Explaining Wrong Queries Using Small Examples

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

ABSTRACT

For testing the correctness of SQL queries, a standard practice is to execute the query in question on some test database instance and compare its result with that of the correct query. Given two queries Q_1 and Q_2, we say that a database instance D is a counterexample (for Q_1 and Q_2) if $Q_1(D)$ differs from $Q_2(D)$; such a counterexample can serve as an explanation of why Q_1 and Q_2 are not equivalent. While the test database instance may serve as a counterexample, it may be too large or complex to understand where the inequivalence arises. Therefore, in this paper, given a known counterexample D for Q_1 and Q_2, we aim to find the smallest counterexample $D' \subseteq D$ where $Q_1(D') \neq Q_2(D')$. The problem in general is NP-hard. Drawing techniques from provenance and constraint solving, we develop a suite of algorithms for finding small counterexamples for different classes of queries, including those involving negation and aggregation. We evaluate the effectiveness and scalability of our algorithms on student queries from an undergraduate database course, and on queries from the TPC-H benchmark. We also report a user study from the course where we deployed our tool to help students with an assignment on relational algebra.

ACM Reference Format:
Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong 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, 18 pages. https://doi.org/10.1145/3299869.3319866

1 INTRODUCTION

Correctness of database queries is often tested by evaluating the queries with respect to a *reference query* and a *test database instance.* A primary application is in teaching students how to write SQL and auto-grading their queries. Typically, we have a test database instance D and a correct query Q_1. The correctness of a student query Q_2 can be checked by testing whether $Q_1(D) = Q_2(D)$. Assuming that Q_2 is at least syntactically correct and its output schema is compatible with that of Q_2 (which can be easily verified), if Q_2 does not solve the intended problem, then there will be at least one tuple in $Q_1(D)$ but not in $Q_2(D)$, or in $Q_2(D)$ but not in $Q_1(D)$. Another application scenario is when people rewrite complex SQL queries to obtain better performance. One approach for checking the correctness of complex rewritten queries is regression testing: execute the rewritten query Q_2 on test instances to make sure that Q_2 returns the same results as the original query Q_1. Finding an answer tuple differentiating two queries and providing an *explanation* for its existence helps students or developers understand the error and fix their queries.

In both applications above, if the test database D is large—either because it is a large real data set or it is synthesized to be large enough to test scalability or ensure coverage of numerous corner cases—it would take much effort to understand where the inequivalence of two queries came from. Suppose we teach a database course using DBLP data [30] for an assignment on SQL or relational algebra. The database has more than 5 million entries, and giving this entire database (or the full query results) to students as a counterexample to their query is not very effective. In practice, the mistakes in most of the queries may be explained by only a small number of tuples, which is much more useful for debugging.

Of course, one could generate a completely different counterexample D' altogether, but using the test database instance D to help generate a counterexample has some distinct advantages. First, it helps to preserve the same context for users by using the same data values and relationships. Second, knowing that the original instance D is already a counterexample can help create the counterexample D' more efficiently. This motivates the problem we study in this paper: *given a database instance D, a reference query Q_1, and a test query Q_2 such that $Q_1(D) \neq Q_2(D)$, find a counterexample as a subinstance $D' \subseteq D$ such that $Q_1(D') \neq Q_2(D')$ and the size of D' is minimized.* We illustrate the setting with an example.

Example 1. *Consider relations* `Student(name, major)` *and* `Registration(name, course, dept, grade)` *storing information about students and course registrations. In a database course, suppose the instructor asked the students to write a SQL*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
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/10.1145/3299869.3319866

name	major	
Mary	*CS*	t_1
John	*ECON*	t_2
Jesse	*CS*	t_3

(a) Table Student S

name	course	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) Table Registration R

Figure 1: Toy database instance for Example 1. Identifiers are shown for all tuples.

name	major	
John	*ECON*	r_1

(a) Result of Q_1

name	major	
Mary	*CS*	r_2
John	*ECON*	r_3
Jesse	*CS*	r_4

(b) Result of Q_2

Figure 2: Results of Q_1, Q_2 in Example 1.

query to find students who registered for exactly one Computer Science (CS) course. The test instance is given in Figure 1. The following query Q_1 solves this problem correctly:

```
SELECT s.name,s.major    -- Q1
FROM Student s, Registration r
WHERE s.name = r.name AND r.dept = 'CS'
EXCEPT
SELECT s.name,s.major
FROM Student s, Registration r1, Registration r2
WHERE s.name = r1.name AND s.name = r2.name AND
r1.course <> r2.course AND r1.dept = 'CS' AND r2.dept = 'CS'
```

However, one student wrote Q_2, which actually finds students who registered for one or more CS courses.

```
SELECT s.name,s.major   -- Q2
FROM Student s, Registration r
WHERE s.name = r.name AND r.dept = 'CS'
```

Figure 2 shows the results of Q_1 and Q_2. The tuples r_2 and r_4 are in the output of Q_2 but not in the output of Q_1. To convince the student that his query is wrong, the instructor can provide the full contents of S, R as a counterexample comprising 11 tuples. However, a smaller and better counterexample can simply contain three tuples (e.g., $t_1 t_4 t_5$) to illustrate the inequivalence of Q_1 and Q_2. The benefit can be much larger if we consider a real enrollment database from a university, whereas the size of the counterexample can remain small.

Prior work in the database community mainly focused on the theoretical study of decidability [13, 35] or generating a comprehensive set of test databases to "kill" as many erroneous queries as possible [10], but does not pay much attention to explaining why two queries are inequivalent. There are recent systems that aim to generate counterexamples for SQL queries. Cosette developed by Chu et al. [12] used formal methods to encode SQL queries as logic formulas to generate a counterexample proving that two SQL queries are inequivalent. XData by Chandra et al. [10] generates test data using mutation techniques. However, counterexamples generated by such systems can lead to arbitrary values, which may not be meaningful to the user. Our approach instead ensures that the user sees familiar values and relationships already present in the test database instances. Moreover, our approach focuses users on one issue with their query at a time, while XData gives instances that test multiple issues.

Specifically, this paper makes the following contributions.

- We formally define the smallest counterexample problem, and connect it to *data provenance* with the definition of the smallest witness problem.
- We give results (NP-hardness proofs and poly-time algorithms) in terms of both data and combined complexity for different subclasses of SPJUD queries.
- We give practical algorithms for SPJUD queries using SAT and SMT solvers, and discuss a suite of optimizations to improve efficiency.
- For aggregate queries, we illustrate the challenges and propose methods to address them, which includes applying *provenance for aggregate queries* [2], adapting the problem definition using the idea of "parameterizing" the queries, and heuristics to improve efficiency.
- We have implemented an end-to-end system called RATest, which has seen live deployment and will be demonstrated at this conference [33].
- We evaluate the effectiveness and scalability of our approach on student queries from an undergraduate course, and on queries from the TPC-H benchmark.
- We provide a case study in a large undergraduate database course at Duke University, where students use RATest to debug their queries in a homework. Quantitative analysis of usage statistics and homework scores show that use of RATest improved student performance; anonymous survey of the students also indicates that they found RATest helpful to their learning.

2 SPJUD QUERIES

In this section, we consider the class of non-aggregate queries with Select (S)-Project (P)-Join (J)-Union (U)-Difference (D) operations expressed as relational algebra (RA) expressions; queries with aggregates are considered in Section 3. We will use RA form and SQL form of queries interchangeably. A subset of these operators using abbreviations will denote the corresponding subclass of such queries; e.g., PJ queries will denote queries involving only projection and join operations. First, Section 2.1 gives a formal problem definition and covers some preliminaries. Then, Section 2.2 presents complexity results for different subclasses of SPJUD queries. Finally, Section 2.3 discusses practical algorithms and optimizations.

