier in this subsection, target bounds for non-combinable repair sites cannot be set optimally in an independent manner; the conservative approach taken by DeriveFixes cannot guarantee a minimum-size repair. Indeed, our running example Example 8 with repair sites $\{x_4, x_{10}, x_{12}\}$ is an instance where DeriveFixes fails to set target bounds optimally, because x_4 has a different parent from x_{10} and x_{12}. To mitigate this problem, we have developed a more sophisticated algorithm for finding fixes for multiple sites holistically. A full discussion of that algorithm is beyond the scope of this paper (see [21] for details). The improved algorithm increases the complexity by another factor of $2^{|\mathcal{S}|}$. It is heuristic in nature (as it prioritizes repair sites by how constrained they are) and cannot guarantee optimality beyond Lemma 5.2. However, it does well in practice and better than the basic version of DeriveFixes. Since $|\mathcal{S}|$ is small in practice, the complexity overhead is a good price to pay.

Example 9. *In Example 8, for repair sites $\{x_4, x_{10}, x_{12}\}$, the improved algorithm finds fixes $x_4 \mapsto A{=}B$; $x_{10} \mapsto D{>}10$; $x_{12} \mapsto E{<}5$.*

6 GROUP BY STAGE

We check the GROUP BY equivalence assuming both the reference and user $Q^\star$, Q have equivalent FROM and WHERE clauses.

In the following, we consider the case where both Q and $Q^\star$ have grouping and/or aggregation. Suppose we have unified the WHERE conditions and GROUP BY expressions in the two queries according to the table mapping $\mathfrak{m}$. Let P denote the resulting formula for $Q^\star$'s WHERE condition (which at this point is logically equivalent to Q's), and let $\vec{o}$ and $\vec{o^\star}$ denote the resulting lists of GROUP BY expressions for Q and $Q^\star$, respectively. Note that the ordering of the GROUP BY expressions is unimportant. Also, if a query involves aggregation but has no GROUP BY, we consider the list of GROUP BY expressions to be an empty list. Same column references across P, $\vec{o}$, and $\vec{o^\star}$ are treated as same variables. Our goal is to compute a subset Δ^- of GROUP BY expressions to be removed from Q, as well as a set Δ^+ of additional GROUP BY expressions to be added to Q, such that the resulting query will always produce the same grouping of intermediate result tuples (produced by FROM-WHERE) as $Q^\star$. In practice, we may not want to reveal Δ^+, but instead simply hint that Q misses some GROUP BY expressions. We may repeat the hinting process several times until GROUP BY is completely fixed.

Repairing grouping is trickier than it seems because seemingly very different GROUP BY lists can produce equivalent grouping, as illustrated by the following example.

Example 6.1. Consider two queries over tables R(A, B) and S(C, D):

```
SELECT B FROM R, S WHERE B=C GROUP BY B, D; -- Q*
SELECT C FROM R, S WHERE B=C GROUP BY C+D, C; -- Q
```

The two queries are equivalent, even though none of the pairs of GROUP BY expressions are equivalent when examined in isolation.

To address this challenge, instead of comparing pairs from $\vec{o^\star}$ and $\vec{o}$ in isolation, we holistically consider these lists as well as the WHERE condition, and go back to the definition of GROUP BY as computing a partitioning of intermediate result tuples. Formally, $\vec{o}$ and $\vec{o^\star}$ achieve the same partitioning if we can show that for any two intermediate result tuples t_1 and t_2, which are known to satisfy P, we have $\bigwedge_i (o_i[t_1] = o_i[t_2]) \Leftrightarrow \bigwedge_i (o_i^\star[t_1] = o_i^\star[t_2])$ (this is the viability check V_3). Here, we use $o[t]$ to denote evaluating e over t.[7] This approach underlines our algorithm FixGrouping (Algorithm 4).

Example 10. *Consider the two queries in Example 6.1. The table mapping is trivial and we simply use column names to name variables. We have: P is $B = C$, $\vec{o^\star} = [B, C]$, and $\vec{o} = [C + D, C]$. The logical statement that establishes the equivalence of grouping is*

$$\forall (A_1, B_1, C_1, D_1), (A_2, B_2, C_2, D_2) :$$
$$(B_1{=}C_1 \wedge B_2{=}C_2) \quad \textit{// both } (A_1, B_1, C_1, D_1) \textit{ and } (A_2, B_2, C_2, D_2) \textit{ satisfy } P$$
$$\Rightarrow \left(\begin{array}{l} (B_1{=}B_2 \wedge D_1{=}D_2) \quad \textit{// } Q^\star\textit{'s grouping criterion} \\ \Leftrightarrow (C_1{+}D_1{=}C_2{+}D_2 \wedge C_1{=}C_2) \quad \textit{// } Q\textit{'s grouping criterion} \end{array} \right).$$

Note that instead of referring to tuples t_1 and t_2, we simply refer to variables representing their column values in the above.

In FixGrouping, to find Δ^-, which are "wrong" expressions in $\vec{o}$, we check, for each o_i, whether it is possible that given $P[t_1] \wedge P[t_2]$, we can have $\bigwedge_i (o_i^\star[t_1] = o_i^\star[t_2])$ but not $o_i[t_1] = o_i[t_2]$. If yes, that means o_i is wrong with respect to $o^\star$, because while t_1 and t_2 should belong to the same group per $o^\star$, grouping by o_i alone would have forced them into separate groups instead. After identifying all wrong expressions in $\vec{o}$ and removing them, we are left with a partitioning potentially coarser than $o^\star$ but otherwise consistent with $o^\star$. We then find Δ^+ to be further added in a similar fashion.

[7] Formally, we treat t as an assignment of variables (column references) in e to variables representing corresponding column values in t. Hence, $e[t]$ is an expression obtained from e by replacing each variable (column reference) v with variable $t(v)$.

Algorithm 4: FixGrouping$(P, \vec{o}, \vec{o^\star})$

Input : a formula P and two expression lists $\vec{o}$ and $\vec{o^\star}$
Output : a pair (Δ^-, Δ^+), where $\Delta^- \subseteq [1..\dim(\vec{o})]$ is a subset of indices of $\vec{o}$ and $\Delta^+ \subseteq [1..\dim(\vec{o^\star})]$ is a subset of indices of $\vec{o^\star}$

```
let v⃗ denote the set of variables in P, o⃗, and o⃗*;
let t1, t2 be two assignments of v⃗ to new sets of variables v⃗1 and v⃗2;
let G* denote the formula ⋀_i (o_i*[t1] = o_i*[t2]);
let Δ⁻ = ∅;
foreach o_i ∈ o⃗ do
    if IsSatisfiable(P[t1] ∧ P[t2] ∧ G* ∧ o_i[t1] ≠ o_i[t2]) then
        let Δ⁻ = Δ⁻ ∪ {i};
let G denote the formula ⋀_{i∉Δ⁻} (o_i[t1] = o_i[t2]);
let Δ⁺ = ∅;
foreach o_i* ∈ o⃗* do
    if IsSatisfiable(P[t1] ∧ P[t2] ∧ G ∧ o_i*[t1] ≠ o_i*[t2]) then
        let Δ⁺ = Δ⁺ ∪ {i};
        let G = G ∧ o_i*[t1] ≠ o_i*[t2];
return (Δ⁻, Δ⁺);
```

Lemma 6.2. *We say that two lists of* GROUP BY *expressions are* equivalent *if they produce the same partitioning for the above query over any database instance. Let $(\Delta^-, \Delta^+) = \mathit{FixGrouping}(P, \vec{o}, \vec{o^\star})$. Assuming that subroutine IsSatisfiable returns no false positives, we have:*

Correctness: GROUP BY*-stage hint leads to a fixed working query Q_3 that 1) passes the viability check ($\vec{o}, \vec{o^\star}$ are equivalent), 2) satisfies $\mathit{FWG}(Q_3) \equiv \mathit{FWG}(Q^\star)$; and 3) leads to eventual correctness.*

Further assuming that IsSatisfiable returns no false negatives, we have:

Strong Minimality of Δ^-: *Let $(\Delta_\circ^-, \Delta_\circ^+)$ denote the minimal Δ^- and Δ^+ respectively, then for any $(\Delta_\circ^-, \Delta_\circ^+)$ such that $\vec{o} \setminus \Delta_\circ^- \cup \Delta_\circ^+$ is equivalent to $\vec{o^\star}$, $\Delta^- \subseteq \Delta_\circ^-$.*

Weak Minimality of Δ^+: *If $\Delta^+ \neq \emptyset$, then there exists no $\Delta_\circ^-$ such that $\vec{o} \setminus \Delta_\circ^-$ is equivalent to $\vec{o^\star}$.*

The strong minimality of Δ^- means that we can hint each expression therein as a "must-fix." The weak minimality of Δ^+ works perfectly as we simply hint that the wrong query needs some additional GROUP BY expressions.

7 HAVING STAGE

At HAVING stage, we check its equivalence assuming that $Q^\star, Q$ have equivalent FROM, WHERE, and GROUP BY, and are unified with a mapping. While HAVING can also be modeled as a logical formula, it imposes new challenges for finding a repair: 1) unlike WHERE, inputs to the formulas are arrays of tuples $[t_1, ..., t_n]$ instead of single tuples, 2) we need to consider aggregate functions, and 3) we cannot test HAVING alone without considering WHERE's effect.

Example 11. *Consider the following over* R(A, B) *and* S(C, D)*:*

```
SELECT A FROM R, S WHERE A=C AND A>4 GROUP BY A, B
HAVING A > B + 3 AND 2*SUM(A) > 10; -- Q*
SELECT A FROM R, S WHERE A=C GROUP BY A, B, C
HAVING C > B + 3 AND SUM(A * 2) > 10 AND A>4; -- Q
```

The two queries are equivalent due to $A = C$ in WHERE. Since A and C are arrays of values under the context of HAVING, they need to be rewritten as $\forall i \in \mathbb{N} : A[i] = C[i]$ when incorporating this constraint for HAVING. Furthermore, when translating HAVING into a logical formula, we need to distinguish between non-aggregate predicates (e.g. $A > B + 3$, apply to each tuple in the group) and aggregate predicates (e.g. $2 * \mathrm{SUM}(A) > 10$, apply to the entire group

of tuples), as each variable represents an array of values. To this end, we universally quantify the formula with an integer i and treat non-aggregate predicates as constraints on each tuple within the group. Then the SQL formula for $Q^\star$ in Example 11 becomes $\forall i \in \mathbb{N} : A[i] > B[i]+3 \wedge 2 * \texttt{SUM}(A) > 10$. Meanwhile, the operator $*$ is not defined between an array and an integer for $\texttt{SUM}(A * 2)$ in Q, we thus create a new array A_0 such that $\forall i, A_0[i] = A[i] * 2$, and rewrite $\texttt{SUM}(A*2) > 10$ to be $A_0[i] = A[i]*2 \wedge \texttt{SUM}(A_0) > 10$. Given these constraints, we thus create a context C (as in Example 3), under which we then test the equivalence of HAVING.

Example 12. *Let $H^\star, H$ denote the formulas for $Q^\star, Q$ respectively. Consider the formulas for $Q^\star$ and Q in Example 11.*

$$(H^\star) \qquad \forall i : (A[i]>B[i]+3) \wedge (2*\texttt{SUM}(A)>10)$$
$$(H) \qquad \forall i : (C[i]>B[i]+3) \wedge (\texttt{SUM}(A_0)>10) \wedge (A[i]>4)$$

To test equivalence, we create the following context C and predicates:

$$C : \left\{ \begin{array}{c} \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{A_0} \text{ has type } \mathit{Array}(\mathbf{N}) \\ \texttt{SUM} \text{ has type } \mathit{Array}(\mathbb{N}) \rightarrow \mathbb{N} \\ \forall i \in \mathbb{N} : \mathbf{A}[i] = \mathbf{C}[i] \wedge \mathbf{A}[i] > 4 \\ \forall i \in \mathbb{N} : \mathbf{A_0}[i] = \mathbf{A}[i] * 2 \end{array} \right\}, \qquad h_1 : H^\star \wedge \neg H, \quad h_2 : \neg H^\star \wedge H$$

In this case, HAVING *clauses of $Q^\star, Q$ are equivalent only if both $\mathit{IsUnSatisfiable}_C(h_1)$ and $\mathit{IsUnSatisfiable}_C(h_2)$ return "unsatisfiable".*

While various constraints can be encoded, the semantics of some aggregate functions cannot be captured using first-order logic (e.g. $\text{IsUnSatisfiable}_C(h_1)$). Thus, aggregate functions are declared as uninterpreted functions. However, Z3 allows encoding of function properties (e.g. linearity of SUM), enabling us to define a base context (see [21]) with some default constraints (WHERE constraints and other variables are added later):

- Declare all variables in both WHERE and HAVINGas arrays.
- Declare uninterpreted function SUM, AVG, COUNT, MAX, MIN.
- Linearity of SUM, AVG.

