I-Rex: An Interactive Relational Query Explainer for SQL

Zhengjie Miao, Tiangang Chen, Alexander Bendeck, Kevin Day,
Sudeepa Roy, Jun Yang
Duke University, Durham, NC, USA
{zjmiao,sudeepa,junyang}@cs.duke.edu,
{tiangang.chen,alexander.bendeck,kevin.day}@duke.edu

ABSTRACT

We demonstrate I-REX[1], a system designed to help users understand SQL query evaluation and debug SQL queries. I-REX lets users interactively "trace" the evaluation of complex SQL queries, including those with correlated subqueries. I-REX also explains why a query returns an incorrect answer with respect to a reference query over a test database instance—a common use case in education and software regression testing. To avoid the cognitive overload caused by debugging over a large database instance, I-REX lets users focus on smaller instances contained in the large one (which we call "counterexamples") that still distinguish the two queries. Supporting these features for SQL queries poses two key challenges. First, unlike debugging for procedural languages, it is not clear how to trace a declarative SQL query, because its execution plan often differs from how it was originally written. I-REX offers a novel interface for tracing SQL query evaluation in a way faithful to how queries are written syntactically, even for complex queries involving multiple levels of nesting and correlation. Second, we need a method for finding small counterexamples that handles the complexity of practical SQL. I-REX extends provenance support for SQL in non-trivial ways to work with various query constructs. This demonstration walks through use cases in which I-REX helps users understand and debug SQL queries.

PVLDB Reference Format:
Zhengjie Miao, Tiangang Chen, Alexander Bendeck, Kevin Day, Sudeepa Roy, and Jun Yang. I-Rex: Interactive Relational Query Explainer for SQL. *PVLDB*, 13(12): 2997-3000, 2020.
DOI: https://doi.org/10.14778/3415478.3415528

1. INTRODUCTION

Data analytics has emerged as one of the most important skills for modern jobs. Much analysis of structured data happens through SQL. However, learning and debugging SQL can be challenging, even for people with considerable experience with procedural programming languages, partly because of the declarative nature of SQL. I-REX is a system designed to help users understand SQL query evaluation and debug SQL queries. I-REX addresses two key challenges. First, despite a plethora of debugging tools for procedural languages, there exist few tools for SQL—it is not even clear how to "trace" a declarative query because it is often optimized and executed differently from the way it is written. Second, in many use cases, users may find out that a SQL query returns a wrong answer for a given test database instance, yet debugging the query over a large database is slow and poses significant cognitive burden, especially for novices. These challenges are compounded by the complexity of the SQL language, with constructs such as nested, correlated subqueries. In the following, we illustrate these challenges with examples and motivate our approach.

For the first challenge of tracing SQL evaluation, a strawman solution is to convert the SQL query of interest into a logical plan of relational algebra operators, and trace the evaluation of this plan operator by operator, allowing users to examine all intermediate results in a bottom-up fashion. While this solution is more palatable than tracing the execution plan, there can still be a huge disconnect between the logical plan and the original query, especially for those with constructs such as correlated subqueries that are frequently used but have no direct counterparts in relational algebra.

(a) Drinker relation

name	addr	
Ben	101 W.M. St.	d_1
Coy	101 W.M. St.	d_2
Dan	300 N.D. St.	d_3

(b) Frequents relation

drinker	bar	times	
Ben	The Edge	3	f_1
Ben	ToT	1	f_2
Dan	JJ Pub	1	f_3
Dan	ToT	2	f_4

(c) Likes relation

drinker	beer	
Ben	Amstel	l_1
Ben	Budweiser	l_2
Coy	Dixie	l_3
Dan	Amstel	l_4
Dan	Budweiser	l_5
Dan	Corona	l_6

(d) Serves relation

bar	beer	price	
The Edge	Amstel	2.75	s_1
The Edge	Budweiser	2.00	s_2
JJ Pub	Amstel	3.00	s_3
JJ Pub	Corona	3.25	s_4
JJ Pub	Dixie	3.00	s_5
ToT	Corona	2.50	s_6
ToT	Dixie	2.75	s_7

Figure 1: Example relations with tuples identifiers.