2.1 Problem Definition and Preliminaries

For a database instance D (involving one or more relations) and a query Q, $Q(D)$ denotes the result of Q on D. Let Γ denote a set of integrity constraints on the schema of the

database instance D. We consider the following standard integrity constraints: keys, foreign keys, not null, and functional dependencies. If D satisfies Γ, we write $D \models \Gamma$. We use $|D|$ to denote the total number of tuples in D.

We will use unique identifiers to refer to the tuples in the database and query answers. In our example tables, they are written in the right-most column (see Figures 1 and 2), e.g., in Figure 1, t_1 refers to the tuple Student$(Mary, CS)$.

2.1.1 Smallest Counterexample Problem. Consider two queries Q_1 and Q_2 such that $Q_1(D) \neq Q_2(D)$ on a database instance D, where $D \models \Gamma$ for a given set of constraints Γ. In other words, D explains why Q_1 and Q_2 are not equivalent. Based on D, we want to find a small counterexample $D' \subseteq D$ that also explains the inequivalence of Q_1, Q_2.

Definition 1 (Counterexample and the Smallest Counterexample Problem). *Given a database schema with a set of integrity constraints Γ and two queries Q_1 and Q_2, we say that a database instance D is a* counterexample *for Q_1 and Q_2 conforming to Γ if $D \models \Gamma$ and $Q_1(D) \neq Q_2(D)$.*

Given a counterexample D for Q_1 and Q_2 conforming to Γ, the goal of the smallest counterexample problem *$\mathsf{SCP}(D, \Gamma, Q_1, Q_2)$ is to find a counterexample $D' \subseteq D$ for Q_1 and Q_2 conforming to Γ such that the total number of tuples in D' is minimized (i.e., for all counterexamples $D'' \subseteq D$, $|D''| \geq |D'|$).*

In the above definition, we assume that the query results are *union-compatible* (i.e., $Q_1(D)$ and $Q_2(D)$ have the same schema), which is easy to check syntactically; otherwise the difference in their schema provides an explanation of their inequivalence. From now on, where it is clear from the context, we will implicitly assume that $D' \subseteq D$ discussed as a counterexample conforms to the given constraints Γ; we will discuss how constraints are handled in Section 2.3.4.

Example 2. *In Example 1, the given test instance in Figure 1 is a counterexample for Q_1 and Q_2. However, some subinstances are also counterexamples. Among these, two smallest counterexamples can be formed with t_1 in S and t_4t_5 in R, or with t_3 in S and t_9t_{10} in R. (Two other subinstances that are smallest counterexamples can be formed by varying the two courses of Jesse, but no counterexample has fewer than 3 tuples.)*

Our goal is to explain the query inequivalence to users by showing the smallest counterexample over which the two queries return different results. Even in our running example with a toy database instance, this reduced the number of tuples from 11 to only 3, whereas the benefit is likely to be much more for test database instances in practice as observed in our experiments. The brute-force method to find the smallest counterexample is to enumerate all subinstances of D, and search for the smallest subinstance D' where $Q_1(D')$ and $Q_2(D')$ differ. However, enumerating all possible subinstances is inefficient and it does not utilize the fact that D itself is a counterexample. Therefore, to solve this problem more efficiently, we relate it to the concepts of *witnesses* and *data provenance* as discussed below.

2.1.2 Smallest Witness Problem. Buneman et al. [9] proposed the concept of *witnesses* to capture *why-provenance* of a query answer. Intuitively, a witness is a collection of input tuples that provides a *proof* for a given result tuple. Formally, given a database instance D, a query Q, and a tuple $t \in Q(D)$, a witness for t w.r.t. Q and D is a subinstance $D' \subseteq D$ where $t \in Q(D')$. For instance, in Example 1, $\{t_1, t_4\}$, $\{t_1, t_5\}$, and $\{t_1, t_4, t_5\}$ are three witnesses of the result tuple r_2 w.r.t. Q_2 and D. We use $\mathcal{W}(Q, D, t)$ to denote the set of all witnesses for $t \in Q(D)$ w.r.t. Q and D. Since we also consider non-monotone queries, we extend the definition of witness with the concept of *potential answers*. We give the definition below, and will discuss more details later.

Definition 2 (Potential Answers). *Given a query Q and a database instance D, we call a tuple t a* potential answer *w.r.t. Q and D if there exists $D' \subseteq D$ such that $t \in Q(D')$. Let $\mathcal{A}(Q, D)$ denote the set of all potential answers w.r.t. Q and D.*

Intuitively, members of $\mathcal{A}(Q, D)$ can be obtained by deleting zero or more tuples from D and evaluating Q. Obviously, all tuples in $Q(D)$ are potential answers.

For instance, in Example 1, only $(John, ECON)$ is in the result of Q_1. However, other tuples can also be potential answers. In particular, if we remove some registration records from R, then (Mary, CS) and (Jesse, CS) may appear in the query result of Q_1 over the modified database, so they are potential answers.

The following proposition states that the number of potential answers is polynomial in *data complexity* [45] (when the query is fixed). The proof is by induction on the height of the operator tree for the query (we defer the proof to the full version of this paper [32] because of space constraints).

Proposition 1. *Given a database instance D, an SPJUD query Q, the number of tuples in $\mathcal{A}(Q, D)$ is polynomial in terms of number of tuples in D.*

Recall that witness is previously defined for a tuple in result of Q over D. With the notion of potential answers, we now extend the domain of witness so that it is also defined for any tuple in $\mathcal{A}(Q, D)$.

A witness may contain many tuples and is sensitive to the query structure. Buneman et al. [9] defined *minimal witness* as a minimal element of $\mathcal{W}(Q, D, t)$; i.e., for a minimal witness $w \in \mathcal{W}(Q, D, t)$, there exist no other witnesses $w' \in \mathcal{W}(Q, D, t)$ such that $w' \subset w$. In Example 1, $\{t_1, t_4\}$ and $\{t_1, t_5\}$ are minimal witnesses of the result tuple r_2 w.r.t.

Q_2 and D, but $\{t_1, t_4, t_5\}$ is not. Note that a witness with the smallest cardinality must be a minimal witness.

How do witnesses relate to counterexamples? It turns out that any counterexample for Q_1 and Q_2 is also a witness for some potential answer w.r.t. either $Q_1 - Q_2$ or $Q_2 - Q_1$, as the following proposition shows more formally.

Proposition 2. *Given a counterexample D for Q_1 and Q_2, any counterexample $D' \subseteq D$ for Q_1 and Q_2 must be in $\mathcal{W}(Q_1 - Q_2, D, t) \cup \mathcal{W}(Q_2 - Q_1, D, t)$ for some $t \in \mathcal{A}(Q_1 - Q_2, D) \cup \mathcal{A}(Q_2 - Q_1, D)$.*

Proof. Since D' is a counterexample, there exists a tuple t such that $t \in (Q_1 - Q_2)(D')$ or $t \in (Q_2 - Q_1)(D')$. Because $D' \subseteq D$, by definition of $\mathcal{A}$, $t \in \mathcal{A}(Q_1 - Q_2, D) \cup \mathcal{A}(Q_2 - Q_1, D)$. Furthermore, D' is a witness for t w.r.t. $Q_1 - Q_2$ and D, or a witness for t w.r.t. $Q_2 - Q_1$ and D. □