The viability check for HAVING (Theorem 3.1, stage 4) is that H is logically equivalent to $H^\star$ under HAVING base context C, i.e.

$$V_4 : \textit{Check if } H \Leftrightarrow H^\star \textit{ under } C$$

When two formulas are tested to be inequivalent, we then invoke the exact same procedures as for WHERE to find a repair. The only difference is that each predicate is implicitly quantified with a universal quantifier over an integer i.

Pre-processing Non-aggregate Predicates. The existence of non-aggregate predicates in HAVING might affect repair in WHERE. In Example 11, running Algorithm 1 on Q's WHERE without considering $A > 4$ in Q's HAVING might confuse users as the framework hints that such a predicate is missing. Observe that predicates that do not contain any aggregate functions might be moved into WHERE as a conjunct (e.g. $A > 4$ in Q of Example 11), we have the following preprocessing before WHERE-stage:

(1) Convert HAVING formula into conjunctive normal form (CNF).
(2) Scan each CNF clause and move only clauses that do not have any aggregate functions into WHERE formula as a conjunct. Note that only conjuncts in CNF of HAVING can be moved into WHERE.

Handling "unknown". As mentioned in Section 3, Z3 does not guarantee to return definite answers at all times. In general, there are two major scenarios where we need to handle the "unknown", and we try to be conservative for safety. (1) When "unknown" for equivalence between two formulas under a context, we treat the formulas as inequivalent and proceed to find repair. (2) When "unknown" for verifying if a formula falls within repair bounds returned by Algorithm 2, we proceed to look for other repair sites.

Lemma 7.1. *At* HAVING*-stage, given the limitation of Z3 and the conservative handling of "unknown". Qr-Hint's* HAVING*-stage hint leads to a fixed working query Q_4 that 1) passes the viability check; 2) satisfies $\mathit{FWGH}(Q_4) \equiv \mathit{FWGH}(Q^\star)$; and 3) leads to eventual correctness.*

We guarantee correctness as H can be replaced entirely with $H^\star$ in the worst case. For optimality, since we run the same algorithms as WHERE, HAVING shares the same minimality guarantee if Z3 does not return any "unknown". Otherwise, there is no guarantee on the overall minimality of the repair w.r.t. the cost function.

8 SELECT STAGE

At SELECT-stage, we check its equivalence between $Q^\star, Q$ assuming both queries have equivalent FROM, WHERE, GROUP BY and HAVING. While SELECT also contains a set of expressions, stricter restrictions are imposed: (1) the ordering of the expression is important, (2) the number of expressions must be exactly the same.

Suppose we have unified the $Q^\star, Q$ with mapping $\mathfrak{m}$. Since $Q^\star, Q$ can be either *SPJ* or *SPJA* queries, we test the equivalence between SELECT with a context designed for each type of query:

- *SPJ*: C has the WHERE formula as the only constraint.
- *SPJA*: C is inherited from HAVING-stage.

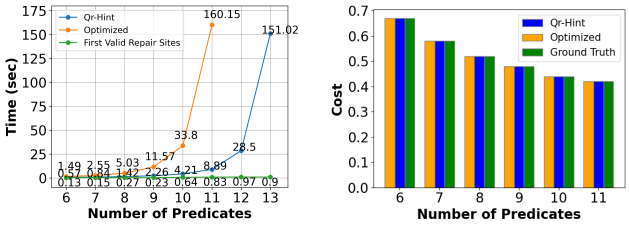
Let $\vec{o}$ and $\vec{o^\star}$ denote the resulting ordered lists of SELECT expressions for $Q, Q^\star$, respectively. For SELECT to be equivalent, we need to verify that $\dim(\vec{o}) = \dim(\vec{o^\star})$ and $\vec{o}[i]$ is equivalent to $\vec{o^\star}[i]$ for $1 \leq i \leq \dim(\vec{o^\star})$, i.e. for any tuple t in the result, $\text{IsSatisfiable}_C(o_i[t] \neq o_i^\star[t])$ returns false for $1 \leq i \leq \dim(\vec{o^\star})$. If SELECT clauses are not equivalent between $Q^\star, Q$, our goal becomes to compute Δ^- of SELECT expression to be removed from Q at the corresponding index position and a list Δ^+ of expressions to be added to Q at the corresponding index position.

The algorithm (see [21]) first goes through each pair of expressions from both $Q^\star$ and Q in order and then checks if there are any excessive expressions in either $Q^\star$ or Q. For any pair of expressions $(\vec{o}[i], \vec{o^\star}[i])$ that cannot be tested as equivalent, they are added to Δ^- and Δ^+ respectively. Finally, excessive expressions in Q or $Q^\star$ will also be added to Δ^- and Δ^+ respectively. After fixing SELECT, we guarantee $Q^\star \equiv Q$ (lemma and proof in [21]).

9 EXPERIMENTS

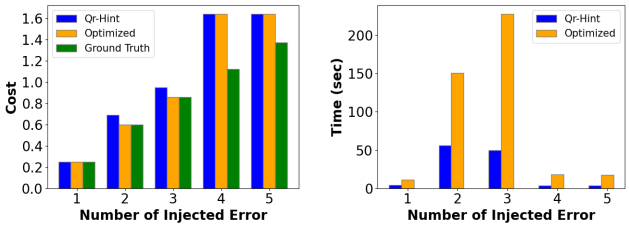
Purpose. We test two aspects of Qr-Hint: accuracy and runtime, and address the performance of Algorithm 1, which is the bottleneck of Qr-Hint. As fix minimization incurs exponential runtime, we want to know 1) how the number of unique predicates affects runtime, 2) if queries are not conjunctive, how does the generated repair compare with the ideal repair, 3) compare the runtime and cost of DeriveFixes and its optimized algorithm.

Implementation/Test Environment. We implemented Qr-Hint in Python 3.10 using Apache Calcite [5] to parse SQL queries and Z3 SMT Solver [19] to test constraint satisfiability. We run the experiments locally on a 64-bit Ubuntu 20.04 LTS server with 3.20GHz



(a) Runtime (b) Cost

Figure 3: Runtime Test for DeriveFixes and DeriveFixes-optimized



(a) Cost (b) Runtime

Figure 4: Cost and runtime with number of injected errors

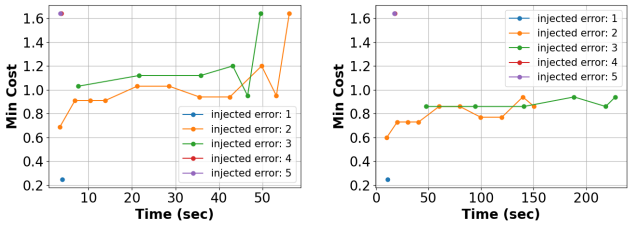
Intel Core i7-8700 CPU and 32GB 2666MHz DDR4. We tested Qr-Hint with the TPC-H [6] schema and queries to understand how Qr-Hint might perform on realistic queries.

Runtime Test. Since most WHERE in TPC-H queries are conjunctive, we pick 7 TPC-H queries that have 4-11 atomic predicates (TPC 4,3,10,9,5,8,21 respectively). We arbitrarily tweaked two predicates from each query to make up 8 query pairs. Thus, each query has 6-13 unique predicates, respectively. Per Lemma 5.2, DeriveFixes and its optimization are guaranteed to return the minimal fixes. Thus, we focus on testing how the number of predicates affects runtime.

Accuracy Test. Since Qr-Hint always returns optimal fixes on conjunctive formulas, we pick TPC-7 query to test accuracy as it has multiple nested AND/OR logical operators. To create query pairs, we tweak TPC-7 to have 1-5 injected errors by changing atomic predicates or logical operators (i.e. internal nodes). Thus, we have 5 query pairs on which we run both DeriveFixes and its optimization. In all cases, we control the number of unique predicates to always be 10 for fair comparison. While there might be more than 2 repair sites in the ideal repair, we always look for only up to 2 repair sites. Finally, we compare the runtime and cost of fixes between DeriveFixes and its optimized algorithm.

9.1 Results and Discussion

Runtime Test. The result of runtime test is shown in Figure 3. The cost comparison in Figure 3b verifies that both DeriveFixes and its optimization always return on conjunctive formulas (as all repair sites are under $\wedge$ node and are thus merged as one repair site). On the other hand, Figure 3a shows that both algorithms run in exponential time with respect to the total number of unique predicates, while DeriveFixes is faster by a small factor. Meanwhile, Figure 3a also shows the runtime to find the corresponding repair sites in green. Compared with the time for computing fixes, such time (under 1 second) is almost negligible. For the optimized algorithm, we did not show runtimes for the number of predicates larger than 11 as they exceeded our timeout (300 seconds) for the experiments.



(a) DeriveFixes (b) DeriveFixes-optimized

Figure 5: Cost of each iteration of repair sites and fixes

Accuracy Test. The results of accuracy test are shown in Figure 4 and Figure 5. As shown in Figure 4a, the costs returned by DeriveFixes and its optimization are equal to that of ideal repair (ground truth) when there is only 1 error, and this is consistent with Lemma 5.2. In other cases, DeriveFixes usually has a higher cost (e.g. when there are 2 and 3 errors). When there are 4 and 5 errors, both algorithms have the same cost because the total number of errors has become too large to be covered by only 2 repair sites, and both algorithms decide to replace the entire wrong query with the reference query. Though DeriveFixes incurs a higher cost, it runs faster than its optimization as the optimization focuses more on minimizing the size of fixes, shown in Figure 4b. Figure 5 shows how the overall minimum cost changes over time. When there is only one repair site (the single blue dot) or too many repair sites (the red and purple dots), both algorithms stop after the first set of repair sites is found, as explained previously. When the errors can be covered by 2 repair sites (orange and green lines), Figure 5 shows that the repair sites that incur the lowest cost surface very early in the process (within 10s for DeriveFixes and within 50s for its optimization). This observation implies that Qr-Hint can provide hints to users in a timely manner.

Feasibility. While the runtime of Qr-Hint grows exponentially w.r.t. the number of predicates, most real queries do not contain many predicates. TPC-H contains 22 queries; each has 6-7 predicates on average, and the most predicates a query has is 15. In a classroom setting, students should not receive the fixes that give away answers. Instead, it is better to point out the repair sites for students to think and practice debugging.

10 USER STUDY

The user study serves as a validation before deployment. We conducted a small-scale user study to evaluate Qr-Hint: (1) whether students can understand what is wrong with the proposed hints; (2) how the hints generated by Qr-Hint are comparable to the ones provided by "*expert users*" (teaching assistants in our study).

Query Error Analysis. We first analyzed 341 students' submitted wrong queries from four SQL questions (covered by Qr-Hint) from a past intro-level database course. The statistics are shown in [21]. In summary, our statistics echo the findings of [2]. Most errors come from the WHERE and HAVING (130 out of 341 are wrong due to WHERE), and students usually miss join conditions, especially when queries involve more tables. Other common mistakes include wrong/redundant/missing tables in FROM (35 out of 341) and wrong order/missing/redundant expressions in SELECT (39 out of 341). The four questions covered by Qr-Hint do not focus on GROUP BY (only one question requires it), but it is also a challenge based on [2]. Of 341 wrong queries, Qr-Hint can correct 306 (89%).