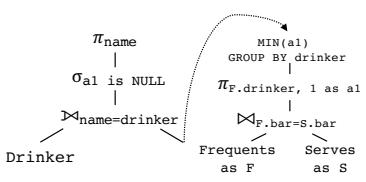

Figure 2: Decorrelated query plan for Example 1.

(a) Result of ref. Q_0

name	
Coy	r_1
Dan	r_2

(b) Result of wrong Q

name	
Coy	r_3

Figure 3: Results of queries in Example 2.

[1]A video and other information about our system can be found at `https://dukedb-hnrq.github.io/`; we will continue to update this website as we add new features or release new versions.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415528

EXAMPLE 1 (TRACING SQL QUERIES). *Figure 1 shows a database about drinkers, beers, bars, and relationships among them. Consider the following query:*

```
SELECT name FROM Drinker
WHERE NOT EXISTS
 (SELECT s.bar FROM Serves s, frequents f
  WHERE f.bar=s.bar AND name=f.drinker);
```

Note that `name` *in the* `WHERE` *clause of the nested subquery refers to a column from the* `Drinker` *table in the outer query. Because of this correlation, we cannot "trace" intermediate results of the query in a bottom-up fashion from the subquery to the outer query. One possible workaround is to decorrelate the query, such that the resulting query can be translated into a logical plan that allows bottom-up tracing. This workaround is universal as decorrelation is routinely done in most database query optimizers. However, a serious drawback is that the decorrelated query may not resemble the original. For example, for the query above, the popular open-source SQL query planning tool, Apache Calcite [1], would generate an overly complex decorrelated logical plan depicted in Figure 2 (already simplified for presentation). Tracing this decorrelated query plan will not help users understand the original query and may cause more confusions.*

Instead, I-REX provides a novel interactive interface that allows users to trace query evaluation in a way faithful to how the query is written originally. Intuitively, each subquery is executed in a "context" with specific variable bindings provided during the evaluation of its outer queries. For example, the result of the subquery above (within `WHERE NOT EXISTS (...)`) depends on the particular `name` value from the `Drinker` tuple being examined by the outer query. I-REX's interface adheres to this standard evaluation semantics of correlated subqueries and supports tracing through nested SQL query blocks, without requiring any additional knowledge of relational algebra or its mapping from SQL.

For the second challenge of debugging queries over a large database, we focus on a common use case that arises in education and software regression testing, where users notice that a query is wrong because it returns different answers from a "reference" (correct) query on a test database instance.[2] To avoid the cognitive overload caused by debugging over a large instance, I-REX lets users focus on small instances contained in the large one (which we call "counterexamples") that still distinguish the two queries.

EXAMPLE 2 (SMALL COUNTEREXAMPLES). *Suppose a SQL assignment asks students to write a query to find drinkers who frequent* only *bars that serve* some *beers they like. The instructor provides the reference query Q_0:*

```
SELECT name FROM drinker
WHERE NOT EXISTS (
  SELECT bar FROM frequents
  WHERE drinker = name AND bar NOT IN
   (SELECT bar FROM serves, likes
    WHERE drinker=name AND serves.beer=likes.beer));
```

A student writes the following incorrect query Q, which considers the three relationship tables in a different order and actually finds drinkers who frequent only *bars that serve* only *beers they like (we have observed this mistake to be typical for students learning SQL in our database courses):*

```
SELECT name FROM Drinker
WHERE NOT EXISTS (
  SELECT s.bar FROM Serves s, Frequents f
  WHERE f.bar=s.bar AND name=f.drinker
  AND s.beer NOT IN (
    SELECT l.beer FROM likes l WHERE name=l.drinker));
```