Definition 3 (Smallest Witness Problem). *Given a database instance D, two union-compatible queries Q_1 and Q_2 such that $Q_1(D) \neq Q_2(D)$, and a tuple t such that $t \in \mathcal{A}(Q_1 - Q_2, D)$ or $t \in \mathcal{A}(Q_2 - Q_1, D)$, the goal of the* smallest witness problem SWP(D, Q_1, Q_2, t) *is to find a witness $w \in \mathcal{W}(Q_1 - Q_2, D, t) \cup \mathcal{W}(Q_2 - Q_1, D, t)$ such that the total number of tuples in w is minimized.*

Using Propositions 1 and 2, we can reduce the smallest counterexample problem SCP(D, Q_1, Q_2) to the smallest witness problem SWP(D, Q_1, Q_2, t) in polynomial time by enumerating every potential answer t w.r.t. $Q_1 - Q_2$ and D or w.r.t. $Q_2 - Q_1$ and D, solving SWP(D, Q_1, Q_2, t), and finding the globally minimum witness across all such t's; i.e.,

$$\mathsf{SCP}(D, Q_1, Q_2) = \min_{t \in \mathcal{A}(Q_1 - Q_2, D) \cup \mathcal{A}(Q_2 - Q_1, D)} \mathsf{SWP}(D, Q_1, Q_2, t).$$

From now on, without loss of generality, we will assume that in SCP(D, Q_1, Q_2), there exists a tuple $t \in Q_1(D)$ but $t \notin Q_2(D)$. In the rest of the paper, we will mainly focus on the smallest witness problem SWP(D, Q_1, Q_2, t) for such a tuple. We further discuss the connection between SCP and SWP in Section 2.3.

Discussion. Note that we have to consider Q_1 and Q_2 jointly in general when finding counterexamples and witnesses. For instance, suppose $t \in Q_1(D) - Q_2(D)$, and we want to find a smallest subset D' of D such that $t \in Q_1(D') - Q_2(D')$. Answering "why t is in $Q_1(D)$" is enough only when Q_2 is monotone. When Q_2 is non-monotone, a witness $w \in \mathcal{W}(Q_1, D, t)$ may happen to make t appear in the result of $Q_2(w)$, failing to differentiate Q_1 and Q_2.[1] Similarly, answering both "why t is in $Q_1(D)$" and "why t is not in $Q_2(D)$" may also fail—if we find a witness $w_1 \in \mathcal{W}(Q_1, D, t)$ and a small subinstance w_2 of D with the help of a *why-not* approach [21–23, 29], such that $t \in Q_1(w_1)$ and $t \notin Q_2(w_2)$, it is possible that $t \notin (Q_1 - Q_2)(w_1 \cup w_2)$: *first*, t could be missing from $Q_1(w_1 \cup w_2)$ if Q_1 is non-monotone; *second*, t could be in the result of $Q_2(w_1 \cup w_2)$ even if $t \notin Q_2(w_2)$. Therefore, we have to consider $Q_1 - Q_2$ or $Q_2 - Q_1$ as a whole.

[1]In contrast, if Q_2 is monotone, t cannot be in $Q_2(D')$ for any $D' \subseteq D$ because we know $t \notin Q_2(D)$ to begin with.

Query Class of Q_1, Q_2	Data Complexity	Combined Complexity
SJ	**P** (Thm. 1)	**P** (Thm. 1)
SPU	**P** (Thm. 2)	**P** (Thm. 2)
PJ	**P** (Thm. 6)	**NP-hard** (Thm. 3)
JU	**P** (Thm. 6)	**NP-hard** (Thm. 4)
JU*	**P** (Thm. 5)	**P** (Thm. 5)
SPJUD*	**P** (Thm. 7)	**NP-hard** if falls into class PJ or JU
PJD	**NP-hard** (Thm. 8)	**NP-hard** (Thm. 8)

Table 1: Complexity results on SWP. All theorems and proofs appear in the appendix.

2.2 Complexity for SPJUD Queries

Table 1 summarizes the complexity of the smallest witness problem (SWP) for any subclass of SPJUD queries. In terms of complexity, we consider *data complexity* (fixed query size), *query complexity* (fixed data size), and *combined complexity* (in terms of both data and query size) [45]. Thus polynomial combined complexity indicates polynomial data complexity.

In Table 1, the class JU* has the restriction that all unions appear after all joins. The class SPJUD* is defined as: $Q \to q^+ | Q - Q$, where q^+ is a SPJU query. Proofs are given in Appendix A. For queries involving PJ, in general even the query evaluation problem is NP-hard in query complexity. It is the same for queries involving JU, however, the problem is in poly-time for the subclass JU*, because we can directly look into the join-only parts of a JU* query. For general SPJU queries, the problem has poly-time data complexity, and thus we can provide a poly-time algorithm for SPJUD* queries in data complexity. What is noteworthy is that for the class of queries involving projection, join, and difference without any restrictions, it is already NP-hard in data complexity to find the smallest witness for a result tuple; and the result holds even when the queries are of bounded sizes and the database instance only contains two relations. While in the complexity results, we assume both Q_1, Q_2 belong to the same query class, if $t \in Q_1(D) \setminus Q_2(D)$, for all monotone cases the exact class of Q_2 does not matter as long as it is monotone. We can show that SCP for PJD queries is also NP-hard in data complexity by a simple reduction from SWP.

2.3 Methods for SPJUD Queries

The smallest witness problem is in general NP-hard even for queries of bounded size, and the poly-time algorithms in Table 1 are not efficient for practical purposes. To address these challenges, we introduce a constraint-based approach

to the smallest witness problem. We map the problem into the *min-ones satisfiability problem* [28] by tracking the Boolean provenance of potential answers. The min-ones satisfiability problem is an extension of the classic *Satisfiability (SAT)* problem: *given a Boolean formula ϕ, it checks whether ϕ is satisfiable with at most k variables set to true.* This problem can be solved by either using a *SAT solver* (e.g., MiniSAT [43], CaDiCaL [7]), or an *SMT Solver* (e.g., CVC4 [4]and Z3 [15]). *Satisfiability Modulo Theories (SMT)* is a form of *constraint satisfaction problem.* It refers to the problem of determining whether a first-order formula is satisfiable w.r.t. other background first-order formulas, and is a generalization of the SAT problem [6]. SAT and SMT problems are known to be NP-hard with respect to the number of clauses, constraints, and undetermined variables. However, there is a variety of solvers that work very well in practice, and with these solvers we can find a small solution to a SWP instance. The rest of this subsection will describe how to encode how-provenance, and then use a state-of-the-art solver to find the smallest witness for a potential answer. The implementation details will be discussed in Section 4.

2.3.1 Boolean How-Provenance. In order to compute the smallest witness efficiently for general SPJUD queries, we use the concept of *how-provenance* or *lineage* [1, 19] by *provenance semirings.* How-provenance encodes how a given result tuple is derived from the given input tuples using a Boolean expression, and its first use can be traced back to Imilienski and Lipski [24] who used it to describe incomplete databases or c-tables. The computation of how-provenance of a tuple $t \in \mathcal{A}(Q, D)$, denoted by $\texttt{Prv}_{Q,D}(t)$ or $\texttt{Prv}(t)$ when clear from the context, is well known and intuitive: tuples in the given input relations are annotated with unique identifiers (as shown in the right-most columns in Figure 1). As the query Q executes, for selections or duplicate-preserving projections, the annotations remain the same; for joint usages of sub-expressions (joins), their annotations are combined with conjunction ($\wedge$ or $\cdot$), and for alternative usages of sub-expressions (de-duplicates or unions), the annotations are combined with disjunction ($\vee$ or $+$). For simplicity, we use $+$ for disjunction, and omit symbols for conjunction. For instance, in Example 1, in $Q_2(D)$,

$$\texttt{Prv}_{Q_2,D}(r_2) = t_1 t_4 + t_1 t_5 = t_1(t_4 + t_5) = \phi_1(\text{say}) \quad (1)$$