Preparation. According to the error analysis, we designed four SQL questions Q_1, Q_2, Q_3, Q_4 using the DBLP database schema (details in full version [21]). 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, one error in `WHERE` of Q_3, and one each `WHERE` and `HAVING` errors in Q_4. 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, to prevent participants from recognizing the generated hints, whether from experts or QR-HINT by observing the hint format, we paraphrased all hints as "In [SQL Clause], [Hints]".

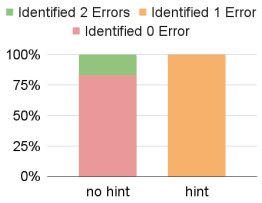
Generation of hints by human experts. First, a smaller study was performed with four graduate teaching assistants (TAs) to generate hints given by expert users. For each wrong query, each TA was asked to pinpoint and list all mistakes in a query as if they were giving hints to help students debug wrong queries. To simulate an office hour setting, we asked TAs to finish all four questions in one sitting. We collected all hints provided by the TAs.

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

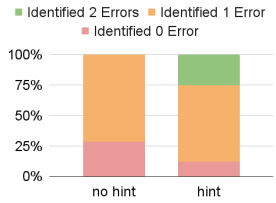
Participants. We recruited 15 students who have taken/are taking a graduate or undergraduate database course[8]. All participants have finished training for writing and debugging SQL queries.

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*", "*Helpful but require thinking*", and "*Obvious and giving away the answer*". Participants were asked to finish all questions in one sitting. We recorded the time a participant spent on each question[9]. In our study, for Q1, 8 students answered it with no hints and 7 with hints from QR-HINT. For Q2, these numbers are 7 and 8 respectively. For the third question, 7 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 6a and Figure 6b, 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 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 doing so, while most participants spent no more than 10 minutes on the same question without hints.

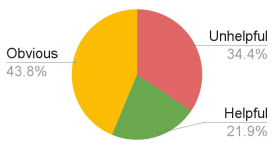


(a) Q1: No Hints vs. QR-HINT

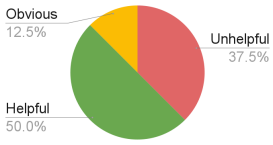


(b) Q2: No Hints vs. QR-HINT

Figure 6: User performance with/without QR-HINT

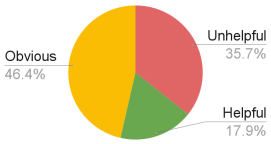


(a) Q3: Vote Cast for TA Hints

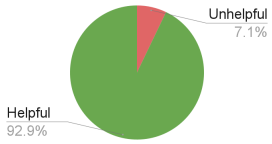


(b) Q3: Vote Cast for QR-HINT

Figure 7: Hint categorization from participants for Q3



(a) Q4: Vote Cast for TA Hints



(b) Q4: Vote Cast for QR-HINT

Figure 8: Hint categorization from participants for Q4

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 participants vote for each of the three categories of hint ranks: "Obvious", "Unhepful", and "Helpful". The results are shown in Figures 7a, 7b, 8a and 8b. In summary, the quality of TAs' hints varies greatly as perceived by participants. On the other hand, hints generated by QR-HINT are consistently perceived by participants as "helpful but require thinking", which might be best suited for classroom settings.

11 CONCLUSION AND FUTURE WORK

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 fix all clauses in a query 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, outer-joins, 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 important future work.

[8]Except an incentive of receiving a small gift card, the participation of students was voluntary, and we were able to get 15 complete responses.

[9]Q_1 without/with hints took 704s/460s on average; Q_2 took 756s/658s. Students completed the survey asynchronously, so the time recorded may not be accurate.

REFERENCES

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

[2] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. 2016. Students' semantic mistakes in writing seven different types of SQL queries. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 272–277.

[3] 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.

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

[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] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. https://proceedings.mlr.press/v139/berabi21a.html

[8] 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.

[9] Robert K Brayton, Gary D Hachtel, Lane A Hemachandra, A Richard Newton, and Alberto Luigi M Sangiovanni-Vincentelli. 1982. A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization. In *Proceedings of the International Symposium on Circuits and Systems*. 42–48.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[11] David Buchfuhrer and Christopher Umans. 2011. The complexity of Boolean formula minimization. *J. Comput. Syst. Sci.* 77, 1 (2011), 142–153. https://doi.org/10.1016/j.jcss.2010.06.011

[12] 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.

[13] 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.

[14] 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.

[15] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[16] 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.

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

[18] 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.

[19] 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.

[20] Curtis Fenner. 2019. https://cs.stackexchange.com/questions/110674/is-query-equivalence-decidable.

[21] Full version of this submission. [n.d.]. https://anonymous.4open.science/r/qr-hint-2A7C/. ([n. d.]).

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

[23] 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.

[24] 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.

[25] 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.

[26] 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).

[27] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th international conference on software engineering*. 12–23.

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

[29] 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.

[30] 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.

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

[32] Jarosław Kwiecień, Jerzy Marcinkowski, and Piotr Ostropolski-Nalewaja. 2022. Determinacy of Real Conjunctive Queries. The Boolean Case. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. Association for Computing Machinery, 347–358. https://doi.org/10.1145/3517804.3524168

[33] 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.

[34] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

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

[36] 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.

[37] Daniel Perelman, Sumit Gulwani, and Dan Grossman. 2014. Test-driven synthesis for automated feedback for introductory computer science assignments. *Proceedings of Data Mining for Educational Assessment and Feedback (ASSESS 2014)* (2014).

[38] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning*. PMLR, 1093–1102.

[39] 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.

[40] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27 (2017), 37–64.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*. 789–799.

[47] 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.

A FROM STAGE SUPPLEMENT

A.1 Finding Table Mapping

In this section, We describe our heuristics for determining a table mapping. Since looking at FROM alone does not have enough information for determining a mapping, we gather information from other clauses (i.e., WHERE, GROUP BY, SELECT) to help us decide. The general idea is to create a "table signature" for each table involved in self-join, build a bipartite graph where weights of edges represent the difference between two table signatures, and select a mapping by solving the minimum-cost bipartite matching problem.

We first describe our heuristics for creating a table signature for each table in FROM of a query Q:

(1) Scanning Q's WHERE and HAVING, for each selected operator ($=, <, >, \leq, \geq$) and each attribute a in the table, we create a set of attributes that "interact" with a in some atomic predicates in WHERE and/or HAVING(note: we rewrite the predicates to make sure a is on the left-hand side of the operator in the case of inequality). If a does not appear in WHERE or HAVING or it does not appear in a predicate with the selected operator, the corresponding set will be empty. After creating each set, we expand it to the entire equivalence class of its current attributes. We then replace each attribute in the set with the name of the original table they belong to.
(2) Scanning Q's GROUP BY, create a set of attributes from the table that appear in any GROUP BY expression.
(3) Scanning Q's SELECT, for each attribute a in the table, create a set of indices such that this attribute appears in the SELECT expression at the indexed position.

In summary, let t denote a table in a query Q, a table signature is a 3-tuple $\sigma = (W, G, S)$, where W is a function $W(a, o) \mapsto T, a \in \text{Attributes}(t), o \in =, <, >, \leq, \geq$ (T is a set of tables), G is a set such that $G \subseteq \text{Attributes}(t)$, and S is a function $S(a) \mapsto I, a \in \text{Attributes}(t)$ (I is a set of integer indexes).

With table signatures, let $O = \{=, <, >, \leq, \geq\}$ denote the set of operators, we then define the following metric for calculating a normalized distance between two signatures $\sigma = (W, G, S), \sigma' = (W', G, S')$:

$$\text{Cost}(\sigma, \sigma') = \frac{\sum_{a \in \text{Attributes}(t), o \in O} \text{dist}(W(a, o), W'(a, o))}{|\text{Attributes}(t)| \times |O|} + \text{dist}(G, G') + \frac{\sum_{a \in \text{Attributes}(t)} \text{dist}(S(a), S'(a))}{|\text{Attributes}(t)|}$$

Here we define dist as the Jaccard similarity between two sets. Each component is a normalized Jaccard similarity between the corresponding sets, and we take the sum of three Jaccard similarities (i.e. for WHERE, GROUP BY, SELECT respectively) as our final distance metric. Note that when two sets are empty, we count their Jaccard similarity as 1.

With such a distance metric, we then build a bipartite graph where

- Each node in partition 1 represents a table in $Q^\star$, and each node in partition 2 represents a table in Q.
- Each node in partition 1 is connected with at least one counterpart node that refers to the same table in partition 2. The weight of the edge between two nodes is the absolute difference between their signatures.

Once we have built the bipartite graph, we can use a linear program to solve the minimum-weight perfect matching problem. We now use an example to demonstrate the heuristic.

Example 13. *Continuing with Example 1, the initial signatures for S1, S2, s1, s2 are shown in Example 4. After replacing attribute names with table names, the final signatures are the following:*

		$S1$ in $Q^\star$	$S2$ in $Q^\star$	$s1$ in Q	$s2$ in Q
WHERE & HAVING	*bar:*	$=\{Frequents\}$	$=\{Frequents\}$	*None*	*None*
	beer:	$=\{Likes, Serves\}$	$=\{Likes, Serves\}$	$=\{Likes, Serves\}$	$=\{Likes, Serves\}$
	price:	$\leq\{Serves\}$	$\geq\{Serves\}$	$>\{Serves\}$	$<\{Serves\}$
GROUP BY		$\{bar, beer\}$	$\{beer\}$	$\{beer\}$	$\{beer\}$
SELECT	*bar:*	$\{2\}$	$\emptyset$	$\emptyset$	$\{2\}$
	beer:	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$
	price:	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

The weight of the edge between S1 and s1 is calculated as followed: $\frac{4+5+3}{15} + \frac{1}{2} + \frac{0+1+1}{3} \approx 1.97$. *Comparing the* WHERE *and* HAVING *in S1 and s1's signatures, the Jaccard distance between* $W(bar, =)$ *and* $W'(bar, =)$ *is 0 as they do not have common element. Since the other 4 operators do not involve bar, their corresponding sets are all empty and thus yield a Jaccard distance of 1 between S1 and s1. Thus the total Jaccard distance for bar is 4. Similarly, we obtain 5, 3, respectively for beer and price, and the sum of these Jaccard distances is normalized. The Jaccard distance between the* GROUP BY *is simply* $\frac{1}{2}$ *as there is only one common element. For* SELECT, *beer and price have the same sets between S1 and s1, so the normalized Jaccard distance is* $\frac{2}{3}$.

Following the same fashion, the weight of the rest of edges are below:

- $S1 \mapsto s2: \frac{4+5+3}{15} + \frac{1}{2} + \frac{1+1+1}{3} = 2.3$
- $S2 \mapsto s1: \frac{4+5+3}{15} + 1 + \frac{1+1+1}{3} = 2.8$
- $S2 \mapsto s2: \frac{4+5+3}{15} + 1 + \frac{0+1+1}{3} \approx 2.47$

With the signatures, the corresponding bipartite graph is shown below:

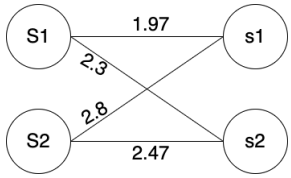


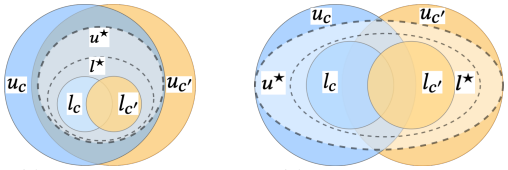
Figure 9: Constructed bipartite graph for Example 13

By negating the weight of each edge, we then convert the problem into solving the minimum-weight perfect matching on the graph, Qr-Hint *eventually choose the following mapping:* $S1 \mapsto s2, S2 \mapsto s1$, *which has the minimum overall weight* $(-2.3 + -2.8 = -5.1)$ *among all matchings, and it is the best one for suggesting fixes in the downstream stages.*

A.2 Proof for Lemma 4.2

Proof of Lemma 4.2. Without loss of generality, suppose that $Q^\star$ returns a non-empty result over some database instance D. We construct a new database instance D' as follows:

(1) For each unique table $T \in \text{Tables}(Q^\star)$, 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 $T_1 \neq T_2$).