Over the test database instance shown in Figure 1, Q_0 outputs tuples $r_1 = \langle Coy \rangle, r_2 = \langle Dan \rangle$, while Q outputs only one tuple $r_3 = \langle Coy \rangle$. Instead of showing the full instance with 20 tuples, we can show a counterexample consisting of only 5 tuples d_3, f_3, l_4, s_3, s_5(highlighted in the tables), over which Q_0 would still return r_2 (Dan) but Q will return an empty result. Besides reducing the cognitive overload, this small counterexample also "pinpoints" an error in Q: that Q would fail to find a drinker when the drinker frequents some bar serving both beers he likes and beers he does not like. For a real test database instance involving thousands of tuples, the small counterexample still contains only 5 tuples, significantly smaller than the entire database.

In this demonstration, we will showcase a representative debugging scenario, to illustrate how I-REX can help explain errors in user queries using small counterexamples and further help users trace the evaluation of their queries to understand their behaviors. With each counterexample pinpointing one error in the user query, users can focus on fixing their queries one "bug" at a time, until they pass the whole test database instance.

Related work. Conceptually, I-REX shares its two key features with our previous system, RATEST [6], which supports evaluation tracing and finding small counterexamples for *relational algebra* queries. RATEST has been deployed in undergraduate database courses and benefited hundreds of students; its success motivated us to support similar features for SQL. It turns out that moving from procedural relational algebra to declarative SQL, also a much more complex language, requires both a complete redesign of the front-end tracing interface and significant reworking of the back-end processing methods. As motivated in Example 1, the simple bottom-up tracing of RATEST does not work for I-REX. Support for *provenance* [5], which underpins our techniques for finding small counterexamples, has been lacking for general SQL queries; I-REX has extended provenance support to more SQL constructs beyond SPJUDA (select-project-join-union-difference-aggregate) queries.

Besides RATEST, a number of other systems are also related to I-REX. For tracing SQL evaluation, Dietrich and Grust [3] built an observational debugger that allows users to mark some parts of the query and then observe the intermediate results produced by the selected parts, helping users learn how a specific query component executes. A discussion of additional related work can be found in [6]. For instance, the work on "Why-Not" queries aim to explain missing answer tuples by data or query modifications that would instead include them in the query result. In contrast, I-REX intends to explain and trace more complex SQL queries while remaining faithful to the original database (albeit via a much smaller counterexample) and original queries.

2. IMPLEMENTATION AND SYSTEM

Figure 5 depicts the architecture of I-REX. As a web application, I-REX allows users to issue SQL queries in the front-end. Then, the two major components of the back-end handle the query separately: the *counterexample finder* returns a small counterexample database w.r.t. the user query and a given reference query; the *query evaluation tracer* supports interactive tracing in the front-end by decomposing the user query and executing rewritten subqueries against the database. More details are presented below.

Capturing How-Provenance for SQL Query. In our recent work [6], we studied the *smallest counterexample problem* (SCP):

[2] Note that returning correct answers for a finite number of test instances does *not* guarantee that a query is correct—however, this practice of using test instances to gauge correctness is both common and practical given the general undecidability of query equivalance testing.

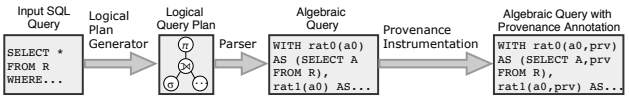

Figure 4: Provenance instrumentation pipeline for SQL.

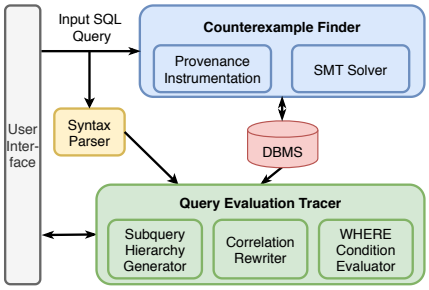

Figure 5: I-REX system architecture.