For set difference operation, the extension of provenance with potential answers is important. Consider $Q = Q_1 - Q_2$ where all tuples in $\mathcal{A}(Q_1, D)$ and $\mathcal{A}(Q_2, D)$ are annotated with how-provenance. For a tuple t to appear in $\mathcal{A}(Q, D)$, it must appear in $\mathcal{A}(Q_1, D)$. Suppose $\texttt{Prv}_{Q_1,D}(t) = \phi$. If t does not appear in $\mathcal{A}(Q_2, D)$, $\texttt{Prv}_{Q,D}(t) = \phi$. If t does appear in $\mathcal{A}(Q_2, D)$ with $\texttt{Prv}_{Q_2,D}(t) = \psi$, then $\texttt{Prv}_{Q,D}(t) = \phi \cdot \overline{\psi}$, where $\overline{\psi} = \neg\psi$ denotes the negation of the Boolean expression ψ. The tuple t appears in the final result of $Q(D)$ if t appears in $\mathcal{A}(Q_1, D)$ but not in $\mathcal{A}(Q_2, D)$, or if t appears in both of them and $\texttt{Prv}_{Q,D}(t)$ is true. The removal of any tuple from D will make its Boolean variable set to false and thus may affect the value of $\texttt{Prv}_{Q,D}(t)$. We consider all potential answers w.r.t. Q and D when computing the how-provenance for $Q(D)$.

Example 2.1. In Example 1, consider the following RA expressions for Q_2 and Q_1, using abbreviations S and R for Students and Registration, where $\bowtie$ denotes natural join.

$$Q_2 = \pi_{\texttt{name,major}} \sigma_{\texttt{dept='CS'}}(\texttt{S} \bowtie \texttt{R}) \quad (2)$$

Suppose $Q_3 = \pi_{\texttt{name,major}} \sigma_\eta(\texttt{S} \bowtie \texttt{R r1} \bowtie \texttt{R r2})$, where η denotes the selection condition: $r1.\texttt{dept} = \texttt{'CS'} \wedge r2.\texttt{dept} = \texttt{'CS'} \wedge r1.\texttt{course} \neq r2.\texttt{course}$. Then $Q_1 = Q_2 - Q_3$. Consider the result tuple r_2 = (Mary, CS), which is in $(Q_2 - Q_1)(D)$ (Figure 2). The provenance of r_2 = (Mary, CS) in $Q_2(D)$ is given in Equation (1). It does not appear in $Q_1(D)$ since it appears in both Q_2, Q_3 in (2). For Q_3, $\texttt{Prv}_{Q_3,D}(r_2) = t_1 t_4 t_5 = \phi_2$(say). Hence, $\texttt{Prv}_{Q_1,D}(r_2) = \phi_1 \cdot \overline{\phi_2}$, and $\texttt{Prv}_{Q_2-Q_1,D}(r_2)$ $= \phi_1 \cdot \overline{[\phi_1 \cdot \overline{\phi_2}]} = \phi_1 \cdot [\overline{\phi_1} + \phi_2] = \phi_1 \cdot \phi_2 = (t_1(t_4 + t_5)) \cdot (t_1 t_4 t_5)$ $= t_1 t_4 t_5$. In other words, the tuple (Mary, CS) can distinguish the queries Q_1, Q_2 in a small witness with t_1 in S and $t_4 t_5$ in R, which solves both SWP and SCP problems.

For the above example, the smallest witness or the smallest counterexample could be found by inspection, since Q_1, Q_2 are similar. For arbitrary and more complex queries, how-provenance gives a systematic approach to find a small witness as we will discuss in the following two sections.

2.3.2 Passing How-Provenance to a Solver. Since $Prv(t)$ is composed of a combination of Boolean variables annotating tuples in the input relations, a Boolean variable is true iff the corresponding tuple is present in the input relation. Then an instance of the smallest witness problem is mapped to an instance of the min-ones satisfiability problem: find a satisfying model to $Prv(t)$ with least number of variables set to true, and the variables set to true in the satisfying model indicate tuples in the smallest witness.

The pseudocode of the algorithm to solve SWP and then SCP can be found in the full version [32]. Briefly, one approach to solving SWP is by repeated invocation of a SAT solver. Since the solver will return an arbitrary satisfying model, to get the minimum model we need to ask the solver to return a different model every time we rerun it. We set a maximum number of runs to limit the running time, and the algorithm stops when there is no more satisfying models or it has reached the maximum number of runs. It may not find the minimum model when it stops, but it is likely to find a small one if given enough time.

To solve SCP, recall from Section 2.1 that it suffices to consider SWP for all potential answers in $Q_1 - Q_2$ and $Q_2 - Q_1$.

Therefore, a basic algorithm for SCP, which we call SCP-ALL, will compute the provenance for $Q_1 - Q_2$ and $Q_2 - Q_1$, identifying all potential answers and their provenance in the process. Then, for each potential answer, we solve SWP as described above. The overall smallest witness will be chosen.

2.3.3 Improving the Basic Approach. The basic approach above has two limitations: 1) for SWP, it cannot find the smallest witness until it searches all possible models that satisfy Prv(t); 2) SCP-ALL computes provenance for every potential answer, resulting in high overhead. Therefore, we propose two improvements.

The first improvement is to pick only one tuple t from $Q_1(D)\backslash Q_2(D)$, and capture only the provenance of t by adding a selection operator to select tuple t on top of the query tree of $Q_1 - Q_2$. During provenance computation, which we implemented by rewriting SQL queries (see Section 4 for more details), the SQL optimizer is likely to push down the additional selection operator, thereby reducing unnecessary intermediate computation and accelerating provenance computation. We call this strategy SCP-SEL for "selective" provenance, which solves only one instance of SWP. Of course, if $Q_1(D) \subseteq Q_2(D)$, we would then consider $Q_2(D) \setminus Q_1(D)$.

The second improvement is to treat SWP as an optimization problem instead of finding different models with a SAT solver. Integer linear programming solvers cannot be applied because transforming how-provenance into linear constraints can be exponential. To solve this problem, we use *optimizing SMT solvers* that are now available with recent advances in the programming languages and verification research community [8, 31]. Given a formula ϕ and an objective function $\mathcal{F}$, an optimizing SMT solver finds a satisfying assignment of ϕ that maximizes or minimizes the value of $\mathcal{F}$. In our case, we encode Prv(t) as the constraint of the optimizing SMT solver, set the number of true variables as the objective function, and solve for the optimal model. Our SMT formulation includes only Boolean variables, so we encode the number of true variables by first converting the variables into 0 or 1 and then summing them up. The pseudocode of this improved algorithm can be found in [32].

As an example, the following listing illustrates how we encode the provenance in the SMT-LIB standard format [5] as the input to an SMT solver, which returns a solution to SWP(D, Q_1, Q_2, (Jesse,CS)) in Example 1:

```
1  (declare-const t1 Bool)
2  ...
3  (declare-const t11 Bool)
4  (define-fun b2i ((x Bool)) Int (ite x 1 0))
5  (assert (and (or t4 t5) (not (and (or (and t1 t4) (and t1
       t5)) (not (and t1 (and t4 t5)))))))
6  (minimize (+ (b2i t1) (b2i t2) ... (b2i t11)))
```

Lines 1-3 define Boolean variables for each tuple. Line 4 defines function *b2i* that converts Boolean variables into 0 and 1. Line 5 adds the how-provenance as an SMT constraint. Then with function *b2i* we take the sum of 0-1 variables to get the total number of variables set to true, and then set minimizing this sum as the objective function (Line 6).