(a) The bounds for $\wedge$ **(b) The bounds for $\vee$**

Figure 10: Venn Diagrams for visualizing relationship among formulae in Algorithm 3

(2) For each table $T \notin \text{Tables}(Q^\star)$, make T empty.

We have

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

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

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

It then follows from the Prime Factorization Theorem [22] that $\text{Tables}(Q)$ and $\text{Tables}(Q)$ contain the same *set* of tables, and that for each unique $T \in \text{Tables}(Q)$ (or $\text{Tables}(Q)$), $|\text{Aliases}(Q,T)| = |\text{Aliases}(Q^\star,T)|$. In other words, $\text{Tables}(Q) \stackrel{\square}{=} \text{Tables}(Q^\star)$. □

B WHERE STAGE SUPPLEMENT

In this section, we give a complete story for the derivation of fixes (i.e. Algorithm 3) and its optimization, as well as give proof to Lemma 5.1, Lemma 5.2, Lemma 5.3 and Lemma 5.4.

B.1 DeriveFixes Revisited

B.1.1 Target Bound Derivation. We first completely map out the story for lines 15–22 of Algorithm 3, which contain the strategy for pushing down target bound.

For illustration and without loss of generality, we again assume that all $\wedge, \vee$ nodes have only two children c, c'. While addressing a particular node x, we use the same notation as in Section 5, where $[l_c, u_c], [l_{c'}, u_{c'}]$ represent the repair bound of the child c, c' respectively, and $[l^\star, u^\star]$ denotes the x's target bound, and $[l_c^\star, u_c^\star], [l_{c'}^\star, u_{c'}^\star]$ denote the target bound of c, c' respectively.

The principle that guides the formulation of the target bound is simple: we want to recursively expand the range of the target bound as we traverse down to each repair site so that their target bounds will potentially contain small fixes. We use an example to demonstrate this principle.

Example 14. *Let $l \equiv (p_1 \vee p_2) \wedge p_3$ and $u \equiv p_1 \vee p_2$ be the lower and upper bounds, where p_1, p_2, p_3 are arbitrary predicates. The minimal formula within $[l, u]$ is $p_1 \vee p_2$. However, let $u' \equiv p_1 \vee p_2 \vee p_3$ be the new upper bound, the minimal formula that falls within $[l, u']$ becomes p_3. p_3 is smaller than $p_1 \vee p_2$ in terms of the size of their syntax trees. Here we expand the range of $[l, u]$ by relaxing the upper bound (i.e. adding a disjunct to the upper bound).*

Symmetrically, we can create a case when range expansion is done by restricting the lower bound (i.e. adding a conjunct to the lower bound). Let $l \equiv p_1 \wedge p_2$ and $u \equiv (p_1 \wedge p_2) \vee p_3$ be the lower and upper bounds. If we restrict l further by constructing $l' \equiv p_1 \wedge p_2 \wedge p_3$, the minimal formula moves from $p_1 \wedge p_2$ to p_3 for bounds $[l, u]$ and $[l', u]$ respectively.

With such a principle, the next question becomes how to expand the range of target bounds $[l^\star, u^\star]$ in the context of Algorithm 3. Situations differ based on the logical operator at each node.

When x is rooted at $\wedge$, the formula has the form $c \wedge c'$. Assuming both c, c' contain some repair sites, we want to have $p_c, p_{c'}$ satisfying the following constraints after fixes are applied:

(1) $p_c \wedge p_{c'} \in [l^\star, u^\star]$, i.e., combining $p_c, p_{c'}$ using $\wedge$ forms a new formula that falls within the target bounds at x.
(2) $p_c \in [l_c, u_c]$ and $p_{c'} \in [l_{c'}, u_{c'}]$.
(3) The target bounds to be pushed down to c, c' are contained in their repair bounds, i.e., (i) $[l_c^\star u_c^\star] \in [l_c, u_c]$, and (ii) $[l_{c'}^\star, u_{c'}^\star] \in [l_c, u_{c'}]$.

Given the above constraints, the relationship among target bounds and repair bounds are depicted in Figure 10a: $l_c \wedge l_{c'} \Rightarrow l^\star \Rightarrow u^\star \Rightarrow u_c \wedge u_{c'}$.

We now make the following observation that helps us determine how we expand the range of $[l^\star, u^\star]$ to form $[l_c^\star, u_c^\star]$ and $[l_{c'}^\star, u_{c'}^\star]$.

- $p_1 \wedge p_2 \Leftrightarrow (p_1 \vee \neg p_2) \wedge p_2 \equiv p_1 \wedge (p_2 \vee \neg p_1)$

Such observation indicates that when pushing new target bounds to c, we can expand the current target bound by excluding the semantics of c' (and vice versa), and such expansion can be done by relaxing the upper bound (i.e. $u^\star$) since we are adding a disjunct. However, given the repair bound of c', which formula within $[l_{c'}, u_{c'}]$ should we pick to ensure $p_c, p_{c'}$ satisfy the above constraints? To answer this question, we make another observation as follows:

- $p_1 \wedge p_2 \Leftrightarrow (p_1 \vee \neg p_3) \wedge p_2$, if $p_2 \Rightarrow p_3$.
- $p_1 \wedge p_2 \Leftrightarrow p_1 \wedge (p_2 \vee \neg p_3)$, if $p_1 \Rightarrow p_3$.

This implies that we can guarantee the formula at the current $\wedge$ node falls within $[l^\star, u^\star]$ as long as we relax $u^\star$ with a formula that is implied by all possible formulas within the repair bounds (because at this time, we do not know what formula p_c and $p_{c'}$ will eventually be, so we need to make a safe assumption that they could be any formula within the repair bound), and it is clear that only the repair upper bounds (i.e. $u_c, u_{c'}$) satisfy such constraints. Thus, we obtain $u_c^\star = u^\star \vee \neg u_{c'}$ and $u_{c'}^\star = u^\star \vee \neg u_c$ as the new target upper bounds for c and c' respectively.

However, at this point, both new target upper bounds fail to satisfy the second constraint outlined earlier (i.e. fall out of the repair bounds) as $u^\star \vee \neg u_{c'} \not\Rightarrow u_c$ and $u^\star \vee \neg u_c \not\Rightarrow u_{c'}$. Consequently, we have to add u_c and $u_{c'}$ as a conjunct to restrict $u_c^\star$ and $u_{c'}^\star$ respectively so that they stay in the corresponding repair bounds. Now all constraints are satisfied, and we do not change the lower bound based on the observation that relaxing lower bounds and upper bounds simultaneously does not necessarily expand the range.

When x is rooted at $\vee$, the formula has the form $c \vee c'$. Similar to an $\wedge$ node, assuming both c, c' contain repair sites, we want to have $p_c, p_{c'}$ satisfying the following constraints:

(1) $p_c \vee p_{c'} \in [l^\star, u^\star]$, i.e., combining $p_c, p_{c'}$ (after fixes are applied) falls within the target bounds at the root $\vee$ node.
(2) $p_c \in [l_c, u_c]$ and $t_1 \in [l_{c'}, u_{c'}]$.
(3) The target bounds to be pushed down to $p_c, p_{c'}$ are contained in their repair bounds, i.e., (i) $[l_c^\star, u_c^\star] \in [l_c, u_c]$, and (ii) $[l_{c'}^\star, u_{c'}^\star] \in [l_{c'}, u_{c'}]$.

Algorithm 6: MinFix($l^\star, u^\star$)

Input : predicates $l^\star$ and $u^\star$ together defining a target bound $[l^\star, u^\star]$
Output : a predicate bounded by $[l^\star, u^\star]$ that is as simple as possible

1 **let** $\vec{a}, \vec{\mathrm{a}}, \phi$ = MapAtomPreds($\{l^\star, u^\star\}$);
2 **let** $g_l = \phi(l^\star)$ and $g_u = \phi(u^\star)$; // both are Boolean functions of $\vec{\mathrm{a}}$
3 **let** $g^\star$ = BuildTruthTable($\vec{a}, \vec{\mathrm{a}}, g_l, g_u$);
4 **let** g = MinBoolExp($g^\star$);
5 **return** predicate obtained from g by replacing each variable a_i with atomic predicate a_i;

SUBROUTINE BuildTruthTable($\vec{a}, \vec{\mathrm{a}}, g_l, g_u$)
Output : a partial Boolean function $g^\star$ of $\vec{\mathrm{a}}$, consistent with the bounds defined by g_l and g_u, represented as a mapping $\{0,1\}^{\dim(\vec{\mathrm{a}})} \to \{*, 0, 1\}$

1 **let** $g^\star$ = empty mapping;
2 **foreach** *assignment* $\vec{v} \in \{0,1\}^{\dim(\vec{\mathrm{a}})}$ *of* $\vec{\mathrm{a}}$ **do**
3 **let** $x = \bigwedge_{i \in [1..\dim(\vec{a})]} x_i$, where x_i is a_i if $\vec{v}$ assigns a_i to 1, or $\neg a_i$ if $\vec{v}$ assigns a_i to 0;
4 **if** *IsUnSatisfiable*(x) **then** // input setting not possible
5 **let** $g^\star(\vec{v}) = *$;
6 **else if** $g_l(\vec{v}) = g_u(\vec{v})$ **then** // both true or both false
7 **let** $g^\star(\vec{v}) = g_l(\vec{v})$;
8 **else** // $g_l(\vec{v}) = 0$ and $g_u(\vec{v}) = 1$, because $l \Rightarrow u$
9 **let** $g^\star(\vec{v}) = *$;
10 **return** $g^\star$;

Algorithm 5: MapAtomPreds($\mathcal{P}$)

Input : a set $\mathcal{P}$ of predicates
Output : a list of atomic predicates $\vec{a} = [a_1, a_2, \ldots, a_{\dim(\vec{a})}]$ denoted by Boolean variables $\vec{\mathrm{a}} = [\mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_{\dim(\vec{\mathrm{a}})}]$, and a mapping ϕ such that for any Boolean subexpression s in any predicate in $\mathcal{P}$, $\phi(s)$ returns a Boolean function (with variables in $\vec{\mathrm{a}}$) that is equivalent to s after replacing each variable a_i with predicate a_i

1 **let** $\vec{a} = [\,]$, $\vec{\mathrm{a}} = [\,]$, and ϕ = empty mapping;
2 **foreach** *predicate in* $\mathcal{P}$ *and each atomic predicate* t *therein* **do**
3 **foreach** $a_i \in \vec{a}$ **do** // see if t is expressible by a chosen predicate
4 **if** *IsEquiv*(t, a_i) **then**
5 **let** $\phi(t) = \mathrm{a}_i$; **break**;
6 **else if** *IsEquiv*($t, \neg a_i$) **then**
7 **let** $\phi(t) = \neg\mathrm{a}_i$; **break**;
8 **if** $\phi(t)$ *is undefined* **then** // a "new" atomic predicate found
9 $\vec{a}$.append(t); $\vec{\mathrm{a}}$.append($\mathrm{a}_{\dim(\vec{a})}$); **let** $\phi(t) = \mathrm{a}_{\dim(\vec{\mathrm{a}})}$;
10 **foreach** *predicate in* $\mathcal{P}$ *and each Boolean subexpression* s *therein* **do**
11 **if** $\phi(s)$ *is undefined* **then**
12 **let** $\phi(s)$ be the Boolean function obtained from s by replacing each atomic predicate t with $\phi(t)$;
13 **return** $\vec{a}, \vec{\mathrm{a}}, \phi$;

With symmetric observations and reasoning, instead of relaxing the upper bound as for $\wedge$ node, here we further restrict the current target lower bound (i.e. $l^\star$) based on the following observation:

- $p_1 \vee p_2 \Leftrightarrow (p_1 \wedge \neg p_3) \vee p_2$, if $p_3 \Rightarrow p_2$.
- $p_1 \vee p_2 \Leftrightarrow p_1 \vee (p_2 \wedge \neg p_3)$, if $p_3 \Rightarrow p_1$.

We can guarantee the formula at the current $\vee$ node falls within $[l^\star, u^\star]$ as long as we restrict $l^\star$ with a formula that implies all possible formulas within the repair bounds, and it is clear that only the repair lower bounds (i.e. $l_c, l_{c'}$) satisfy such constraints. Furthermore, to keep the new target lower bounds within the repair bounds, we have to add l_c and $l_{c'}$ as disjunct to $l_c^\star$ and $l_{c'}^\star$ respectively, thus forming the final target lower bounds for c and c'. Note that we do not change the upper bounds based on the observation, as restricting upper bounds and lower bounds simultaneously does not necessarily expand the range.

