RATest: Explaining Wrong Relational Queries Using Small Examples

Zhengjie Miao, Sudeepa Roy, and Jun Yang
Duke University
{zjmiao,sudeepa,junyang}@cs.duke.edu

ABSTRACT

We present a system called RATest, designed to help debug relational queries against reference queries and test database instances. In many applications, e.g., classroom learning and regression testing, we test the correctness of a user query Q by evaluating it over a test database instance D and comparing its result with that of evaluating a reference (correct) query Q_0 over D. If $Q(D)$ differs from $Q_0(D)$, the user knows Q is incorrect. However, D can be large (often by design), which makes debugging Q difficult. The key idea behind RATest is to show the user a much smaller database instance $D' \subseteq D$, which we call a *counterexample*, such that $Q(D') \neq Q_0(D')$. RATest builds on data provenance and constraint solving, and employs a suite of techniques to support, at interactive speed, complex queries involving differences and group-by aggregation. We demonstrate an application of RATest in learning: it has been used successfully by a large undergraduate database course in a university to help students with a relational algebra assignment.

CCS CONCEPTS

• **Information systems** → *Database utilities and tools.*

KEYWORDS

Data provenance; Explanations; Relational query grading

ACM Reference Format:
Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. RATest: Explaining Wrong Relational Queries Using Small Examples. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320236

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3320236

1 INTRODUCTION

In many applications, we test correctness of database queries—which we call *candidate queries*—by comparing their results with those of *reference queries* (which are deemed correct) over some *reference database instance*. For example, during software development, when changes are made to queries used by applications, we use regression testing to verify new queries by comparing their results against those of original queries (presumed correct) over test databases. In a classroom setting, where students learn to write relational queries, we can evaluate a student query and compare its result with that of the correct solution query over a test database instance. In these applications, a key to debugging candidate queries is the reference database instance D, which serves as a *counterexample* that illustrates the difference in the results of candidate query Q and reference query Q_0.

Unfortunately, in practice, D is often large, either because it is based on real or production data, or it is synthesized to cover numerous corner cases and to stress-test the scalability of the queries. Such a large counterexample would take too much effort to understand where the inequivalence of the queries comes from. For example, in a classroom setting, if we want to use real data from the DBLP publication database in an assignment, D would contain millions of tuples, and the difference in query results may contain many tuples as well, easily overwhelming students.

The key idea behind our approach, implemented in a system we call RATest, is to show the user a much smaller counterexample database instance $D' \subseteq D$, such that $Q(D') \neq Q_0(D')$, i.e., still enough to illustrate the difference between Q and Q_0. D' would also satisfy the same database constraints as D. Besides the small size of D', which makes it easier to work with, this approach has a number of other advantages. **First**, while it is possible to generate a counterexample D' completely unrelated to D (like [4]), constraining D' to be a subset of D helps preserve some context for users by using the same data values and relationships from D. Also, technically, knowing that D itself is a counterexample (albeit large) helps us find small counterexamples more easily. For example, without access to D, it would be difficult to automatically generate values that satisfy one predicate involving complex SQL functions but not another. **Second**, our counterexample can choose to illustrate just one particular result tuple

name	major	
Mary	CS	t_1
John	ECON	t_2
Jesse	CS	t_3

(a) Student relation S

name	number	dept	grade	
Mary	216	CS	100	t_4
Mary	230	CS	75	t_5
Mary	208D	ECON	95	t_6
John	316	CS	90	t_7
John	208D	ECON	88	t_8
Jesse	216	CS	95	t_9
Jesse	316	CS	90	t_{10}
Jesse	330	CS	85	t_{11}

(b) Registration relation R

name	major	
John	ECON	r_1

(c) Result of reference query Q_0

name	major	
Mary	CS	r_2
John	ECON	r_3
Jesse	CS	r_4

(d) Result of candidate query Q

Figure 1: Example relations with tuples identifiers.

that differentiates Q and Q_0, which allows users to focus on identifying and fixing one issue with their query at a time. **Third**, users get enough help in debugging just by seeing D', $Q(D')$, and $Q_0(D')$ (along with a description of what the query is intended to return). There is no need to reveal the reference query Q_0 or the entire test database instance D; this feature is particularly useful in the classroom setting.[1]

We illustrate the problem of generating a small counterexample with an example.