2.3.4 Handling Database Constraints. Since we output a subinstance of the input database instance as the witness, database constraints like keys, not null, and functional dependencies are trivially satisfied if the input instance is valid. On the other hand, foreign key constraints can be naturally represented as Boolean formulas like provenance expressions. For instance, in our running example in Figure 1, the `name` column in the `Registration` table may refer to the `name` column in the `Student` table. So, if we want to keep any tuple in the `Registration` table, we must also keep the tuple with the same `name` value in the `Student` table. This constraint can be expressed in the $a \Rightarrow b$ form, e.g., $t_1 + \overline{t_4}$, $t_2 + \overline{t_7}$, .., etc., corresponding to the constraint that the tuples in the `Registration` table cannot exist unless the tuple it refers to exists in the `Student` table. Then, for each tuple that appears in the provenance expression added to the SAT or SMT solver, we add its foreign key constraint expression to the solver as a constraint.

3 AGGREGATE QUERIES

So far, we have focused on SPJUD queries. We now extend our discussion to aggregate queries. We make some assumptions on the form of such queries: 1) no aggregate values or NULL are allowed in group-by attributes; 2) selection predicates involving previously aggregated values (i.e., HAVING conditions) involve only numeric comparisons; 3) there is no difference operator above aggregation in either Q_1 or Q_2, (of course, we handle the top-level difference in $Q_1 - Q_2$ and $Q_2 - Q_1$ when Q_1 and Q_2 may involve aggregation).

3.1 Challenges of Aggregate Queries

First, (selective) how-provenance does not work well for SCP for aggregate queries. Recall from Section 2.3 that a practical heuristic we used for SPJUD queries, SCP-SEL, just picks one tuple t from the symmetric difference between Q_1 and Q_2 and focuses on computing how-provenance and finding a witness for t. Unfortunately, this heuristic may not work for an aggregates result tuple t, because its aggregate value generally depends on all member tuples in the input group corresponding to t; removal of any such tuple may change t's aggregate value, so there may not exist any proper subset of tuples in the input group that can serve as a witness for t.

EXAMPLE 3 (CHALLENGE OF PRESERVING AGGREGATE VALUES). *Suppose we have a reference query Q_1 aimed at computing the average grade of students in CS courses, using the two tables in Figure 1:*

```sql
SELECT s.name, AVG(r.grade) as avg_grade
FROM Student s, Registration r
WHERE s.name=r.name AND r.dept='CS'
GROUP BY s.name;
```

Suppose a second query Q_2 forgets the condition `r.dept='CS'`. *These two queries would return the following results:*

$Q_1(D)$

name	*avg_grade*
Mary	*87.5*
John	*90*
Jesse	*90*

$Q_2(D)$

name	*avg_grade*
Mary	*90*
John	*89*
Jesse	*90*

Suppose we pick (Mary, 90) in $Q_2(D)$ to focus on. The witness for this result tuple—which by definition needs to preserve the aggregate value 90—would have to include all of Mary's registration records. In reality, however, to show that Q_1 returns a different result from Q_2 for some *counterexample, one registration record (Mary, 208D, ECON, 88) will suffice: Q_1 would return empty while Q_2 would return (Mary, 88).*

To circumvent the problem, can we resort to considering witnesses for all potential answers instead of just t, as in SCP-ALL? Indeed, in the above example, (Mary, 88) is a potential answer that can be constructed by extending how-provenance to aggregates, essentially through enumeration of all possible combinations of tuples in a group [41]. However, this approach leads to exponential overhead and becomes impractical for even moderate-size groups.

Finally, as soon as we consider further selection involving aggregate values (such as `HAVING`), there can be cases where the size of any counterexample is necessarily large by our formulation of SCP in Definition 1. Consider the following.

Example 4 (Challenge of Inherently Large Counterexamples). *Continuing with Example 3, let us extend both queries to find the average grade of CS courses of students who registered for at least 3 CS courses. Basically, both Q_1 and Q_2 get an additional* `HAVING` *clause. For example, Q_1 becomes:*

```sql
SELECT s.name, AVG(r.grade) as avg_grade
FROM Student s, Registration r
WHERE s.name=r.name AND r.dept='CS'
GROUP BY s.name
HAVING COUNT(r.course) >= 3;
```

Q_1 and Q_2 would return the following results (recall that Q_2 misses the condition `r.dept='CS'`*):*

$Q_1(D)$

name	*avg_grade*
Jesse	*90*

$Q_2(D)$

name	*avg_grade*
Mary	*90*
Jesse	*90*

Because of the `HAVING` *condition, we must include all (3) of Mary's registration records in a counterexample, or else the queries would be indistinguishable because neither would return Mary's group.*

While the group size in the above example is small, it is trivial to construct examples where `HAVING` would force arbitrarily large groups of input tuples to be included in a counterexample. No approach (including painstakingly considering all potential answers) would help because the problem is inherent in the formulation of SCP in Definition 1.

	name	avg_grade	provenance
Q_1	Mary	$val_1 : \mathfrak{avg}(t_4 \otimes 100,\ t_5 \otimes 75)$	$prv_1 : (t_1(t_4 + t_5))\ (\mathfrak{sum}(t_4 \otimes 1, t_5 \otimes 1) \geq 3)$
	John	$val_2 : \mathfrak{avg}(t_7 \otimes 90)$	$prv_2 : (t_2 t_7)\ (t_7 \otimes 1 \geq 3)$
	Jesse	$val_3 : \mathfrak{avg}(t_9 \otimes 95, t_{10} \otimes 90, t_{11} \otimes 85)$	$prv_3 : (t_3\ (t_9 + t_{10} + t_{11}))\ (\mathfrak{sum}(t_9 \otimes 1, t_{10} \otimes 1, t_{11} \otimes 1) \geq 3)$

	name	avg_grade	provenance
Q_2	Mary	$val_4 : \mathfrak{avg}(t_4 \otimes 100, t_5 \otimes 75, t_6 \otimes 95)$	$prv_4 : (t_1(t_4 + t_5 + t_6))\ (\mathfrak{sum}(t_4 \otimes 1, t_5 \otimes 1, t_6 \otimes 1) \geq 3)$
	John	$val_5 : \mathfrak{avg}(t_7 \otimes 90, t_8 \otimes 8)$	$prv_5 : (t_2(t_7 + t_8))\ (\mathfrak{sum}(t_7 \otimes 1, t_8 \otimes 1) \geq 3)$
	Jesse	$val_6 : \mathfrak{avg}(t_9 \otimes 95, t_{10} \otimes 90, t_{11} \otimes 85)$	$prv_6 : (t_3(t_9 + t_{10} + t_{11}))\ (\mathfrak{sum}(t_9 \otimes 1, t_{10} \otimes 1, t_{11} \otimes 1) \geq 3)$

Table 2: Provenance for aggregate queries in Example 4. Note that instead of infix notations for $\mathfrak{avg}$ and $\mathfrak{sum}$ as in [2], we use prefix notations for readability.

3.2 Methods for Aggregate Queries

We first tackle the ineffectiveness of how-provenance, by applying *provenance semirings for aggregate queries* proposed by Amsterdamer et al. [2] and proposing a baseline method called AGG-BASE. We further address the inherent limitation of Definition 1 for handling aggregates by proposing a new definition based on the ideas of parameterizing queries, and showing how AGG-BASE can be adapted. Finally, we present a heuristic method called AGG-IMPR that improves the running time of AGG-BASE for queries involving a single aggregation followed by an optional selection.