When x is rooted at $\neg$, it has only one child, and we thus push down the target bounds by setting them to be $[\neg u^\star, \neg l^\star]$ as negation inverts the direction of implication.

B.1.2 Fix Minimization. At each repair site, Algorithm 3 calls MinFix to compute the minimal fix given the target bound. The pseudocode of MinFix is shown in Algorithm 6. It takes a lower bound $l^\star$ and an upper bound $u^\star$ as inputs, builds a desired truth table, and utilizes the Quine-McCluskey's method [35] to find a formula t in disjunctive normal form, having the minimum number of minterms among all formulas that fall within the bounds.

However, since the truth table and Quine-McCluskey's method only work with Boolean variables instead of atomic predicates, MinFix leverages a subroutine MapAtomPreds (Algorithm 5, line 1 in Algorithm 6) to

(1) scan through both $l^\star$ and $u^\star$ and extracts all semantically unique atomic predicates.
(2) determine a mapping that maps each semantically unique atomic predicate in both $l^\star$, $u^\star$ to a set of unique Boolean variables that represent the atomic predicates (e.g. $a = b$ is semantically equivalent to $a + 1 = b + 1$, thus they will be mapped to the same Boolean variable).
(3) construct a mapping ϕ which maps any subexpression in a predicate (formula) to a Boolean function so that $\phi(l^\star)$ and $\phi(u^\star)$ return the Boolean functions that represent the truth table of $l^\star$ and $u^\star$ respectively.

Given the Boolean functions for $l^\star$ and $u^\star$ (line 2), MinFix then calls a subroutine BuildTruthTable to construct the Boolean function representing the truth table of the minimal formula $t \in [l^\star, u^\star]$ (line 3 in MinFix) by going through all possible truth value assignments for $l^\star$ and $u^\star$ (line 2 in BuildTruthTable) with the following criteria:

- If the conjunction of all atomic predicates in a row is not satisfiable (e.g. $a = b$ and $a > b$ cannot both be true simultaneously), we mark the truth value for t by "$*$" (don't-care) since such situation can never happen (line 4-5 in BuildTruthTable).
- If the conjunction of all atomic predicates in a row is satisfiable (line 6-9 in BuildTruthTable):
  (1) if $l^\star$ and $u^\star$ are evaluated to the same truth value (i.e. both true or both false), then the same truth value will also be assigned to t.
  (2) if $l^\star$ and $u^\star$ are evaluated to different truth values (i.e. false for $l^\star$ and true for $u^\star$), then a don't-care is assigned to t. The purpose of such a don't-care assignment is to allow more flexibility for Quine-McCluskey's method to minimize t as much as possible. Note that the situation where $l^\star$ is evaluated to true and $u^\star$ is evaluated to false can never happen due to $l^\star \Rightarrow u^\star$.

After obtaining the Boolean function (i.e. truth table) for t, MinFix feeds it to a subroutine MinBoolExp which minimizes a given Boolean function f with possible "don't-care outputs" (denoted $*$). Boolean minimization in general is NP-complete, but good heuristics exist that often find near-optimal solutions for even a large number of variables. Our implementation uses the standard Quine-McCluskey algorithm [35], but better alternatives such as *ESPRESSO* [9] can also be used. We demonstrate how MinFix works using the following example.

$a \geq b$	$f = e$	$a = b$	$a > b$	$l^\star$	$u^\star$	t
0	0	0	0	0	0	0
0	0	0	1	*	*	*
0	0	1	0	*	*	*
0	0	1	1	*	*	*
0	1	0	0	0	1	*
0	1	0	1	0	1	*
0	1	1	0	*	*	*
0	1	1	1	*	*	*
1	0	0	0	*	*	*
1	0	0	1	0	1	*
1	0	1	0	1	1	1
1	0	1	1	*	*	*
1	1	0	0	*	*	*
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	*	*	*

Figure 11: Compact Truth tables for $l^\star$, $u^\star$ and t in Example 15.

Example 15. *Consider the following bounds for a minimal formula t: $l^\star \equiv (a \geq b \wedge f = e) \vee a = b$, and $u^\star \equiv a = b \vee e = f \vee a > b$. 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 t yields $t \equiv a \geq b$, and $t \in [l^\star, u^\star]$.

Given MinFix, the fixes computed for the repair sites in Example 8 are the following:

- $x_4 : a = b \vee (a = c \wedge d > 10) \vee (a = c \wedge d < 7)$
- $x_{10}, x_{12} : (N_{10}, N_{12}) : (a = b \wedge d \neq e) \vee (a = b \wedge d > f) \vee (a = c \wedge d > 10) \vee (a = c \wedge e < 5)$

While the sizes of these fixes are still quite large, it is verifiable that the resulting formula is equivalent to $P^\star$ in Example 5 after inserting the fixes into P.

B.2 Optimization for Algorithm 3

The suboptimality of multiple fixes comes from the independent derivation of the target formula bounds (Algorithm 3). Using the same notations as in Algorithm 3, consider the target bounds for the children of an $\vee$ node with only two children and their Venn Diagram in Figure 10b:

$$[l_c^\star, u_c^\star] = [l_c \vee (l^\star \wedge \neg l_{c'}), u^\star],\ [l_{c'}^\star, u_{c'}^\star] = [l_{c'} \vee (l^\star \wedge \neg l_c), u^\star]$$

Here we use the Venn diagram to illustrate the source of suboptimality. The regions representing $l_c^\star$ and $l_{c'}^\star$ overlap (same for $u_c^\star, u_{c'}^\star$). Though their union matches exactly the region of $l^\star$ ($u^\star$ respectively), such overlap indicates an overlap in their semantics, meaning semantic redundancy exists in the target bounds of Children(x). This implies that any combination of $p_c \in [l_c^\star, u_c^\star]$ and $p_{c'} \in [l_{c'}^\star, u_{c'}^\star]$ have semantic overlaps, causing suboptimality in the subsequent derivation of fixes. To illustrate this, reconsider Example 8. The predicates $a = c$, $d > 10$, and $d < 7$ appear in the fix for x_4 unnecessarily.

To reduce such suboptimality, we propose to use a new procedure **instead of Algorithm 3** to consider all repair sites simultaneously by leveraging the fact that Algorithm 3 returns optimal fix for a single repair site (Lemma 5.2). The overall routine is shown in Algorithm 7.

The general intuition of Algorithm 7 is to start with $l^\star$ and $u^\star$ being the reference formula and updates them in the same manner as Algorithm 3 until the lowest common ancestor (LCA) of all repair sites. Treating the LCA as a single repair site, the corresponding optimal fix lies within its $[l^\star, u^\star]$, and Algorithm 7 aims to make up such optimal fix by collectively deriving fixes for all actual repair sites. We next describe the major steps of Algorithm 7. For a concise and clear illustration, we denote the optimal single fix at the LCA with T.

Build consistency table. We first replace each repair site with a unique Boolean variable, forming a new Boolean predicate P' at the LCA, and the goal is to make $P' \equiv T$. Because two Boolean formulas are equivalent if they share the same truth table, we then want to observe under what value assignments T and P' are consistent (i.e., evaluated to the same truth value) before determining how to construct each individual fix. Therefore, Algorithm 7 first generates a "**consistency table**" (line 4) where the inputs are all unique Boolean variables and atomic predicates from T, P', and the formulas being evaluated are $l^\star, u^\star, T, P'$, here $l^\star, u^\star$ are present for the derivation of T which follows the same procedures as in MinFix (Algorithm 6).

Example 16. *Consider the following formulae: $P^\star \equiv a = 1 \vee (b = 2 \wedge c = 3)$, $P \equiv c = 3 \vee (b = 2 \wedge a = 1)$.*

Let the repair sites in P be $c = 3$ and $a = 1$ with Boolean variables r_1, r_2, and their lowest common ancestor be the top-level $\vee$. We can obtain $l^\star \Leftrightarrow u^\star \Leftrightarrow P^\star$. Therefore, the consistency table can be constructed as shown in Figure 12.

The consistency table gives a direct view of the occasions where T and P' are consistent (e.g. blue highlights, evaluated to the same truth value) and inconsistent (e.g. red highlights, evaluated to different truth values). This helps us determine how to construct each fix in later steps as we want to avoid any inconsistency between T and P' to achieve $T \Leftrightarrow P'$.

Build constraint table. After acquiring the consistency table, the next question becomes how to use all available atomic predicates to construct fixes for each repair site while avoiding inconsistencies. For this purpose, we turn all atomic predicates as "constraints" for all Boolean variables that represent the repair sites, thus constructing a "**constraint table**" (line 7). A constraint table is a truth table where inputs are all the atomic predicates in the consistency table, and the output is the concatenation of truth values of Boolean variables. For each row (i.e. truth assignment of all atomic predicates), the constraint table aggregates and lists all truth assignments of all Boolean variables where T and P' are consistent, and these are the potential truth values to be assigned individually to each Boolean variable.

Algorithm 8: Helper functions for MinFixMult

SUBROUTINE InitFeasibility($\vec{a}, \vec{s}, g_x, g^\star$)
Output : $\mathfrak{C} : \{0,1\}^{\dim(\vec{a})} \to \{*\} \cup \mathbb{P}(\{0,1\}^{\dim(\vec{s})})$, a mapping such that for each assignment $\vec{v}$ of variables in $\vec{a}$, $\mathfrak{C}(\vec{v})$ returns the set of feasible truth value settings for $\vec{s}$ (such that g_x is consistent with $g^\star(\vec{v})$), or $*$ if $\vec{v}$ is impossible or irrelevant (i.e., $g^\star(\vec{v}) = *$)

let $\mathfrak{C}$ = empty mapping;
foreach *assignment* $\vec{v} \in \{0,1\}^{\dim(\vec{a})}$ *of* $\vec{a}$ **do**
  if $g^\star(\vec{v}) = *$ **then** // impossible or irrelevant
    let $\mathfrak{C}(\vec{v}) = *$; **continue**;
  let $\mathfrak{C}(\vec{v}) = \emptyset$;
  foreach *assignment* $\vec{u} \in \{0,1\}^{\dim(\vec{s})}$ *of* $\vec{s}$ **do**
    if $g_x(\vec{v} \parallel \vec{u}) = g^\star(\vec{v})$ **then let** $\mathfrak{C}(\vec{v}) = \mathfrak{C}(\vec{v}) \cup \{\vec{u}\}$;
return $\mathfrak{C}$;

SUBROUTINE UpdateFeasibility($\vec{a}, \vec{s}, \mathfrak{C}, d, g_d$)
Output : $\mathfrak{C}'$, a more constrained version of $\mathfrak{C}$ that reflects the effect of wiring variable s_d to the Boolean function g_d

let $\mathfrak{C}'$ = empty mapping;
foreach *assignment* $\vec{v} \in \{0,1\}^{\dim(\vec{a})}$ *of* $\vec{a}$ **do**
  if $\mathfrak{C}(\vec{v}) = *$ **then** // impossible or irrelevant
    let $\mathfrak{C}'(\vec{v}) = *$;
  else // only include settings consistent with g_d
    let $\mathfrak{C}'(\vec{v}) = \{\vec{u} \in \mathfrak{C}(\vec{v}) \mid u_d = g_d(\vec{v})\}$;
return $\mathfrak{C}'$;

