Qr-Hint: Formally Verified and AI-Explained SQL Tutoring

Zhouyi Jiang
zhouyi.jiang@duke.edu
Duke University
Durham, NC, USA

Yihao Hu
yihao.hu@duke.edu
Duke University
Durham, NC, USA

Amir Gilad
amirg@cs.huji.ac.il
Hebrew University
Jerusalem, Israel

Sudeepa Roy
sudeepa@cs.duke.edu
Duke University
Durham, NC, USA

Jun Yang
junyang@cs.duke.edu
Duke University
Durham, NC, USA

ABSTRACT

Providing effective feedback on SQL queries is a notorious challenge in database education, as "logical bugs" — where a query runs but produces incorrect results — are difficult for automated tools to diagnose. In this demonstration, we present Qr-Hint, a system designed to support debugging for both students and educators. Rather than relying on test cases or simple syntactical comparison between incorrect and correct queries, Qr-Hint identifies the smallest syntactic edit to a wrong query to make it semantically equivalent to a correct one. It then synthesizes its findings into actionable, natural-language hints using AI, guiding learners to fix the wrong query without revealing the answer. We showcase Qr-Hint in two distinct roles: as an interactive tutor for students that guides them through the debugging process step by step, and as a helper tool for teaching assistants and instructors to identify issues in student queries quickly. Through live scenarios involving missing joins, aggregation errors, and duplicate handling, we demonstrate how Qr-Hint reduces the manual burden of debugging while improving the pedagogical quality of feedback.

PVLDB Reference Format:
Zhouyi Jiang, Yihao Hu, Amir Gilad, Sudeepa Roy, and Jun Yang. Qr-Hint: Formally Verified and AI-Explained SQL Tutoring. PVLDB, 19(12): 4566 - 4569, 2026.
doi:10.14778/3827998.3828067

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

1 INTRODUCTION

SQL proficiency is a cornerstone of modern data analytics, yet mastering the language remains a significant hurdle for novices. While database engines readily trap syntax errors, learners frequently struggle with "logical bugs" — queries that execute successfully but yield incorrect results due to subtle semantic misunderstandings. In educational settings, the burden of diagnosing these errors creates a massive bottleneck. For students, debugging demands rigorous verification of their logic against expected outcomes, a skill that is often still in development. For instructors and teaching assistants (TAs), even with access to reference solutions, manually deciphering a student's flawed logic and suggesting tailored fixes accordingly is cognitively demanding and unscalable — making the generation of personalized pedagogical hints nearly impossible for large classes. While AI assistants (e.g., ChatGPT) can expedite debugging, they lack formal correctness guarantees and often fail as query complexity increases. Furthermore, AI-generated feedback is frequently unstructured when addressing multiple concurrent errors, which can confuse students and undermine the systematic problem-solving process necessary for true mastery.

While there is plenty of previous work in the database community for checking query equivalence [3, 5, 11], and some tools [2, 8, 9] have been developed to come up with instances that differentiate the semantics of two queries, none of them is able to attribute errors to syntactical parts of the query and suggest how to fix them.