Example 1. *Consider two relations storing information about students and the courses they took:* $S(\underline{\texttt{name}}, \texttt{major})$ *and* $R(\underline{\texttt{name}}, \underline{\texttt{number}}, \underline{\texttt{dept}}, \texttt{grade})$. *Suppose a user is asked to write a relational algebra query to find student records for those who took exactly one Computer Science (CS) course. One correct solution (reference query)* Q_0 *is the following:*

$$S \bowtie \left(\begin{array}{l} \pi_{\texttt{name}} \sigma_{\texttt{dept='CS'}}(R) - \pi_{R_1.\texttt{name}} \\ \left(\begin{array}{l} (\sigma_{\texttt{dept='CS'}} \rho_{R_1}(R)) \bowtie_{R_1.\texttt{name} = R_2.\texttt{name} \wedge R_1.\texttt{number} \neq R_2.\texttt{number}} \\ (\sigma_{\texttt{dept='CS'}} \rho_{R_2}(R)) \end{array} \right) \end{array} \right).$$

An incorrect candidate query may be the following, which actually finds students who took one or more CS courses:

$$Q : \pi_{\texttt{name,major}}(S \bowtie \sigma_{\texttt{dept='CS'}} R).$$

A reference database instance and the results of these queries over this instance are shown in Figure 1. The tuples $r_2 = \langle \texttt{Mary}, \texttt{CS} \rangle$ *and* $r_3 = \langle \texttt{Jesse}, \texttt{CS} \rangle$ *are in the result of Q but not in the result of* Q_0*. To convince the user that Q is wrong, we could provide the entirety of S and R as a counterexample comprising 11 tuples. However, a smaller counterexample simply contains three tuples,* t_1, t_4, t_5 *(a subset of tuples pertaining to Mary), over which* Q_0 *would return an empty result while Q would return just* r_2 *(Mary). This counterexample is not only smaller, but also helps "pinpoint" the problem. For a real database with tens of thousands of students, there can be a vast difference between the size of the entire database and that of the smallest counterexample (which remains at* 3*).*

In our recent work [7], we solve the problem of finding small counterexamples by building on the work in data provenance and constraint solving. We identify connections to known problems, such as *minimal witness*, and show why and how to adapt our problem definition for complex cases such as when queries involve aggregation. We tackle the challenges of finding the smallest counterexample at interactive speed and supporting complex queries involving differences, group-by aggregation, as well as SQL functions and operators in selection and join conditions. A detailed discussion on related work can be found in [7].

In this demonstration, we showcase a representative application of RATest in a classroom setting, which we deployed earlier in a large undergraduate database course at Duke University. We set up RATest for a relational algebra assignment, with a large test database and the correct solution queries, all of which are hidden from students. A student can submit candidate queries to RATest. If a query returns an incorrect result, RATest will show a counterexample with a small number of tuples drawn from the test instance, and show how the result of the submitted query differs from the correct one on the counterexample. The student can revise the query and get further feedback that may help reveal additional bugs, until the query returns the same result as the hidden solution query on the full hidden test instance.

2 BACKGROUND AND PROBLEM

Consider a database instance D where $D \models \Gamma$ for a given set Γ of integrity constraints, and two queries Q_1 and Q_2 such that $Q_1(D) \neq Q_2(D)$. The *smallest counterexample problem* (SCP for short) is to find a $D' \subseteq D$ with minimum number of tuples such that $D' \models \Gamma$ and $Q_1(D') \neq Q_2(D')$, i.e., D' also differentiates Q_1 and Q_2.[2]

As stated in Section 1, we connected SCP to data provenance: if Q_1 and Q_2 return different results over a D, then there must exist one tuple $t \in Q_1(D) \setminus Q_2(D)$ or $t \in Q_2(D) \setminus Q_1(D)$. We refer to the concept of *witnesses* [3]: a witness for a tuple t w.r.t. a query Q and database instance D is a subinstance $D' \subseteq D$ where $t \in Q(D')$. Therefore, for any counterexample $D' \subseteq D$ for which $Q_1(D') \neq Q_2(D')$, there exists some tuple t in $(Q_1 - Q_2)(D')$ or $(Q_2 - Q_1)(D')$, i.e., D' is a witness of t w.r.t. $Q_1 - Q_2$ or $Q_2 - Q_1$ and D.