SUBROUTINE PickSite($\vec{a}, \vec{s}, \mathfrak{C}, \mathcal{I}$)
Output : index $d \in \mathcal{I}$ as the next site to fix, and a partial Boolean function $g_d^\star$ of $\vec{a}$, represented as a mapping $\{0,1\}^{\dim(\vec{a})} \to \{*, 0, 1\}$, specified in accordance with the feasibility map $\mathfrak{C}$

foreach $i \in \mathcal{I}$ **do let** $c_i = 0$; // init accumulators for priority calculation
foreach *assignment* $\vec{v} \in \{0,1\}^{\dim(\vec{a})}$ *of* $\vec{a}$ **do**
  if $\mathfrak{C}(\vec{v}) = *$ **then continue**; // impossible or irrelevant
  foreach $i \in \mathcal{I}$ **do**
    let $r = |\{\vec{u} \in \mathfrak{C}(\vec{v}) \mid u_i = 1\}| \;/\; |\mathfrak{C}(\vec{v})|$;
    let $c_i = c_i + |r - 0.5|$; // prioritize uneven splits
let $d = \max \arg_{i \in I} c_i$;
let $g_d^\star$ = empty mapping;
foreach *assignment* $\vec{v} \in \{0,1\}^{\dim(\vec{a})}$ *of* $\vec{a}$ **do**
  if $\mathfrak{C}(\vec{v}) = *$ **then** // impossible or irrelevant
    let $g_d^\star(\vec{v}) = *$; **continue**;
  let $B = \{u_d \mid \vec{u} \in \mathfrak{C}(\vec{v})\}$; // 0/1 settings for s_d, deduplicated
  if $|B| = 1$ **then** // forced to choose one truth value
    let $g_d^\star(\vec{v})$ = the only element of B;
  else // $|B| = 2$, so choose either 0 or 1 (note that $B \neq \emptyset$)
    let $g_d^\star(\vec{v}) = *$;
return $(d, g_d^\star)$;

$a=1$	$b=2$	$c=3$	r_1, r_2
0	0	0	00,01
0	0	1	00,01
0	1	0	00
0	1	1	01,10,11
1	0	0	10,11
1	0	1	10,11
1	1	0	01,10,11
1	1	1	01,10,11

Figure 13: Constraint table for Example 16

Algorithm 7: MinFixMult($x, \mathcal{S}, l^\star, u^\star$)

Input : a formula x, a set $\mathcal{S}$ of disjoint subtrees (repair sites) of x, and a target bound $[l^\star, u^\star]$ for x to achieve by fixes
Output : a repair $(\mathcal{S}, F)$, where F maps each site in $\mathcal{S}$ to a formula

let $\mathcal{U}$ denote the set of atomic formulas in x that belong to none of the subtrees in $\mathcal{S}$;
let $\vec{a}, \vec{a}, \phi$ = MapAtomPreds($\mathcal{U} \cup \{l^\star, u^\star\}$);
let $g_l = \phi(l^\star)$ and $g_u = \phi(u^\star)$; // both are Boolean functions of $\vec{a}$
let $g^\star$ = BuildTruthTable($\vec{a}, \vec{a}, g_l, g_u$);
// Compute feasibility of truth values for repair sites:
let $\vec{s} = [s_1, s_2, \ldots]$ be the list of subexpressions in $\mathcal{S}$, denoted by the list of Boolean variables $\vec{s} = [s_1, s_2, \ldots]$;
let g_x be a Boolean function with variables $\vec{a} \parallel \vec{s}$, obtained from x by replacing each subexpression s_i with variable s_i, and replacing each atomic formula $t \in \mathcal{U}$ by $\phi(t)$;
let $\mathfrak{C}$ = InitFeasibility($\vec{a}, \vec{s}, g_x, g^\star$);
// Fix one site at a time, and incrementally update feasibility:
let F = empty mapping, and $\mathcal{I} = [1..\dim(\vec{s})]$;
while $\mathcal{I} \neq \emptyset$ **do**
  let $d, g_d^\star$ = PickSite($\vec{a}, \vec{s}, \mathfrak{C}, \mathcal{I}$);
  let g_d = MinBoolExp($g_d^\star$);
  let $F(s_d)$ = formula obtained from g_d by replacing each variable a_i with atomic formula a_i;
  let $\mathfrak{C}$ = UpdateFeasibility($\vec{a}, \vec{s}, \mathfrak{C}, d, g_d$);
  let $\mathcal{I} = \mathcal{I} \setminus \{d\}$;
return $(\mathcal{S}, F)$;

Example 17. *Consider the consistency table in Figure 12 from Example 16. The corresponding constraint table is shown in Figure 13.*

The blue-highlighted row reflects the highlighting in the consistency table, where only the truth assignments $(0,1)$, $(1,0)$ *or* $(1,1)$ *for* (r_1, r_2) *produce consistent evaluations for* T *and* P. *This indicates that when constructing* r_1 *and* r_2 *using the available atomic predicates* $a = 1, b = 2$ *and* $c = 3$, *we must guarantee the truth assignment* $(1,1,1)$ *for* $(a = 1, b = 2, c = 3)$ *would cause* (r_1, r_2) *to be evaluated*

$a=1$	$b=2$	$c=3$	r_1, r_2	r_1	r_2
0	0	0	00,01	0	$* \rightarrow 0$
0	0	1	00,01	0	$* \rightarrow 1$
0	1	0	00	0	0
0	1	1	01,10,11	$* \rightarrow 0$	1
1	0	0	10,11	1	$* \rightarrow 0$
1	0	1	10,11	1	$* \rightarrow 1$
1	1	0	01,10,11	$* \rightarrow 1$	0
1	1	1	01,10,11	$* \rightarrow 1$	$* \rightarrow 1$

Figure 14: Extended constraint table for Example 18

$r1$	$r2$	$a=1$	$b=2$	$c=3$	$l^\star$	$u^\star$	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	0
0	1	0	0	1	0	0	0	0
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	0
0	1	1	0	1	1	1	1	0
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	1
1	0	0	0	1	0	0	0	1
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	1
1	0	1	0	1	1	1	1	1
1	0	1	1	0	1	1	1	1
1	0	1	1	1	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 consistency table for Example 16

to either $(0,1)$, $(1,0)$ *or* $(1,1)$ *respectively. All other rows in the constraint table are constructed in the same manner and carry the same implication.*

Compute minimal fixes. The final step is to compute a fix for each repair site according to the constraint table. While a constraint table lists all possible simultaneous truth assignments for all repair sites (e.g. last column of Figure 13), we cannot make independent choices for the truth value of each repair site as dependencies exist. For example, in the highlighted row in Figure 13, if r_1 is assigned 0, then r_2 can only be assigned 1 for consistency. On the other hand, if r_1 is assigned 1, then r_2 can be assigned either 0 or 1. At this point, it is unclear how truth values can be assigned to each repair site to obtain minimum fixes, we thus follow a greedy procedure (line 10-14):

(1) Randomly pick a repair site r.
(2) Iterate over each row in the last column of the constraint table and give r the most flexible assignment (i.e. if both 0 and 1 are available, assign a don't-care).
(3) Use Quine-McCluskey's method to compute the minimal fix for the repair site.
(4) Update the assigned don't-cares to a determined value by evaluating the fix with the corresponding truth value assignment.
(5) Update the available options in the last column of the constraint table based on the truth assignment of r.

Example 18. *Continue from Example 17, the derivation process for* r_1 *and* r_2 *are shown in an extended constraint table in Figure 14. Given the previous procedure, we first give* r_1 *maximum flexibility for constructing its formula, which yields* $a = 1$*. We then update the don't-cares accordingly before computing* r_2*. As for* r_2*, we follow the same procedure and derive the truth assignment based on the truth values of* r_1*. Finally, the procedure yields* $c = 3$*. These are indeed the optimal fixes.*

In addition, running Qr-Hint*-optimized over Example 5 yields optimal fixes* $a = b$ *and* $d > 10 \wedge e < 5$ *for repair sites* x_4, (x_{10}, x_{12}).

B.3 Proof of Lemma 5.1

Proof of Lemma 5.1. We prove the correctness of WHERE-stage by proving the correctness of repair returned by Algorithm 1, i.e. applying the repair sites and fixes returned by Algorithm 1 yields a new formula P' such that $P^\star \Leftrightarrow P'$. The proof contains two steps:

Step 1: Algorithm 2 returns the correct repair sites. Assume that there exists a set of fixes $\mathcal{F}$ for a set of repair sites $\mathcal{S}$ in P. By Lemma 5.3, we can create a lower bound $P_\perp$ and an upper bound $P_\top$ such that $P' \in [P_\perp, P_\top]$, where P' is the formula obtained by applying fixes to repair sites. Since $P' \Leftrightarrow P^\star$, $P^\star \in [P_\perp, P_\top]$. Thus, if a set of fixes exists for a set of repair sites, $P^\star$ must fall within the corresponding repair bounds at the root of P. This validates the procedure for determining repair sites.

Step 2: Algorithm 3 returns the correct fixes. Given a lower bound $P_\perp$ and an upper bound $P_\top$ for P with respect to $\mathcal{S}$ such that $P^\star \in [P_\perp, P_\top]$, by Lemma 5.4, we can derive a set of fixes $\mathcal{F}$ for $\mathcal{S}$ through Example 8. □

B.4 Proof for Lemma 5.2

Proof. We use the induction over the structure of P to prove the minimality of the fix f for a single repair site s in P.

Base case. The base case is simply when s is P (i.e. the entire P is the repair site). In such case, f is a minimal DNF of $P^\star$ returned by the Quine-McCluskey's method. Thus, removing any clause from f causes $f \not\Leftrightarrow P^\star$.

Induction Step. When s is not P, there are three possible cases.

Case 1. P is in the form of $c_1 \wedge \ldots \wedge c_n$, where c_i is the repair site. The target bounds for c_i are derived to be $[P^\star, \top \wedge (P^\star \vee \neg \bigwedge_{j=1, j\neq i}^{n} c_j)]$ where the repair bounds of $\bigwedge_{j=1, j\neq i}^{n} c_j$ is simply $[\bigwedge_{j=1, j\neq i}^{n} c_j, \bigwedge_{j=1, j\neq i}^{n} c_j]$ as it does not contain any repair site. Running Quine-McCluskey's method over the target bounds of c_i yields a formula f, which is guaranteed to be in minimal DNF. Since we know $P^\star \Leftrightarrow \bigwedge_{j=1, j\neq i}^{n} c_j \wedge f$ and f is in minimal DNF, removing any of the clauses in f would cause $\bigwedge_{j=1, j\neq i}^{n} c_j \wedge f \Rightarrow P^\star$ but not vice versa.

Case 2. P is in the form of $c_1 \vee \ldots \vee c_n$, where c_i is the repair site. The target bounds for c_i are derived to be $[\perp \vee (P^\star \wedge \neg \bigvee_{j=1, j\neq i}^{n} c_j), P^\star]$ where the repair bounds of $\bigvee_{j=1, j\neq i}^{n} c_j$ are simply $[\bigvee_{j=1, j\neq i}^{n} c_j, \bigvee_{j=1, j\neq i}^{n} c_j]$. Running Quine-McCluskey's method over the target bounds of c_i yields a formula f, which is guaranteed to be in minimal DNF. Since

we know $P^\star \Leftrightarrow \bigvee_{j=1, j\neq i}^{n} c_j \vee f$ and f is in minimal DNF, removing any of the clauses in f would cause $\bigvee_{j=1, j\neq i}^{n} c_j \vee f \Rightarrow P^\star$ but not vice versa.

Case 3. P is in the form of $\neg c$. Here $P^\star \Leftrightarrow \neg f$. Since f is in minimal DNF, removing a clause results in $P^\star \not\Leftrightarrow \neg f$. □

B.5 Proof of Lemma 5.3

Proof of Lemma 5.3. We use induction over the structure of P to prove that for any subtree x in P (for which CreateBounds is invoked), $\text{CreateBounds}(x, \mathcal{S}[x])$ returns a correct bound for x: i.e., applying any repair to $\mathcal{S}[x]$ in x will result in a formula x' bounded by $\text{CreateBounds}(x, \mathcal{S}[x])$.

Base case. Suppose x is an atomic formula, CreateBounds returns $[x, x]$ which bounds x. When x is a repair site, CreateBounds returns [false, true], which certainly bounds x or any Boolean expression with which we can replace x.

Induction step. Suppose x is not atomic and is not itself a repair site. Let $\Theta = \text{op}(x)$ denote the logical operator at the root of x. Every repair on x (with the given $\mathcal{S}[x]$) is obtained by (potentially) repairing each child of x, but without changing Θ. In other words, every repair x results in $x' = \Theta_{c\in\text{Children}(x)} c'$, where c' is the result of some repair of c at sites $\mathcal{S}[c]$. By the inductive hypothesis, $\forall c \in \text{Children}(x) : c' \in [l_c, u_c] = \text{CreateBounds}(c, \mathcal{S}[c])$.