Example 1 (from [7]). *Consider the motivating example in [7] with 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 an 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 F1, Frequents F2, Serves S1,
     Serves S2
WHERE L.drinker = F1.drinker AND F1.bar = S1.bar
  AND L.drinker = F2.drinker AND F2.bar = S2.bar
  AND L.beer = S1.beer AND S1.beer = S2.beer
  AND S1.price <= S2.price
GROUP BY F1.drinker, L.beer, S1.bar
HAVING F1.drinker = 'Amy';
```

Now consider a wrong student query Q:

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

While the above wrong query seems easy to debug, it is not immediately obvious how to fix it. First, syntactically different queries might be semantically similar, so relying solely on syntax to propose fixes is ineffective. For example, adding `HAVING` to Q in Example 1 is not necessary because the condition `drinker='Amy'` in Q's `WHERE` serves the same purpose. Second, declaring part of Q being "wrong" is confusing, since the remainder of Q might be revised to compensate for it. For example, `s1.price>s2.price` in Q might seem "wrong", but the following correct condition actually includes it: `(s1.price>s2.price OR s1.price=s2.price)`. Finally, users might be easily

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828067

overwhelmed when multiple issues arise from a wrong query, and it is hard to fix all errors simultaneously.

To this end, we propose Qr-Hint,[1] a system that automates the debugging of incorrect queries against a reference solution. While our original framework [7] guarantees the semantic correctness of repairs, the resulting syntactic edits can be counterintuitive for learners. To bridge this gap, Qr-Hint transforms rigorous, technical repairs into intuitive explanations by leveraging recent AI advancements, merging the accuracy of formal methods with the usability of AI to streamline the debugging workflow.

Example 2. *To fix the wrong query Q in Example 1, Qr-Hint automatically generates the sequence of hints below:*

Clause	*Qr-Hint repair*	*Repair explained by AI in natural language*
FROM	*Frequents needed*	Think about which tables contain the information about bars that Amy frequents. Your current FROM clause may not include all the necessary tables to connect Amy to those bars.
WHERE	*s1.price>s2.price ↦ s1.price≥s2.price*	Think about how your WHERE clause compares prices: using s1.price>s2.price might exclude bars where the price is equal. Consider whether you want to count all bars with a price less than or equal to the current bar's price to determine its rank.

Note that the above hints come in strict order as the query evolves after repairing each clause. Qr-Hint first focuses on FROM *and will only proceed to* WHERE *after* FROM *is "viable". After adding Frequents to* FROM*, the user needs to add appropriate join conditions in* WHERE*; otherwise the second step above would suggest additional repairs.*

Can AI/LLM do the tasks by Qr-Hint? How does Qr-Hint aid database education in classrooms? While modern AI is powerful enough to analyze queries and assist with debugging by itself, it essentially acts as a probabilistic oracle and is prone to errors. Qr-Hint mitigates this unreliability by bridging between principled algorithms and AI. It guarantees the semantic correctness of its repairs through provable verification, while employing AI strictly as a translation layer to turn those technical fixes into friendly natural-language explanations. This deliberate design choice aligns Qr-Hint with established best practices in computer science education [6]. While standard AI tools often provide complete solutions all at once — a practice that inadvertently encourages superficial learning and bypasses the "productive struggle" necessary for true comprehension — Qr-Hint's step-by-step, hint-first architecture acts as a crucial metacognitive scaffold. By withholding the direct answer and instead guiding students through sequential, formally verified corrections, the system actively trains learners to think systematically and internalize the debugging process. In addition, Qr-Hint is designed to serve as a diagnostic assistant for teaching staff. By rapidly pinpointing exact logical flaws and providing verified solutions, it eliminates the cognitive drain of manual debugging, allowing educators to quickly diagnose issues and scale pedagogically sound support to much larger classrooms.

2 TECHNICAL BACKGROUND

Query Fragments. We consider select-project-join queries in SQL with an optional single level of grouping and aggregation. These are single-block SQL queries with SELECT (potentially with DISTINCT), FROM (without JOIN operators), and WHERE (with condition defaulting to TRUE if missing) clauses, together with optional GROUP BY and HAVING clauses. We can handle queries with common table expressions (WITH) and subqueries in FROM that are aggregation-free, as well as non-outer JOINs in FROM and arbitrary nested positive correlated subqueries (EXISTS, ANY, SOME s.t. the conditions involve them are not negated) by rewriting the query into single-block SQL. While Qr-Hint supports single-block query currently, the currently supported fragment covers the most fundamental and important features for beginners in the educational setting.

Given a reference query $Q^\star$ and a wrong query Q, Qr-Hint examines Q in five stages corresponding to five clauses involved in a single block of query. The progression follows FROM → WHERE → GROUPBY → HAVING → SELECT. Examination of each clause involves: (1) a viability check to test if two clauses are equivalent; (2) finding proper repairs to make the clauses pass the viability check.

FROM Stage. The viability check of FROM clause is simply checking if the wrong query Q has the same multiset of tables as the reference query $Q^\star$. If this is not the case, Qr-Hint stops and reports any redundant or missing tables.

WHERE Stage. Given the FROM clauses in $Q^\star$ and Q are equivalent, Qr-Hint model the WHERE clauses as logical formulae, the viability check is simply the equivalence between two formulae. When the check fails, the true challenge in repairing it is that it can be hard to mark a condition or a subformula as "semantically incorrect". For example, suppose the reference query enforces serves.price ≠ 5 while a student writes serves.price > 5 instead. We cannot say serves.price > 5 is incorrect because there exists a correct query that contains this condition (e.g., serves.price > 5 OR serves.price < 5).

To tackle the challenge, Qr-Hint uses the following approach:

(1) Qr-Hint first parses the incorrect formula into an abstract syntax tree where internal nodes are logical connectives and leaf nodes are atomic predicates.
(2) Qr-Hint then uses a meticulously designed algorithm to search for potential sets of disjoint subtrees called "repair sites", such that the incorrect formula has the potential to become equivalent to the reference formula by replacing these repair sites with a set of new subformulae.
(3) Given a list of sets of repair sites, for each set of repair sites, Qr-Hint then compute a corresponding set of subformulae called "fixes". Finally, Qr-Hint compute the edit distance for all pairs of repair sites and fixes and choose the repair sites and fixes that incur the smallest edit distance.

Example 3. *For simplicity, consider the following logical formulae $P^\star$ and P where A, B, C, D, E are integers:*

$P^\star$: $(A{=}B \wedge B{=}C) \vee B{>}C$
P: $B{>}C \vee (A{=}B \wedge B{<}C)$

To fix P, we can either swap $B{<}C$ (repair site) with $B{=}C$ (fix) or $A{=}B \wedge B{<}C$ (repair site) with $A{=}B \wedge B{=}C$ (fix), and the first repair site and fix incurs a smaller edit distance.

GROUP BY Stage. Assuming $Q^\star$, Q have equivalent FROM and WHERE clauses, repairing grouping is tricky as shown below:

Example 4. *Consider two queries over tables in Example 1:*