Thus, a reasonable approach to SCP is to consider each tuple in $Q_1(D) \setminus Q_2(D)$ (or $Q_2(D) \setminus Q_1(D)$), find its smallest witness w.r.t. $Q_1 - Q_2$ (or $Q_2 - Q_1$, resp.) and D; we then pick the smallest witness overall. If Q_1 and Q_2 are monotone, this approach always yields the smallest counterexample [7].

[1] It is conceivable that the system can be abused to recover D with some effort. We do not address this issue in our current system and have not observed any such abuses in a real classroom deployment, but the problem can be an interesting direction for future work.

[2] Without loss of generality, we assume that Q_1 and Q_2 have the same result schema (otherwise, we do not need an instance to illustrate their difference).

In [7], we also obtained complexity results for SCP for different classes of queries. For queries involving projection, join, and difference, it is noteworthy that finding the smallest witness for a result tuple is already NP-hard in data complexity, even when the queries are of bounded sizes.

Aggregate Queries. More challenges arise once we consider group-by aggregation queries. First, the approach of finding minimal witnesses for result tuples over D will unlikely give us the smallest counterexample. The reason is that removing a tuple from a group may change the aggregate result (e.g., of SUM); a witness that tries to preserve the original aggregate result may not be able to remove any tuple. Hence, instead of finding minimal witnesses for result tuples over D, we want a counterexample D' that still shows some difference between $Q_1(D')$ and $Q_2(D')$, but the result values do not need to the same as those in $Q_1(D)$ or $Q_2(D)$.

A second challenge requires further modifying the definition of SCP, because there are cases when no counterexamples are small in size even if they do not need to preserve aggregate values. For example, suppose both Q_1 and Q_2 return (different) tuples from groups that pass a HAVING condition stipulating that the group size must be more than a million. Any counterexample would necessarily have to produce a group containing more than a million tuples. Nonetheless, we show in [7] how to adapt the problem definition by allowing the queries to be *parameterized*: e.g., the query constant "one million" in the example earlier would be replaced with a parameter λ. Then, we allow a counterexample to show difference between Q_1 and Q_2 for *some* setting of the parameter λ, e.g., a small group size threshold that does not require a large database instance to achieve. For details, see [7].

3 TECHNIQUES AND IMPLEMENTATION

Capturing Provenance. The smallest witness of a tuple t w.r.t. a query Q and a database instance D can be obtained with *how-provenance* or *lineage* [6]. The how-provenance of a result tuple $t \in Q(D)$, denoted by $Prv_{Q,D}(t)$ (or $Prv(t)$ when the context is clear), is a formula involving Boolean variables annotating tuples in the input relations, which encodes how t is derived from input tuples in D. These Boolean variables indicate whether the corresponding tuples are present in the input, and $Prv(t)$ correctly computes whether tuple t is in the query result given the setting of these Boolean variables. Therefore, one can map the the problem of finding the smallest witness into the *min-ones satisfiability problem*: find a satisfying model for $Prv(t)$ with the least number of variables set to true; the true variables in this model would give us the tuples in a smallest witness.

We implemented RATest based on RA(radb) [7], a relational algebra (extended with group-by and aggregation) interpreter that works by translating relational algebra queries into SQL (using WITH to build up complex queries one relational operator at a time). Taking two queries as input, RATest uses RA(radb) to translate them into SQL, but further rewrites the SQL queries to compute provenance. Specifically, we add an extra string-valued prv column for all input and intermediate result relations to store the provenance expression. For each relational operator, we rewrite the SQL fragment generated for this operator with logic to derive output provenance expressions from input ones. All expressions are in the SMT-LIB format [2]. As an example, for the difference operator, following is the rewritten SQL query for $R - S$ (assuming columns A and B):

```
(SELECT R.A, R.B, '(and␣'||R.prv||'␣(not␣'||S.prv||'))'
 FROM R, S WHERE R.A=S.A AND R.B=S.B) UNION
(SELECT A, B, prv FROM R WHERE NOT EXISTS
   (SELECT * FROM S WHERE S.A=R.A AND S.B=R.B));
```

Note that this query computes all "potential" result tuples (that may arise when some input relation tuples are deleted).