There are two cases depending on Θ. In the case that Θ is $\wedge$ or $\vee$, since $\forall c \in \text{Children}(x) : l_c \Rightarrow c' \Rightarrow u_c$, we have $\Theta_{c\in\text{Children}(x)} l_c \Rightarrow \Theta_{c\in\text{Children}(x)} c' \Rightarrow \Theta_{c\in\text{Children}(x)} u_c$, which means x' is within the bound returned by Line 6 of CreateBounds. In the case that Θ is $\neg$, since $l_c \Rightarrow c' \Rightarrow u_c$, we have $\neg u_c \Rightarrow \neg c' \Rightarrow \neg l_c$, which means x' is within the bound returned by Line 10 of CreateBounds. □

B.6 Proof of Lemma 5.4

Proof of Lemma 5.4. We use induction over the structure of P to prove that for any subtree x in P for which $\text{DeriveFixes}(x, \mathcal{S}, l^\star, u^\star)$ is invoked:

- (H1) $l^\star \Rightarrow u^\star$, and the bound $[l^\star, u^\star]$ implies (is equivalent or tighter than) the bound returned by $\text{CreateBounds}(x, \mathcal{S})$.
- (H2) The repair returned by $\text{DeriveFixes}(x, \mathcal{S}, l^\star, u^\star)$ yields some $x' \in [l^\star, u^\star]$.

Note that applying (H2) to the root of P proves Lemma 5.4.

Proving (H1) top-down. The base case is when x is the root of P; we only invoke DeriveFixes if $P^\star \in \text{CreateBounds}(x, \mathcal{S})$, so obviously $[l^\star, u^\star]$ implies $\text{CreateBounds}(x, \mathcal{S})$. For the induction step, assuming that (H1) holds for x, we now show that (H1) for each child of x for which DeriveFixes is invoked. There are three cases.

Case 1. x has form $\neg c$. Let $[l, u] = \text{CreateBounds}(x, \mathcal{S})$. By the inductive hypothesis $l \Rightarrow l^\star \Rightarrow u^\star \Rightarrow u$. Therefore $\neg u \Rightarrow \neg u^\star \Rightarrow \neg l^\star \Rightarrow \neg l$. In other words, we call DeriveFixes on c with a bound (Line 3) that implies $[\neg u, \neg l]$, which is $\text{CreateBounds}(c, \mathcal{S}[c])$ by Line 10 of CreateBounds.

Case 2. x has form $c_1 \wedge \ldots \wedge c_n$. In this case, Algorithm 3 divides the formula as $p_c \wedge p_{c'}$ where $p_c = c_i$, $p_{c'} = \bigwedge_{j=1, j\neq i}^{n} c_j$. We shall show that for $l_i^\star$ and $u_i^\star$ defined under Line 17, $[l_i^\star, u_i^\star]$ implies $[l_i, u_i] = \text{CreateBounds}(c_i, \mathcal{S}[c_i])$ for all i; the cases for all other children are symmetric. Clearly, $l_i \Rightarrow l^\star = l_i^\star$ and $u_i^\star = u_i \wedge (u^\star \vee \neg u_i') \Rightarrow u_i$. Furthermore, note that:

$$
\begin{aligned}
&l_i \Rightarrow u_i;\\
&l_i \Rightarrow u^\star \vee \neg u_i';\\
&l^\star \xRightarrow{\text{ind. hypo. and Line 6 of CreateBounds}} u_i \wedge u_i' \Rightarrow u_0;\\
&l^\star \xRightarrow{\text{ind. hypo.}} u^\star \Rightarrow u^\star \vee \neg u_i'.
\end{aligned}
$$

Therefore, $l_i^\star = l^\star \Rightarrow u_i \wedge (u^\star \vee \neg u_i') = u_i^\star$.

Case 3. x has form $c_1 \vee \ldots \vee c_n$. In this case, Algorithm 3 divides the formula as $p_c \vee p_{c'}$ where $p_c = c_i$, $p_{c'} = \bigvee_{j=1, j\neq i}^{n} c_j$. We shall show that for $l_i^\star$ and $u_i^\star$ defined in the branch starting on Line 19, $[l_i^\star, u_i^\star]$ implies $[l_i, u_i] = \text{CreateBounds}(c_i, \mathcal{S}[c_i])$ for all i; the cases for all other children are symmetric. Clearly, $l_i \Rightarrow l_i \vee (l^\star \wedge \neg l_i') = l_i^\star$ and $u_i^\star = u^\star \Rightarrow u_i$. Furthermore, note that:

$$
\begin{aligned}
&l_i \Rightarrow u_i;\\
&l_i \Rightarrow l_i \vee l_i' \xRightarrow{\text{ind. hypo. and Line 6 of CreateBounds}} u^\star;\\
&l^\star \wedge \neg l_i' \Rightarrow u_i;\\
&l^\star \wedge \neg l_i' \Rightarrow l^\star \xRightarrow{\text{ind. hypo.}} u^\star.
\end{aligned}
$$

Therefore, $l_i^\star = l_i \vee (l^\star \wedge \neg l_i') \Rightarrow u^\star = u_i^\star$.

Proving (H2) bottom-up. For the base case, when $x \in \mathcal{S}$, assuming the correctness of MinFix, Lemma 5.3 and (H1) ensure that $\text{MinFix}(x, l^\star, u^\star)$ yields a repaired formula in $[l^\star, u^\star]$.

For the inductive step, assuming that (H2) holds for each child of x, we now show that (H2) holds for x. There are three cases.

Case 1. x has form $\neg c$. By the inductive hypothesis, DeriveFixes on c returns a repair that results in some c' such that $\neg u^\star \Rightarrow c' \Rightarrow \neg l^\star$. Clearly, the same repair, which is returned by $\text{DeriveFixes}(x, \mathcal{S}, l^\star, u^\star)$, changes x to $\neg c'$, which satisfies $l^\star \Rightarrow \neg c' \Rightarrow u^\star$.

Case 2. x has form $c_1 \wedge \ldots \wedge c_n$. By the inductive hypothesis, for all $1 \leq i \leq n$, DeriveFixes on c_i returns a repair that results in some c_i' such that $l^\star \Rightarrow c_i' \Rightarrow u_i \wedge (u^\star \vee \neg u_i')$. The repair returned by DeriveFixes on x results in $\bigwedge_{i=1}^{n} c_i'$. Clearly, $c_1' \wedge \ldots \wedge c_n' \Leftarrow l^\star \wedge l^\star \Leftrightarrow l^\star$. Also,

$$
\begin{aligned}
\bigwedge_{i=1}^{n} c_i' &\Rightarrow \bigwedge_{i=1}^{n} \left(u_i \wedge (u^\star \vee \neg u_i')\right)\\
&\Leftrightarrow \bigwedge_{i=1}^{n} u_i \wedge \left(u^\star \vee \left(\bigwedge_{i=1}^{n} \neg u_i'\right)\right)\\
&\Leftrightarrow \left(\bigwedge_{i=1}^{n} u_i \wedge u^\star\right) \vee \left(\bigwedge_{i=1}^{n} u_i \wedge \bigwedge_{i=1}^{n} \neg u_i'\right)\\
&\Leftrightarrow u^\star \vee \left(\bigwedge_{i=1}^{n} u_i \wedge \bigwedge_{i=1}^{n} \neg u_i'\right)\\
&\Leftrightarrow u^\star \vee \bot\\
&\Leftrightarrow u^\star.
\end{aligned}
$$

Algorithm 9: FixSelect$(P, \vec{o}, \vec{o^\star})$

Input : a formula P and two expression lists $\vec{o}$ and $\vec{o^\star}$
Output : a pair (Δ^-, Δ^+), where $\Delta^- \subseteq [1..\dim(\vec{o})]$ is a subset of indices of $\vec{o}$ and $\Delta^+ \subseteq [1..\dim(o^\star)]$ is a subset of indices of $\vec{o^\star}$

1 **let** $\Delta^- = \emptyset$;
2 **let** $\Delta^+ = \emptyset$;
3 **let** $n = \min(\dim(\vec{o}), \dim(o^\star))$;
4 **foreach** $o_i \in \vec{o}, o_i^\star \in \vec{o^\star}, 1 \le i \le n$ **do**
5 **if** IsSatisfiable$_{\mathcal{C}}(o_i \neq o_i^\star)$ **then**
6 **let** $\Delta^- = \Delta^- \cup \{i\}$;
7 **let** $\Delta^+ = \Delta^+ \cup \{i\}$;
8 **foreach** $o_i \in \vec{o}, n < i \le \dim(\vec{o})$ **do**
9 **let** $\Delta^- = \Delta^- \cup \{i\}$;
10 **foreach** $o_i^\star \in \vec{o^\star}, n < i \le \dim(o^\star)$ **do**
11 **let** $\Delta^+ = \Delta^+ \cup \{i\}$;
12 **return** (Δ^-, Δ^+);

Case 3. x has form $c_1 \vee ... \vee c_n$. By the inductive hypothesis, for all $1 \le i \le n$, DeriveFixes on c_i returns a repair that results in some c_i' such that $l_i \vee (l^\star \wedge \neg l_i') \Rightarrow c_i' \Rightarrow u^\star$. The repair returned by DeriveFixes on x results in $\bigvee_{i=1}^{n} c_i'$. Clearly, $c_1' \vee ... \vee c_n' \Rightarrow u^\star \vee u^\star \Leftrightarrow u^\star$. Also,

$$\begin{aligned}
\bigvee_{i=1}^{n} c_i' &\Leftarrow \bigvee_{i=1}^{n} (l_i \vee (l^\star \wedge \neg l_i')) \\
&\Leftrightarrow \bigvee_{i=1}^{n} l_i \vee \left(l^\star \wedge \bigvee_{i=1}^{n} \neg l_i' \right) \\
&\Leftrightarrow \left(\bigvee_{i=1}^{n} l_i \vee l^\star \right) \wedge \left(\bigvee_{i=1}^{n} l_i \vee \bigvee_{i=1}^{n} \neg l_i' \right) \\
&\Leftrightarrow l^\star \wedge \top \\
&\Leftrightarrow l^\star.
\end{aligned}$$

□

C GROUP BY STAGE SUPPLEMENT

C.1 Necessity of Fixing GROUP BY

We show that it is necessary to fix GROUP BY.

Lemma C.1. *Consider two single-block SQL queries Q_1 and Q_2, where Q_1 has no* GROUP BY *or aggregation, while Q_2 has* GROUP BY *and/or aggregation but no* HAVING. *Q_1 and Q_2 cannot be equivalent under bag semantics, assuming that no database constraints are present and there exists some database instance for which either Q_1 or Q_2 returns a non-empty result.*

Proof of Lemma C.1. Suppose for some instance D, both Q_1 and Q_2 return the same non-empty results. Pick any $T \in \text{Tables}(Q_1)$. Create a new instance D' by duplicating the contents of T in the same table (i.e., doubling the multiplicity of each tuple in T) while keeping all other tables unchanged. Since Q_1 has no GROUP BY or aggregation, the multiplicity of each tuple in $Q_1(D')$ will be increased by a factor of 2^c compared with that in $Q_1(D)$, where $c > 1$ is the count of T in $\text{Tables}(Q_1)$ (which is multiset). Hence, the size of $Q_1(D')$ is strictly larger than $Q_1(D)$. On the other hand, consider Q_2, which has GROUP BY and/or aggregation. Between $Q_2(D')$ and $Q_2(D)$, the grouping of intermediate join result tuples remains the same, except the number of duplicates within each group. Hence, the size of $Q_2(D')$ remains the same as $Q_2(D)$, which is the total number of groups. Therefore, $Q_1(D')$ and $Q_2(D')$ are different. □

C.2 Proof for Lemma 6.2

Proof of Lemma 6.2. **Correctness.** We prove the correctness of GROUP BY-stage by showing $\vec{o} \setminus \Delta^- \cup \Delta^+$ is equivalent to $\vec{o^\star}$. Assuming $\vec{o} \setminus \Delta^- \cup \Delta^+$ is not equivalent to $\vec{o^\star}$, then among all possible pairs of tuples, there must exist t_1, t_2 such that $\bigwedge_i (o_i[t_1] = o_i[t_2]) \nLeftrightarrow \bigwedge_i (o_i^\star[t_1] = o_i^\star[t_2])$. This implies that IsSatisfiable$(P[t_1] \wedge P[t_2] \wedge G^\star \wedge o_i[t_1] \neq o_i[t_2])$ (line 6) and/or IsSatisfiable$(P[t_1] \wedge P[t_2] \wedge G \wedge o_i^\star[t_1] \neq o_i^\star[t_2])$ (line 11) must be satisfiable, which contradicts the fact that IsSatisfiable returns no false positive.