```
SELECT f.bar FROM frequents f, serves s WHERE f.bar =
    s.bar GROUP BY f.bar, s.beer; -- Q*
```

[1] The research paper that developed the approach appeared in SIGMOD 2024 [7].

```
SELECT s.bar FROM frequents f, serves s WHERE f.bar =
     s.bar GROUP BY s.beer || '@' || s.bar, s.bar; -- Q
```

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

Instead of simply comparing the grouping columns one by one (which often fails because different expressions can result in the same groups), QR-HINT takes a holistic approach based on data partitioning. Assuming the FROM and WHERE are aligned, the viability check tests if the student's grouping expressions and the reference's grouping expressions produce identical partitions of the data. Essentially, it checks whether any two rows that are grouped together in the student's query are also grouped together in the reference query (and vice versa), while accounting for the constraints imposed by WHERE. This ensures that the grouping logic is semantically equivalent even if the syntax looks completely different.

If the viability check fails, QR-HINT reports any missing or redundant GROUP BY expressions in the wrong query.

HAVING Stage. QR-HINT treats the HAVING clause as a logical formula, but with added sophistication to handle aggregate functions and data groups. QR-HINT performs a context-aware viability check to verify whether the student's conditions and the reference conditions are mathematically equivalent. This check goes beyond simple comparisons by incorporating the context established in previous query steps. For example, QR-HINT recognizes mathematical properties of aggregates (e.g., 2*SUM(D) is equivalent to SUM(D*2)). Crucially, it also imports constraints from the WHERE clause, allowing it to validate correctness even when logic is moved between WHERE and HAVING or when variables filtered earlier are used interchangeably in the grouping condition. Like WHERE, QR-HINT reports repair sites and fixes if the viability check fails.

SELECT Stage. For the SELECT clause, QR-HINT verifies the final output structure by comparing the ordered list of projected expressions in the student's query against the reference. Assuming all previous logic (filtering, grouping, etc.) has been successfully aligned, the viability check tests that both queries produce the same number of columns and that the expression at each specific position is semantically equivalent to its counterpart in the reference solution. This allows QR-HINT to precisely flag errors such as missing columns, redundant outputs, or incorrect ordering of the projection list when the viability check fails.

Handling Duplicates. When a wrong query involves DISTINCT or positive correlated subqueries (e.g., EXISTS), it is possible that the wrong query returns the same set but a different bag of tuples compared with the reference query. To accommodate such a situation, QR-HINT emplace the following strategies for both the wrong query and the reference query:

- In case of DISTINCT, QR-HINT adds all SELECT expressions to the GROUP BY clause when GROUP BY is not present in a query. Otherwise, QR-HINT checks for the presence of DISTINCT keyword after the wrong query passes the SELECT viability check.
- In case of positive correlated subqueries (e.g., EXISTS), QR-HINT first rewrites the query to flatten all subqueries into a single block of SFWGH. Since we only consider aggregate-free subqueries, such flattening does not change the set of output tuples. Hence, we can first apply all viability checks to ensure that the query returns the same set of tuples as the reference query, and then hint to the user that the query returns a different bag.

EXAMPLE 5. *Using the same schema in Example 1, consider the following problem: Find the names of bars that serve at least two distinct beers liked by Eve. The reference query is given as follows:*