given a database instance D and two queries Q_1 and Q_2 where $Q_1(D) \neq Q_2(D)$, find a sub-instance $D' \subseteq D$ with minimum number of tuples such that $Q_1(D') \neq Q_2(D')$. We proposed a solution to SCP by considering each tuple t in $Q_1(D) \setminus Q_2(D)$ (or $Q_2(D) \setminus Q_1(D)$) and finding the smallest $D' \subseteq D$ (*witness* for t) such that still $t \in Q_1(D') \setminus Q_2(D')$. The solution consists of first capturing Boolean *how-provenance* [5] (encoding how an output tuple t is derived from input tuples by annotating input tuples with Boolean variables), and then finding a satisfying model for the how-provenance formula with the least number of variables set to true, which can be efficiently done using an SMT (satisfiability modulo theories) solver. We developed additional techniques in [6] to improve the scalability of this approach; experiments show that we can deliver interactive response time even for databases with millions of tuples.

In [6], we implemented a how-provenance tracker by translating relational algebra queries into SQL fragments and then instrumenting queries according to the logic of SPJUDA operators. But this rewriting does not apply to general SQL queries, especially when there are correlated subqueries. Decorrelation is theoretically possible, but not well supported in practice, and often introduces extraneous operations including aggregation that complicate provenance. Therefore, we avoid decorrelation and apply an instrumentation pipeline inspired by [4] as shown in Figure 4 to capture how-provenance for SQL. First, we obtain the logical query plan of the input SQL query using the `explain` command of CockroachDB [2]. The resulting tree-structured logical plan uses an `Apply` operator instead of decorrelating the query. The `Apply` operator takes a relation R as input, evaluates an expression E for each row $r \in R$, and then combines the results of each $E(r)$ as the output. Next, we build a new algebraic query from the logical query plan. For instance, queries with `NOT EXISTS` subqueries like the one we show in Example 1 can be represented as an *antijoin* between the outer relation and the inner query's result. Furthermore, if there are deeper nested correlated subqueries (such as Q and Q_0 in Example 2), `Apply` is introduced and columns from outer relations are replaced with placeholders. Then we rewrite this algebraic query into SQL with additional provenance information, and execute it separately for every value binding. Beyond correlated subqueries, we also have to consider `NULL` and bag semantics, which are not handled in [6]. For example, outerjoins may introduce `NULL` if there are input tuples that do not join with others.

Tracing Query Evaluation through Blocks and Context. As discussed earlier, tracing is non-trivial due to SQL's complexity—for example, it would not make sense to trace a query with correlated subqueries as a tree of operators in a bottom-up manner. Existing approaches all require transforming SQL queries into an alternative representation, e.g., (i) tracing through the physical query plan returned by the query optimizer (which may not resemble the original query at all), (ii) tracing through a tree of operators using a new operator for correlation (e.g., as what we did to capture provenance), or (iii) tracing through equivalent procedural code. All these approaches impose additional cognitive burden: users must familiarize themselves with a new representation and must be able to translate insights they gained from tracing the alternative representation back to how to fix their original SQL query. Instead, we want to trace a SQL query at a *logical* level and in a way consistent with how the query is written.

Instead of letting a query optimizer dictate how to execute a SQL query, I-REX uses the syntax of the query to derive an evaluation structure that faithfully reflects how the query is written. We break the query syntactically down into a hierarchy of *blocks* (subqueries); a block always corresponds to a contiguous substring of the original query string and can be executed under a *context*, which provides values for columns referenced in this block but coming from outside the block. The user can choose which block to focus on individually, and when the user focuses on a particular block to explore, our interface highlights the corresponding part of the original query string. For query Q in Example 2, the root block in the hierarchy represents the entire query; this root has one child block Q_1, which represents the nested `SELECT` statement that serves as the input to `NOT EXISTS` in the outermost `WHERE`; Q_1 in turn has another child block $Q_{1.1}$ representing the subquery that serves as an input to `NOT IN`.