For aggregation, we further apply provenance for aggregate queries [1]. We encode result aggregate values as symbolic expressions involving values from input relations. Then, we can express $Q_1(D) \neq Q_2(D)$ symbolically: assert that a group only exists in one of the query results ($Prv_{Q_1,D}(t) \oplus Prv_{Q_2,D}(t)$ is true), or the group exists in both results but the aggregate values differ ($Prv_{Q_1,D}(t) \wedge Prv_{Q_2,D}(t) \wedge Agv_{Q_1,D}(t) \neq Agv_{Q_2,D}(t)$, where Agv denotes the symbolic expression computing a given group's aggregate value). This approach allows RATest to handle *SPJUDA* (Select-Project-Join-Union-Difference-Aggregate) queries where there is no further grouping using aggregate values and no difference after aggregation within Q_1 and Q_2.

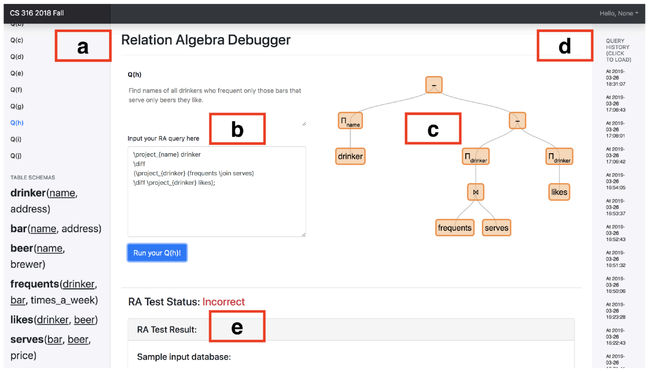
Finding Counterexample with a Solver. Using the provenance information obtained by executing the rewritten SQL queries, we can then take a tuple in the symmetric difference of $Q_1(D)$ and $Q_2(D)$ and formulate an *SMT* (*satisfiability modulo theories*) problem to find a smallest witness. RATest uses the Z3 SMT Solver [5], with the objective function minimizing the number of variables set to true. The satisfying model returned by the solver represents a counterexample.

Example 2. *Consider again Example 1.* ⟨Mary, CS⟩ *and* ⟨Jesse, CS⟩ *are returned by* Q *but not by* Q_0*, and their how-provenance w.r.t.* $Q - Q_0$ *and* D *can be computed, e.g.,*

$$
\begin{aligned}
&Prv_{Q-Q_0,D}(\langle \texttt{Jesse}, \texttt{CS} \rangle) \\
&= Prv_{Q,D}(\langle \texttt{Jesse}, \texttt{CS} \rangle) \wedge \neg Prv_{Q_0,D}(\langle \texttt{Jesse}, \texttt{CS} \rangle) \\
&= \big(t_3(t_9 + t_{10} + t_{11})\big)\ \overline{t_3(t_9 + t_{10} + t_{11})\ \overline{t_3(t_9 t_{10} + t_9 t_{11} + t_{10} t_{11})}} \\
&= t_3 t_9 t_{10} + t_3 t_9 t_{11} + t_3 t_{10} t_{11},
\end{aligned}
$$

where it is clear that any one of $t_3 t_9 t_{10}$, $t_3 t_9 t_{11}$, *and* $t_3 t_{10} t_{11}$ *is a smallest witness for* ⟨Jesse, CS⟩. *By asserting this formula to be true, we can use the SMT solver to find such a witness.*

A basic approach would be to call the solver on each tuple in the symmetric difference of $Q_1(D)$ and $Q_2(D)$, and then



Relation Algebra Debugger

Q(h)

Find names of all drinkers who frequent only those bars that serve only beers they like.

Input your RA query here

```
\project_{name} drinker
\diff
(\project_{drinker} (frequents \join serves)
\diff \project_{drinker} likes);
```

TABLE SCHEMAS

drinker(name, address)

bar(name, address)

beer(name, brewer)

frequents(drinker, bar, times_a_week)

likes(drinker, beer)

serves(bar, beer, price)

RA Test Status: Incorrect

RA Test Result:

Sample input database:

QUERY HISTORY (CLICK TO LOAD)