```
SELECT s.bar FROM Serves s
WHERE EXISTS(SELECT * FROM Likes l
             WHERE l.beer = s.beer AND l.drinker = 'Eve')
GROUP BY s.bar HAVING COUNT(*) >= 2;
```

Consider a wrong student query:

```
SELECT s.bar FROM Serves s, Likes l
WHERE l.beer = s.beer AND l.drinker = 'Eve'
GROUP BY s.bar HAVING COUNT(*) >= 2;
```

The student query is set-equivalent to the reference (it finds the correct bars) but fails bag equivalence. The use of join combines tuples from Serves *and* Likes*, potentially inflating the count if the relationship is one-to-many. In contrast,* EXISTS *acts as a filter that preserves the original cardinality of* Serves*. QR-HINT detects this bag discrepancy and hints to the user to use a correlated subquery to ensure the aggregation counts distinct beers rather than join matches.*

Assumptions and Guarantees. Due to the undecidability of query equivalence [10], it is generally impossible to decide if a student's query is wrong given a reference query. Since the most common way to decide the correctness of a student's query in an educational setting is by comparing query output over multiple test instances, we make a reasonable assumption that students' queries inputted to QR-HINT must return a different output from the reference query on at least one test instance, and QR-HINT guarantees query equivalence after all fixes are applied to the wrong query.

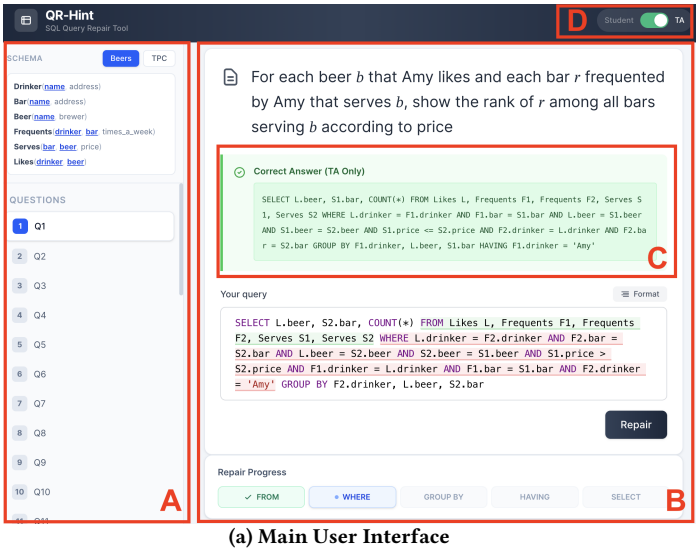
3 SYSTEM IMPLEMENTATION

Overall Architecture. The system is developed as a web application, with an API server in Flask and the user interface in React. The back-end algorithms are implemented in Python 3.10, and all logical formula implication/equivalence testing is done using the Z3 SMT Solver [4]. Since QR-HINT takes a syntactic approach to suggest fixes, it requires only access to the database schema rather than a complete database instance, making QR-HINT lightweight. Finally, the back-end uses Apache Calcite [1] as the query parser to parse the syntax trees of the WHERE and HAVING clauses.

Procedures and Hint Translation. QR-HINT fixes a wrong query in the following order: FROM→WHERE→GROUPBY →HAVING→SELECT. A user cannot progress to the next stage until the query passes the current stage's viability check. If the query fails at any stage, QR-HINT highlight the query syntax that needs to be revised. In instructor mode, QR-HINT additionally proposes a concrete revision to the query, which is not available in student mode. In both modes, users can call the AI assistant to explain the semantic error caused by the highlighted syntax. QR-HINT has meticulously prepared prompts for instructors and students, respectively, to instruct AI to translate syntactic fixes to explanations/hints in natural language.

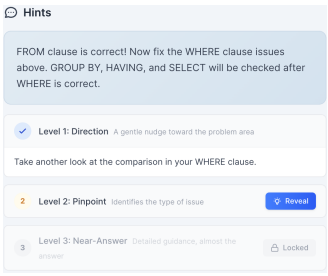
4 DEMONSTRATION PLAN

Our demonstration will utilize a database whose schema is expanded from Example 1, with tables Drinker(name, address), Bar(name,



(a) Main User Interface

(b) Repair Sites and Fixes



(c) AI-generated Hint

Figure 1: The user interface of QR-HINT.

address), Beer(name, brewery), Likes(drinker, beer), Frequents(drinker, bar, times_a_week), and Serves(bar, beer, price). We will showcase QR-HINT in two modes:

- **the TA/instructor Mode**, which provides full visibility into repair sites, specific fixes, and generated explanations; and