Strong Minimality of Δ^-. Assuming $\Delta^- \not\subseteq \Delta_\circ^-$ (i.e., there exists an $o_x \in \vec{o}$ such that $o_x \in \Delta^-$ but $o_x \notin \Delta_\circ^-$), and $\vec{o} \setminus \Delta_\circ^- \cup \Delta^+$ is equivalent to $\vec{o^\star}$. This implies that $o_x[t_1] = o_x[t_2]$ holds for all possible t_1, t_2, which contradicts the fact that p is evaluated to true (otherwise o_x should not be added to Δ^- according to the algorithm). Since IsSatisfiable does not return false positive, o_x does not exist and thus $\Delta^- \subseteq \Delta_\circ^-$.

Weak Minimality of Δ^+. Assuming $\Delta^+ = \{o_x^\star\}$ and $\vec{o} \setminus \Delta_\circ^-$ is equivalent to $\vec{o^\star}$ (denoted by $\vec{o} \setminus \Delta_\circ^- \equiv \vec{o^\star}$). Following Algorithm 4, $\Delta^- \subseteq \Delta_\circ^-$ due to strong minimality. This indicates that $\vec{o} \Rightarrow \vec{o^\star}$ (i.e. $P[t_1] \wedge P[t_2] \wedge G \Rightarrow P[t_1] \wedge P[t_2] \wedge G^\star$) but $\vec{o} \nLeftarrow \vec{o^\star}$ (i.e. $P[t_1] \wedge P[t_2] \wedge G \nLeftarrow P[t_1] \wedge P[t_2] \wedge G^\star$). However, $\Delta^+ = \{o_x^\star\}$ indicates that IsSatisfiable$(P[t_1] \wedge P[t_2] \wedge G \wedge o_x^\star[t_1] \neq o_x^\star[t_2])$ return true, where $o_x^\star$ is a conjunct in $G^\star$. This implies $\vec{o} \setminus \Delta^- \nRightarrow \vec{o^\star}$, thus further implying IsSatisfiable returns a false positive because $\vec{o} \setminus \Delta_\circ^- \equiv \vec{o^\star}$ was assumed at the beginning. Since IsSatisfiable never returns false positive, no such $o_x^\star$ can exist in Δ^+, and thus there does not exist a corresponding $\Delta_\circ^-$ such that $\vec{o} \setminus \Delta_\circ^- \equiv \vec{o^\star}$.

□

D HAVING STAGE SUPPLEMENT

We use the following base context as default for testing satisfiability for HAVING. Note that constraints from WHERE and created variables are not included because they differ from query to query.

$$C: \left\{ \begin{array}{r}
\mathbf{X}, \mathbf{Y}, \mathbf{Z} \text{ have type } \text{Array}(\mathbf{N}) \\
\mathbf{i}, \mathbf{c} \text{ have type Integer} \\
\texttt{SUM, COUNT, AVG, MAX, MIN} \text{ has type } \text{Array}(\mathbb{N}) \to \mathbb{N} \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{i} : \mathbf{X}[\mathbf{i}] + \mathbf{Y}[\mathbf{i}] = \mathbf{Z}[\mathbf{i}] \Rightarrow \texttt{SUM}(\mathbf{X}) + \texttt{SUM}(\mathbf{Y}) = \texttt{SUM}(\mathbf{Z}) \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{i} : \mathbf{X}[\mathbf{i}] - \mathbf{Y}[\mathbf{i}] = \mathbf{Z}[\mathbf{i}] \Rightarrow \texttt{SUM}(\mathbf{X}) - \texttt{SUM}(\mathbf{Y}) = \texttt{SUM}(\mathbf{Z}) \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{i}, \mathbf{c} : \mathbf{X}[\mathbf{i}] \times \mathbf{c} = \mathbf{Y}[\mathbf{i}] \Rightarrow \texttt{SUM}(\mathbf{X}) \times \mathbf{c} = \texttt{SUM}(\mathbf{Y}) \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{i}, \mathbf{c} : \mathbf{X}[\mathbf{i}] \div \mathbf{c} = \mathbf{Y}[\mathbf{i}] \Rightarrow \texttt{SUM}(\mathbf{X}) \div \mathbf{c} = \texttt{SUM}(\mathbf{Y}) \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{i} : \mathbf{X}[\mathbf{i}] + \mathbf{Y}[\mathbf{i}] = \mathbf{Z}[\mathbf{i}] \Rightarrow \texttt{AVG}(\mathbf{X}) + \texttt{AVG}(\mathbf{Y}) = \texttt{AVG}(\mathbf{Z}) \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{i} : \mathbf{X}[\mathbf{i}] - \mathbf{Y}[\mathbf{i}] = \mathbf{Z}[\mathbf{i}] \Rightarrow \texttt{AVG}(\mathbf{X}) - \texttt{AVG}(\mathbf{Y}) = \texttt{AVG}(\mathbf{Z}) \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{i}, \mathbf{c} : \mathbf{X}[\mathbf{i}] \times \mathbf{c} = \mathbf{Y}[\mathbf{i}] \Rightarrow \texttt{AVG}(\mathbf{X}) \times \mathbf{c} = \texttt{AVG}(\mathbf{Y}) \\
\forall \mathbf{X}, \mathbf{Y}, \mathbf{i}, \mathbf{c} : \mathbf{X}[\mathbf{i}] \div \mathbf{c} = \mathbf{Y}[\mathbf{i}] \Rightarrow \texttt{AVG}(\mathbf{X}) \div \mathbf{c} = \texttt{AVG}(\mathbf{Y})
\end{array} \right\}$$

E SELECT STAGE SUPPLEMENT

The pseudocode for fixing SELECT is shown in Algorithm 9.

The correctness and optimality are the following:

Lemma E.1. *We say that two lists of* SELECT *expression are equivalent if they produce the same set of columns in the same ordering. Let* $(\Delta^-, \Delta^+) = \mathit{FixSelect}(P, \vec{o}, \vec{o^\star})$. *Assuming that subroutine* $\mathit{IsSatisfiable}_C$ *returns no false positive, we have:*

Correctness: *Qr-Hint's* SELECT*-stage hint leads to a fixed working query* Q_5 *that 1) passes the viability check (*$\vec{o}$ *and* $\vec{o^\star}$ *are equivalent); 2) satisfies* $Q_5 \equiv Q^\star$. *This applies to both* SPJ *and* SPJA *queries.*

Strong minimality of (Δ^-, Δ^+) **for *SPJ*** *Let* $(\Delta_\circ^-, \Delta_\circ^+)$ *denote the minimal* (Δ^-, Δ^+) *respectively, then for any* $(\Delta_\circ^-, \Delta_\circ^+)$ *that make* $\vec{o}$ *and* $\vec{o^\star}$ *equivalent,* $\Delta^- \subseteq \Delta_\circ^-, \Delta^+ \subseteq \Delta_\circ^+$.

Proof of Lemma E.1. **Correctness.** We prove the correctness by showing $\vec{o} \setminus \Delta^- \cup \Delta^+$ is equivalent to $\vec{o^\star}$. Since $\mathit{IsSatisfiable}_C$ does not return false positive, $\vec{o}[i], \vec{o^\star}[i]$ are guaranteed to be added to Δ^-, Δ^+ respectively upon "satisfiable" or "unknown", and replacing $\vec{o}[i]$ with $\vec{o^\star}[i]$ guarantees the correctness of expression on position i. In addition, for any extra expressions in Δ^+, Δ^- that do not have a counterpart in the other list, they are removed to ensure the number of expressions is the same between the SELECT of $Q^\star, Q$.

Strong minimality of (Δ^+, Δ^-) **in *SPJ* queries.** Assuming $\Delta^- \subsetneq \Delta_\circ^-$ (i.e. there exists an $o_x \in \vec{o}$ s.t. $o_x \in \Delta^-$ but $o_x \notin \Delta_\circ^-$), and $\vec{o} \setminus \mathit{Delta}_\circ^- \cup \Delta^+$ is equivalent to $\vec{o}$. This implies that either of the following is true: 1) o_x is redundant in Q's SELECT and needs to be removed; 2) $\mathit{IsSatisfiable}_C(o_x \neq o_x^\star)$ returns "satisfiable". In the former case, o_x has to be removed and must be in $\Delta_\circ^-$ (otherwise $\Delta_\circ^-$ is not correct); in the latter case, $o_x \notin \Delta_\circ^-$ implies $\mathit{IsSatisfiable}_C$ returns a false positive, which contradicts with our assumption. Thus $\Delta^- \subseteq \Delta_\circ^-$. $\Delta^+ \subseteq \Delta_\circ^+$ follows a similar proof. □

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 2. The wrong queries and hints are shown in Table 3 (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 2: 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 3: Wrong queries and the hints provided (QR-HINT hints are in blue).

Question (a)	Question	Find the names of all beers served at James Joyce Pub.			
	Solutions	SELECT beer FROM serves WHERE bar = 'James Joyce Pub';			
	Error Statistics	Total Wrong Query		22	
		FROM		8	1. Wrong table; 2. Extra table (cross join with bar)
		WHERE		9	Wrong bar name or typo
		SELECT		5	SELECT * or bar alone instead of beer
Question (b)	Question	Find names and addresses of bars that serve Budweiser at a price higher than 2.20.			
	Solutions	SELECT name, address FROM bar, serves WHERE bar.name = serves.bar AND beer = 'Budweiser' AND price > 2.20;			
	Error Statistics	Total Wrong Query		126	Note: 3 of them cannot be processed due to outer-join
		FROM		10	Missing either Bar table or Serves table
		WHERE		96	1. Missing join condition; 2. Use >= instead of >
		SELECT		17	1. Missing Columns; 2. Column order is wrong
Question (c)	Question	Find the names of drinkers who like Corona and frequent James Joyce Pub at least twice a week.			
	Solutions	SELECT likes.drinker FROM likes, frequents WHERE likes.beer = 'Corona' AND likes.drinker = frequents.drinker AND frequents.bar = 'James Joyce Pub' AND frequents.times_a_week >= 2;			
	Error Statistics	Total Wrong Query		143	Note: 20 of them cannot be processed due to usage of set operation, outer joins, complex subqueries
		FROM		11	1. Wrong table involved (serves); 2. Unnecessary drinker table (false positive, true error in SELECT/WHERE)
		WHERE		105	1. Missing join condition; 2. Using > instead of >=, or wrong number; 3. Missing condition on beer or bar
		SELECT		6	SELECT * instead of name
		GROUP BY		1	GROUP BY wrong columns
Question (d)	Question	Find the name of each drinker who likes at least two beers.			
	Solution 1	SELECT drinker FROM likes GROUP BY drinker HAVING COUNT(*) >= 2;			
	Solution 2	SELECT DISTINCT l1.drinker FROM likes l1, likes l2 WHERE l1.drinker = l2.drinker AND l1.beer <> l2.beer;			
	Error Statistics	Total Wrong Query		50	Note: 12 of them cannot be processed due to usage of set operations
		Solution 1	FROM	1	Wrong table
			GROUP BY	1	Group by 1
			HAVING	18	1. Using > instead of >=; 2. COUNT(DINSTINCT *)
			SELECT	4	Extra column COUNT
		Solution 2	FROM	5	Extra/wrong tables (likes / frequents)
			WHERE	2	Wrong conditions: l1.beer = l2.beer, l1.drinker <> l2.drinker
			SELECT	7	Missing DISTINCT

Table 4: Student Query Statistics