If a block b represents a correlated subquery, it may refer to columns from tables in `FROM` clauses of b's ancestor blocks, thus the behavior of b needs to be understood from the context provided by evaluating its ancestors. Note that while we treat the hierarchy of blocks as a tree, the column references for constructing the context is modeled as a directed graph. For example, in $Q_{1.1}$, the column reference `name` refers to the `Drinker` table in Q's `FROM` clause. Imagine that Q examines one `Drinker` row at a time; the particular `Drinker` row being examined by Q provides the context for $Q_{1.1}$ and a specific binding for `name` to be used for evaluating $Q_{1.1}$. In general, the context and bindings for a block b can be provided by any ancestor blocks of b.

When a user focuses on a block b, our interface clearly shows the current context and bindings for b's external (correlated) references, if any. The user can adjust the context and explore how b's result changes accordingly. For an uncorrelated subquery whose result does not depend on the context, our interface intelligently omits the option to adjust context, avoiding clutter and confusion.

Tracing Evaluation of Each Block. Given the context, our interface further allows the user to focus on specific contexts to understand the evaluation of any block b. For example, consider block Q_1, which is a two-table `SELECT` with a conjunctive `WHERE` with three predicates. Below the context for Q_1, our interface shows the contents of the two input tables for Q_1 as well as its output (under the current context provided by Q). By selecting a combination of input rows, the user can see the truth values of the predicates in `WHERE` to understand why this particular combination yields an output row (then it will be highlighted automatically) or not (then a warning will be displayed). In general, our interface would display the entire `WHERE` condition as a Boolean expression tree whose leaves are annotated with truth values; hovering on ref-

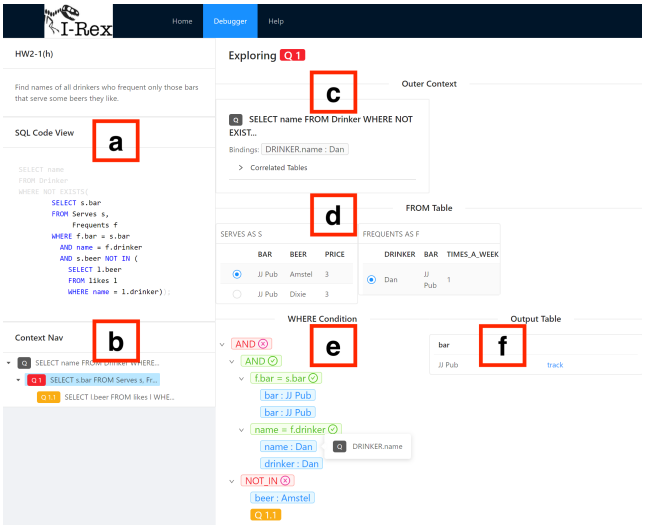

Figure 6: Overview of I-REX debugger interface.

erences made within the condition reveals the corresponding bindings. Conversely, the user can select a specific output row, and the input rows contributing to it will be automatically highlighted.

We support these block-level tracing functionalities by dynamically constructing a query for a block b from its corresponding subquery. Specifically, we replace any external references with bindings from b's current context, and rewrite the query to capture provenance between its input and output rows, as well as truth values returned by constituent predicates in `WHERE`. We execute the constructed query against the backend and extract relevant information to update the state of our interface. The query for block b is constructed every time the user chooses the block and selects the specific binding (and the result would be cached in the frontend), since evaluating subqueries against all possible bindings is not necessary and would hamper interactivity. Moreover, by enabling users to select different external reference bindings and different combinations of input rows, I-REX allows examinations of context that might never arise by simply executing the whole query.

Note that the third predicate in Q_1's `WHERE` further contains a subquery $Q_{1.1}$. If the user is interested in why that predicate evaluates to true or false, she can drill down to block $Q_{1.1}$; in that case, the currently selected combination of input rows to Q_1 becomes part of the context for $Q_{1.1}$.