- **the student Mode**, which simulates the learner's experience by presenting only natural-language hints.

We will invite conference attendees to role-play as students or TAs, submitting incorrect queries and observing how QR-HINT guides them toward the correct solution step by step.

Step 1: User Interface Introduction. The user interface of QR-HINT is shown in Figure 1a. Upon landing on the web app, a user can choose the database schema and the question they would like to work on in Component A. The user can toggle between student/instructor mode using the switch in Component D.[2] The user can then examine the question and enter an incorrect query in the textbox in Component B. We will have some preloaded queries that the user can explore and edit, or write their own queries. After clicking on "Repair", QR-HINT displays progress at the bottom of Component B to indicate which clause is problematic. The repair (shown in Figure 1b) will also be displayed under Component B, and QR-HINT will provide multiple levels of AI-explained hints from indirect to obvious (shown in Figure 1c), and users can unlock the direct hints after thinking through the indirect hints. Note that in the student mode, Component C in Figure 1a and the suggested fix in Figure 1b will be disabled.

Step 2: Demonstration Scenarios. We begin the demonstration by walking through Example 1 with conference attendees, which showcases how QR-HINT fixes a query's `FROM` and `WHERE` clauses. In addition, we prepare another example to showcase how QR-HINT fixes `GROUP BY`, `HAVING`, `SELECT` and duplicates.

Step 3: Interactive Exploration. Following the above scenarios, we will open the floor for attendees to modify queries or propose their own "wrong" solutions to the demonstrated examples. Attendees can use the student mode to test their debugging skills using QR-HINT, or the instructor mode to see the underlying logical proofs and generated repairs in real time. Furthermore, we will prepare additional questions for conference attendees to try out.

[2]This is for demo only. In the actual deployment, each user is assigned a role.

ACKNOWLEDGMENTS

The work of Amir Gilad was funded by the Israel Science Foundation (ISF) under grant 1702/24, the Scharf-Ullman Endowment, and the Alon Scholarship.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[6] Xinying Hou, Zihan Wu, Xu Wang, and Barbara J Ericson. 2025. Personalized Parsons Puzzles as Scaffolding Enhance Practice Engagement Over Just Showing LLM-Powered Solutions. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 2*. 1483–1484.

[7] Yihao Hu, Amir Gilad, Kristin Stephens-Martinez, Sudeepa Roy, and Jun Yang. 2024. Qr-Hint: Actionable Hints Towards Correcting Wrong SQL Queries. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[8] Hanze Meng, Zhengjie Miao, Amir Gilad, Sudeepa Roy, and Jun Yang. 2023. Characterizing and Verifying Queries Via CINSGEN. In *Companion of the 2023 International Conference on Management of Data*. 143–146.

[9] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Ratest: Explaining wrong relational queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data*. 1961–1964.

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

[11] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proc. VLDB Endow.* 17, 11 (July 2024), 3602–3614. https://doi.org/10.14778/3681954.3682024