At 2019-03-26 18:31:07

At 2019-03-26 17:08:43

At 2019-03-26 17:08:01

At 2019-03-26 17:06:42

At 2019-03-26 16:54:05

At 2019-03-26 16:53:37

At 2019-03-26 16:52:43

At 2019-03-26 16:51:32

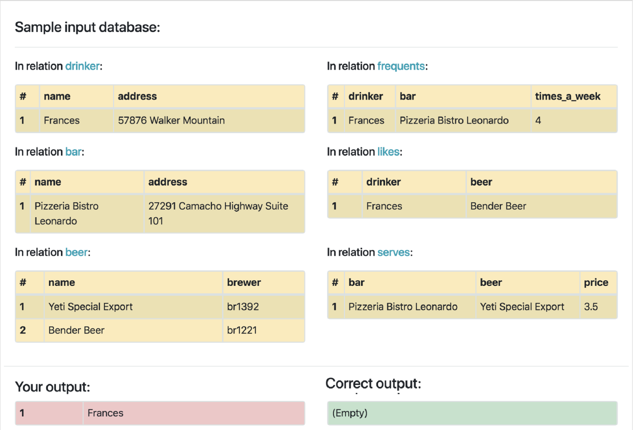
At 2019-03-26 16:50:06

At 2019-03-26 16:23:28

At 2019-03-26 16:22:43

At 2019-03-26

Figure 2: Overview of RATest interface.



Sample input database:

In relation drinker:

#	name	address
1	Frances	57876 Walker Mountain

In relation frequents:

#	drinker	bar	times_a_week
1	Frances	Pizzeria Bistro Leonardo	4

In relation bar:

#	name	address
1	Pizzeria Bistro Leonardo	27291 Camacho Highway Suite 101

In relation likes:

#	drinker	beer
1	Frances	Bender Beer

In relation beer:

#	name	brewer
1	Yeti Special Export	br1392
2	Bender Beer	br1221

In relation serves:

#	bar	beer	price
1	Pizzeria Bistro Leonardo	Yeti Special Export	3.5

Your output:

1	Frances

Correct output:

(Empty)

Figure 3: Counterexample for the query in Figure 2.

pick the overall smallest counterexample. However, computing provenance on all result tuples incurs high overhead. Instead, we can pick one tuple t in the difference, and compute provenance just for t, by adding a selection using t's value on top of the difference query. Pushing this selection down as much as possible through the difference query drastically reduces the cost of provenance computation. Although this approach may not give us the smallest counterexample, it can always find a counterexample, and in practice it works well: it can reduce the running time significantly (up to 42× in our experiments) while still finding counterexamples that are almost as small as the smallest possible.

Note that since the counterexample contains a subset of D's tuples, it trivially satisfies all keys, functional dependencies, and `NOT NULL` constraints. However, referential constraints need explicit enforcement. We capture such constraints using Boolean formulas (involving Boolean variables corresponding to D's tuples), and additionally assert these constraints when invoking the solver.

4 DEMONSTRATION

Figure 2 shows the web interface of RATest. The demonstration goes through a use case of RATest in an educational setting, where students are asked to write relational algebra queries to answer questions against a database about bars, beers, drinkers, and their relationships (bars *serve* beers; drinkers *like* beers and *frequent* bars); see bottom left of Figure 2. Each table contains hundreds to thousands of tuples, and the whole database contains 100,000 tuples.

As an example, consider one of the hardest problems in the assignment: "Find all drinkers who frequent only those bars that serve only beers they like." The solution requires non-trivial uses of joins and differences:

$$\pi_{name} drinker - \pi_{drinker}(\pi_{drinker, beer}(frequents \bowtie serves) - likes).$$

Now consider a student query

$$\pi_{name} drinker \ - \ (\pi_{drinker}(frequents \bowtie serves) - \pi_{drinker} likes),$$

where the second input to the first difference operator is incorrect—this input would find "drinkers who frequent some bars and like no beers." RATest shows a small counterexample explaining this mistake (Figure 3): drinker "Frances" likes "Bender Beer" and frequents "Pizzeria Bistro Leonardo," where only another beer "Yeti Special Export" is served. Thus, "Frances" should not be in the answer.

The user can revise the query and get further feedback that may reveal more bugs; a history feature allows convenient access to queries attempted recently. We also walk through the backend of RATest for those interested.

Overall, this demonstration shows how RATest uses small counterexamples to help users understand why their queries are wrong. We are also releasing RATest to the public to encourage adoption by database courses at other universities.

ACKNOWLEDGMENTS

This work is supported in part by NSF Awards IIS-1408846, IIS-1552538, IIS-1703431, IIS-1718398, IIS-1814493, and by NIH award 1R01EB025021-01.

REFERENCES

[1] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for aggregate queries. In *PODS*. 153–164.

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, Vol. 13. 14.

[3] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *ICDT*. 316–330.

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

[5] 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.

[6] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. 31–40.

[7] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *SIGMOD*.