Besides the SPJ block described above, we also support a variety of other blocks, including blocks with grouping, `HAVING`, and aggregation, as well as SQL set/bag operations such as `UNION` and `EXCEPT (ALL)`. In general, a complex block may be traced as multiple steps, each with its own intermediate result (e.g., a block involving `HAVING` may additionally show the rows before grouping as well as the groups before `HAVING`); the user will be able to select any intermediate result row and trace how it is derived from input(s) and how it contributes to the output of the block. We omit details from this paper but will show details in our demonstration.

3. DEMONSTRATION

We will walk through a use case of I-REX in the classroom setting, where students write SQL queries to answer questions (with corresponding hidden reference queries) against a database. The student first writes a wrong query; I-REX will show a small counterexample and let the student trace the wrong query over it.

Consider again Example 2, where the problem asks for "all drinkers who frequent *only* bars that serve *some* beers they like"; instead, the wrong student query would find drinkers who "frequent *only* bars that serve *only* beers they like." Even if we show a small counterexample and results of both correct query Q_0 and wrong query Q, the student may still have a hard time seeing why her query fails to return $\langle$`Dan`$\rangle$; so she goes to the Debugger panel to find out why.

The debugging process begins with the root block that takes `Drinker` as input but returns empty output. The student starts by selecting the row `Dan` in `Drinker` in the "From Tables" view, to see how the query evaluates this input tuple. The details of `WHERE` conditions will be displayed, and the student finds that the `NOT EXISTS` condition is highlighted red, indicating that it fails. The student then wonders, "Why is the result of the subquery in `NOT EXISTS(...)` not empty?" To answer this question, she focuses on block Q_1 by clicking on the corresponding node in the context navigation ($\boxed{b}$ in Figure 6). I-REX then displays the context information for Q_1 including bindings for external references (e.g., `Drinker.name = 'Dan'`, $\boxed{c}$), input tables for Q_1 ($\boxed{d}$), and its output ($\boxed{f}$).

Upon seeing the output tuple $\langle$`JJ Pub`$\rangle$, the student clicks on it to locate input tuples in `FROM` tables (`Serves` and `Frequents`) that yield this output. Three satisfied `WHERE` conditions will be displayed: `f.bar=s.bar`, `'Dan'=f.drinker`, and `s.beer NOT IN (..)` ($\boxed{e}$). Finally, the student goes to the innermost block $Q_{1.1}$ and finds that beer `Dixie` is served in `JJ Pub` but is not liked by `Dan`. She keeps exploring block Q_1 by selecting another `Serves` tuple $\langle$`JJ Pub, Amstel, 3`$\rangle$, which does not yield any output because the `NOT IN` condition is not satisfied. During the exploration process, the student would understand that her query fails to find the drinker who frequents some bar that serves some beer (from block Q_1) not liked by the drinker (from block $Q_{1.1}$), and might reconsider the interaction between `Likes` and `Serves` in her query.

Demonstration scenarios. During our demonstration, audience can (i) walk through example queries from a SQL assignment to see typical bugs in student submissions and how I-REX helps reveal them; (ii) look into some extremely complex queries that are difficult to understand without the help of I-REX; (iii) trace their own queries over the sample database, which is helpful in understanding SQL evaluation semantics; and (iv) work on our assignment problems to experience how students can use I-REX.

4. ACKNOWLEDGMENTS

This work is supported in part by NSF Awards IIS-1408846, IIS-1552538, IIS-1703431, IIS-1718398, IIS-1814493, and by NIH award R01EB025021. Thanks to Jeffrey Luo for contributions on the final version of the system.

5. REFERENCES

[1] `https://calcite.apache.org/`.
[2] `https://www.cockroachlabs.com/`.
[3] B. Dietrich and T. Grust. A SQL debugger built from spare parts: Turning a SQL: 1999 database system into its own debugger. In *SIGMOD*, pages 865–870, 2015.
[4] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE*, pages 174–185, 2009.
[5] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
[6] Z. Miao, S. Roy, and J. Yang. (RATest:) Explaining Wrong Queries Using Small Examples. In *SIGMOD (paper and demo)*, pages 503–520, 1961–1964, 2019.