3.2.1 Applying Provenance for Aggregates. Following the approach in [2], we encode aggregate values computed by queries as symbolic expressions using abstract arithmetic operations and variables corresponding to tuples in the input relations. Selection conditions involving aggregate values are then encoded as symbolic logical expressions. Table 2 shows the provenance for aggregate queries for Example 4. For instance, $\mathfrak{avg}(t_4 \otimes 100, t_5 \otimes 75)$ represents the `AVG` value of a group containing two tuples t_4 and t_5 in the result of Q_1, and the value of the attribute of tuple t_4 is 100, and for t_5 it is 75 . If t_4 is removed from the input relations, then $t_4 \otimes 100$ will not contribute to the aggregate. Like the how-provenance, $(t_1(t_4 + t_5))\ (\mathfrak{sum}(t_4 \otimes 1, t_5 \otimes 1) \geq 3)$ indicates how the result tuple is derived from the input or intermediate tuples: $t_1(t_4 + t_5)$ means that the group exists iff t_1 exists and one of t_4 and t_5 exists; $\mathfrak{sum}(t_4 \otimes 1, t_5 \otimes 1) \geq 3$ represents the `HAVING` criterion: the `COUNT` (a special case of `SUM`) value should be greater or equal to 3. As with how-provenance, aggregate values and provenance expressions can be computed bottom-up during query evaluation. Unlike how-provenance, there is no longer a need to enumerate all possible aggregate values resulting from subsets of a group of tuples; one single symbolic expression succinctly captures these possibilities.

With this approach, we have a baseline method for finding counterexamples for aggregate queries, which we call AGG-BASE. Given $Q_1(D)$ and $Q_2(D)$ that differ, we pick a group that exists in one result but not the other, or a group that exists in both but the aggregate values differ. In either case,

we can assert a symbolic Boolean expression derived using the aggregate provenance expressions, and let an SMT solver find a satisfying model with minimum number of input tuples included. For example, a counterexample for Q_1 and Q_2 w.r.t. tuple (Mary, 90) can be found by solving the constraint $(prv_4 \oplus prv_1) \vee (val_4 \neq val_1)$. Note that the constraint only insists that aggregate values produced by Q_1 and Q_2 are different; there is no stipulation that this counterexample reproduces the same aggregate value 90.

3.2.2 Parameterizing the Problem. To address the problem that certain queries require large counterexamples by Definition 1, we modify our problem definition by parameterizing the queries, thereby allowing extra degrees of freedom when constructing counterexamples. Specifically, we replace the constants in selection predicates (such as 3 used in the `HAVING` condition of Example 4) by a parameter; a subinstance can be a counterexample as long as it differentiates the two queries for some setting of the parameter, not necessarily the original one.

Definition 4 (Smallest Parameterized Counterexample Problem). *Given two queries Q_1 and Q_2 parameterized by λ, as well as a database instance D and a setting $\lambda = v$, where $Q_1(v, D) \neq Q_2(v, D)$, the smallest parameterized counterexample problem (SPCP) is to find a subinstance D' of D and a parameter setting $\lambda = v'$, such that such that $Q_1(v', D') \neq Q_2(v', D')$, and the total number of tuples in D' is minimized.*

Example 5 (Smallest Parameterized Counterexample). *Continuing with Example 4, we can turn the threshold number of courses (3) in the `HAVING` conditions of both Q_1 and Q_2 into a parameter λ. For example, Q_1 becomes:*

```
SELECT s.name, AVG(r.grade) as avg_grade
FROM Student s, Registration r
WHERE s.name=r.name AND r.dept='CS'
GROUP BY s.name
HAVING COUNT(r.course) >= λ;
```

If we insist on $\lambda = 3$, the smallest counterexample we can find would be $t_1t_4t_5t_6$ (see Example 4). However, with the flexibility of SPCP, we can find a small counterexample t_1t_6, which differentiates Q_1 and Q_2 for $\lambda = 1$.

We can adapt the algorithm AGG-base to work for SPCP. The only change would be that we treat the query parameter λ as a symbolic variable when computing aggregate provenance. For example, doing so amounts to replacing occurrences of the constant 3 with λ in Table 2 for Example 4. We impose no constraint on this variable to the solver, so it can find small counterexamples without adhering to the original parameter setting.

3.2.3 A Heuristic Improvement. AGG-base may not scale very well when a group contains many tuples, because the SMT constraints will involve many variables and become difficult to solve. Thus, we develop a heuristic improvement called AGG-impr, targeting Q_1 and Q_2 where both end with an aggregation operator followed by an optional selection operator. The key intuition behind AGG-impr is to heuristically focus on differentiating two corresponding groups of tuples computed by Q_1 and Q_2 before they are aggregated. A counterexample that differentiates these two groups will likely lead to different result tuples after aggregation. Recall Example 3, where Q_1 and Q_2 differ in the aggregate value of result group for Mary (because Q_2 forgot to restrict to CS courses). Disregarding aggregation, if we simply focus on differentiating members in the Mary group, we will obtain a counterexample t_1t_6, which happens to differentiate Q_1 and Q_2 after aggregation.

In more detail, AGG-impr works as follows. First, it picks one group, identified by a specific group-by value a, for which $Q_1(D)$ and $Q_2(D)$ differ (either the group is in one result but not the other, or its aggregate value differs). Then, we formulate two queries Q_1' and Q_2' that compute the members of group a for Q_1 and Q_2 (resp.); they are formed by taking the subqueries of Q_1 and Q_2 below the aggregation operator and subjecting them to a selection condition setting the group-by attribute to a. We simply solve `SCP` on $Q_1'(D)$ and $Q_2'(D)$ to obtain a counterexample C.

Next, AGG-impr tests if C also works for Q_1 and Q_2, and if not, tries to change any query parameter in the final selection to make it work. To this end, let Q_1^* and Q_2^* denote the "remainder" queries that produce the final result tuples for group a from the results of Q_1' and Q_2', i.e., $Q_1(C) = Q_1^*(Q_1'(C))$ and $Q_2(C) = Q_2^*(Q_2'(C))$. Q_1^* and Q_2^* should involve only one aggregation followed by a selection parameterized by λ. In general, following an approach similar to earlier sections, we can derive a Boolean expression from Q_1^*, Q_2^*, and the actual contents of $Q_1'(C)$ and $Q_2'(C)$ to check whether $Q_1(C) \neq Q_2(C)$, with only one variable λ. We then assert this expression and solve for a feasible setting of λ. In practice, the cases for various types of final selection conditions are simple enough that we use a set of hand-coded rules to set λ.

Finally, there is still a chance that C does not work for Q_1 and Q_2. In this case, AGG-impr can attempt another counterexample for Q_1' and Q_2', and repeat this process a number of times before giving up.

4 IMPLEMENTATION & EXPERIMENTS

Provenance has been not only studied extensively theoretically [2, 9, 19], but also implemented in various systems [3, 17, 18, 27, 38, 42]. However, to the best of our knowledge, no current system readily meets our need for supporting general SPJUD and aggregate queries. We have implemented our system, called RATest, on top of a SQL-based

database system by translating relational queries and provenance computation into SQL for execution. RATest builds on an open-source RA interpreter [48] that translates relational algebra queries (extended with group-by and aggregation) into SQL, using `WITH` to build up complex queries one relational operator at a time. To compute provenance, RATest rewrites the SQL fragment generated for each relational operator with logic to derive output provenance expressions from its input ones. All input and result (including intermediate) relations are augmented with an extra string-valued column for storing provenance expressions (Section 2.3); columns for aggregate values are replaced with string-valued ones for storing the corresponding symbolic expressions (Section 3.2.1). These expressions are computed using SQL in a bottom-up fashion through the query tree, by manipulating the strings encoding these expressions in the SMT-LIB format [5]. For additional details, see [32].

Once provenance has been computed by evaluating the rewritten SQL queries, RATest generates the SMT constraints and solves them using Z3 4.7.1, an efficient optimizing SMT Solver by Microsoft Research [8, 15], with the objective function minimizing the number of Boolean variables set to true. The satisfying models returned by the solver represent the counterexamples.

RATest is implemented mostly in Python, and the experiments in this section ran locally on a 64-bit Ubuntu 16.04 LTS server with 3.60GHz Intel Core i7-4790 CPU and 16GB 1600MHz DDR3 RAM, which also hosts the database (Microsoft SQLServer 2017). RATest additionally features a web-based interface, described in more detail in [33].

The following experiments focus on evaluating the efficiency and scalability of our algorithms as well as the quality of the counterexamples they find. Section 4.1 focuses on SPJUD queries collected from student submissions to a relational algebra assignment in an undergraduate database course at Duke University; therefore, the wrong queries were "real", although test database instances are synthetic. Section 4.2 uses TPC-H [14], where we manually created wrong versions of several benchmark queries.

4.1 Real-World SPJUD Queries

Queries in these experiments come from submissions by 141 students to a relational algebra assignment in Fall 2017, where they were asked to write SPJUD queries using the RA interpreter. There were 8 questions, 7 of which are relevant to our experiments (the first question was so simple that all students got it correct). These questions are similar (but not identical) to those used in our user study in Fall 2018 (see Section 5 for more more details); the database schema is the same. We use a synthetic test database instance D whose size can be adjusted. It was designed to catch some corner cases but there is no guarantee that it has complete coverage (in the sense that it can reveal all incorrect queries), and we observed that as we increase its size, more incorrect queries were caught:[2]

# tuples in D	1,000	4,000	10,000	40,000	100,000
# incorrect queries found	111	167	168	169	170

Some questions involve very complex queries to find tuples satisfying conditions with universal quantification or uniqueness quantification requiring multiple uses of difference (see Section 5 for concrete examples), and elicited some extremely complex student solutions with scores of operators; we are not aware of any directly related work that is able to handle this level of query complexity. We had to drop two overly complicated student queries that involved massive cross products.

SCP-all vs. SCP-sel. As discussed in Section 2.3, although SCP-all in theory is better at finding smallest counterexamples by considering *all potential answers* differentiating Q_1 and Q_2, doing so in practice can be expensive and may not deliver interactive performance – there is one student's query for which the solver did not finish in one hour, and we reported the smallest counterexample returned after one hour. Instead, SCP-sel focuses on only one tuple in the symmetric difference of $Q_1(D)$ and $Q_2(D)$ (we always arbitrarily pick the first one in return order), which is guaranteed to find a counterexample but not necessarily the smallest one. We would like to see how SCP-sel compares with SCP-all in terms of speed and quality of their solutions overall. Here, both approaches uses SMT:

	SCP-all	SCP-sel
average running time (sec)	75.89	3.80
average size of counterexample	3.52	3.52

Surprisingly, solutions found by SCP-sel for this workload have the same size as those found by SCP-all, even though SCP-sel considers only one candidate tuple and runs much master. Upon closer examination, for 168 out of the 170 wrong queries we discovered, all candidate tuples considered for the given wrong query have equally sized smallest witnesses. Hence, at least for this real-world query workload, SCP-sel has a very high probability of finding the global minimum while being much faster than SCP-all.

Solver Strategies. Here we further evaluate the time-quality trade-off of different solver strategies. SMT uses the optimizing SMT solver; SAT*Δ refers to the strategy of repeatedly invoking a SAT solver to find up to Δ satisfying models and report the minimum. Since the SAT solver may return an arbitrary model each time, for these experiments we run SAT*Δ 10 times and report the average minimum. Figure 3a shows the solution quality of SMT and SAT*128 across queries

[2] Interestingly, we also found big synthetic test databases to be far more effective in catching incorrect queries than human graders, who had no auto-grading support back in Fall 2017.

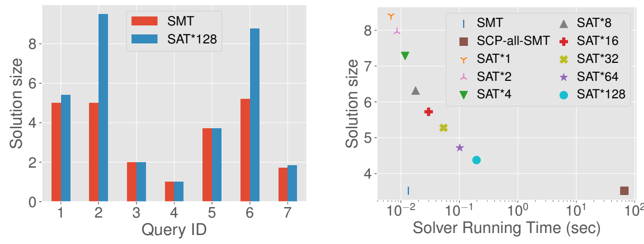


Figure 3: Comparison of solver strategies for student SPJUD queries. $N = 100$K. (a) Counterexample size by query. (b) Counterexample size vs. average solver time.

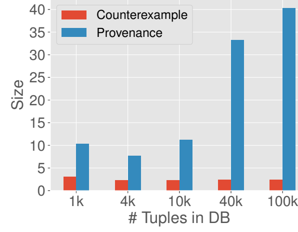


Figure 4: Size of counterexamples vs. provenance for student SPJUD queries. SCP-SEL with SMT.

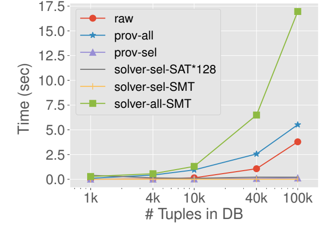


Figure 5: Breakdown of average running time vs. database size for student SPJUD queries.

(using SCP-SEL). SMT is the clear winner, consistently producing smaller counterexamples than SAT*128. For a closer look at the time-quality trade-off, Figure 3b further compares the average solver running time and counterexample size produced by SMT vs. SAT*Δ for varying Δ (from 1 to 128); for comparison, we also show SCP-ALL with SMT. Overall, we see that SMT offers much better time-quality trade-off—it is able find counterexamples smaller than any of SAT*Δ by spending just a little more time than SAT*4. Of course, results of these experiments heavily depend on the solver implementation; a comprehensive evaluation would be beyond the scope of this paper. Here, we simply observe that our implementation choice of SMT provides good performance and solution quality in practice, as it cannot be easily beaten by simply enumerating a number of satisfying models.

Size of Counterexamples vs. Provenance Expressions. Since our approach to finding counterexamples is based on provenance, an interesting question is whether provenance expressions themselves are small enough to serve as explanations instead. To this end, we compare the size of an counterexample given by our approach against the size of the provenance expression (for the same result tuple targeted by the counterexample). Here, we define the size of a provenance expression as the number of distinct variables (input tuples) it contains.[3] Figure 4 compares the average size of counterexamples and that of the corresponding provenance expressions, as we vary the size of the database D. We use SCP-SEL with SMT here. While the counterexample sizes depend on the particularities of each database instance, the general trend is that the average size of provenance increases with the database size, while the average size of counterexamples is much lower and remains stable even as the database becomes larger.

Running Time Breakdown vs. Database Size. To better understand the performance of our approach, we vary the size of the database and study the breakdown of running times in Figure 5. Here, **raw** is the time to evaluate the difference between the student query and the reference query (without computing any provenance); **prov-all** is for evaluating the rewritten query, which computes all potential answers and their provenance (a step needed in SCP-ALL); **prov-sel** is for evaluating the rewritten query with the extra selection targeting one specific tuple in the difference to obtain its provenance (a step used by SCP-SEL). We also show the total times spent on the solver for SCP-SEL with SMT (**solver-sel-SMT**), SCP-SEL with SAT*128 (**solver-sel-SAT*128**), and SCP-ALL with SMT (**solver-all-SMT**). From the figure, we see that full provenance computation adds significant overhead compared with normal query evaluation (**prov-all** vs. **raw**). However, we can drastically reduce this overhead by focusing on only one tuple in the difference: e.g., **prov-sel** is about 42× faster than **prov-all** and 29× faster than **raw** when $|D| = 100K$. Also thanks to this focus, solver time for SCP-SEL with SMT becomes negligible in comparison with other components of the running time for large databases; as shown in earlier experiments, it still produce small counterexamples comparable to those of SCP-ALL with SMT, which can be prohibitive for large databases.

Running Time vs. Query Complexity. Focusing on SCP-SEL with SMT, we now study how query complexity affects the running time (and its breakdown) in Figure 6. Besides the **total** running time, we also report its breakdown in terms of time spent on evaluating the difference between student and reference queries (**raw**), computing provenance for the selected tuple (**prov**), and solving the SMT (**solver**). We consider three measures of query complexity: number of operators, number of difference operators, and the height of the query tree.[4] If multiple queries have the same value for some complexity measure, we report their average running time.

[3]We note that the length of the expression can be much greater than this number because a variable may appear multiple times, and that simply setting variables to true may not lead to valid counterexample. In other words, the burden on the user to decipher a provenance expression will be higher than this particular definition of size implies.

[4]Here, a "query" refers to one (and only one) of $Q_1 - Q_2$ and $Q_2 - Q_1$ that SCP-SEL chooses to focus on, so the top operator is always a difference.

From these plots, we see that the cost of (selective) provenance computation and solving the SMT tends to increase with query complexity, but the total running time is dominated by just the time of checking the difference between student and reference queries, which fluctuates more.

4.2 Synthetic Aggregate Queries

We experimented on the TPC-H benchmark database generated at scale 1 (8.5 million tuples) with queries Q4, Q16, Q18, Q21, and Q21-S, a modified version of Q21 that ends with a final selection on aggregate value. We choose these queries because they do not involve arithmetic operations on aggregate functions. First we dropped the `ORDER BY` operator and rewrote these queries using the RA interpreter, then we experimented both the baseline approach (*AGG-BASE*) and the heuristic improvement (*AGG-IMPR*) discussed in Section 3. We also experimented with AGG-BASE with the parameterization extension on Q18 (it has a subquery involving selection predicate with aggregate value, and ends with another aggregation operator). We intentionally made two wrong queries for each query, whose errors include different selection conditions, incorrect use of difference, and incorrect position of projection. These are common errors in the students queries from the previous experiment.

Size of Counterexamples vs. Provenance Expressions. See Figure 7. **SC-Base** and **SC-Impr** are the sizes of counterexample returned by AGG-BASE and AGG-IMPR, respectively; **Prov-Base** and **Prov-Impr** are the sizes of provenance expressions used in AGG-BASE and AGG-IMPR, respectively. AGG-BASE does not finish running on Q4. The sizes of the counterexamples by both AGG-BASE and AGG-IMPR are significantly smaller than the original provenance for aggregate queries expressions for most queries (note the logarithmic scale). For some queries (Q4 and Q16), AGG-IMPR does not reduce the size of counterexamples over the how-provenance expression, because these queries are much simpler in structure compared to those involving multiple projections and differences in Section 4.1.

For Q18, AGG-BASE with parameterization extension (not shown in figure) returns counterexamples with 3.5 tuples on average, reducing by half of the size of counterexamples by both AGG-BASE and AGG-IMPR, while the solver running time only increases to 0.0210 seconds. This indicates that the parameterization extension helps us avoid large counterexamples required by selections involving aggregate values.

Running Time. Table 3 reports the running time of our algorithms for finding the smallest counterexample for each TPC-H query we experiment with. We present a breakdown of the execution time into time spent on evaluating the difference between wrong and reference queries, computing provenance for the selected tuples, and solving the SMT constraints. We find that the scalability of AGG-IMPR is much better than AGG-BASE, because it focuses on a single group member instead of the entire group. The performance of AGG-BASE is significantly affected by the number of tuples in the group, as the SMT solver struggles to scale.

Query	Raw Query Eval. Time	AGG-BASE		AGG-IMPR	
		Prov. Query Eval. Time	Solver Runtime	Prov. Query Eval. Time	Solver Runtime
Q4	3.6036	4.0403	timeout	0.0029	0.0151
Q16	0.8676	0.1349	0.2471	0.1084	0.0022
Q18	6.8751	0.0086	0.0134	0.0130	0.0039
Q21	21.5184	2.6205	31.1106	0.0577	0.0066
Q21-S	21.5408	2.8034	155.6828	0.0524	0.0061

Table 3: Running time (sec) for TPC-H queries.

5 USER STUDY

Since one motivation of our work is to provide small examples as explanations of why queries are incorrect, we built our RATEST system as a web-based teaching tool and deployed it in an undergraduate database class at Duke University in Fall 2018 with about 170 students. For one homework assignment, students needed to write relational algebra queries to answer 10 questions against a database of six tables about bars, beers, drinkers, and their relationships. The difficulties of these 10 problems range from simple to very difficult. The students had a small sample database instance to try their queries on. Their submissions were tested by an auto-grader against a large, hidden database instance designed to exercise more corner cases and catch more errors; if these answers differed from those returned by the correct queries (also hidden), the students would see the failed tests with some additional information about the error (but not the hidden database instance or the correct queries). The final submissions were then graded manually informed by the auto-grader results; partial credits were given. For the purpose of this user study, we normalize the student score for each question to $[0, 100]$.

We did not wish to create unfair advantages for or undue burdens on students with our user study. This consideration constrained our user study design. For example, we ruled out the option of dividing students into groups where only some of them benefit from RATEST; we also ruled out creating additional homework problems without counting them towards the course grades. Therefore, we made the use of RATEST completely optional (and with no extra incentives other than the help RATEST offers itself). RATEST was given the correct queries and the same database instance used by the auto-grader for testing. If a student query returned an incorrect result, RATEST would show a small database instance (a subset of the hidden one), together with the results of the incorrect query and the hidden correct query on this small instance. We made RATEST available for only 5 out of the 10 problems. Leaving some problems out allowed us to study

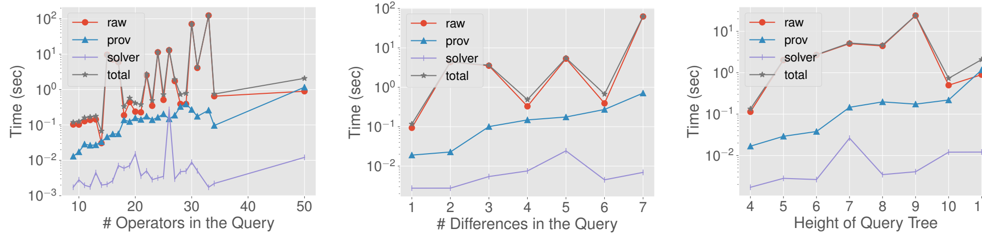


Figure 6: Running time and its breakdown vs. various measures of query complexity. SCP-SEL with SMT. $|D| = 100K$.

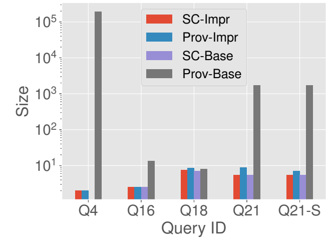


Figure 7: Size of counterexamples vs. provenance for TPC-H queries.

how the same student's performance on different problems might be influenced by the use of RATEST. The 5 problems were chosen to cover the entire range of difficulties:

(b) *Find drinkers who frequent any bar serving Corona.*
(d) *Find drinkers who frequent both JJ Pub and Satisfaction.*
(e) *Find bars frequented by either Ben or Dan, but not both.*
(g) *For each bar, find the drinker who frequents it the greatest number of times.*
(h) *Find all drinkers who frequent only those bars that serve some beers they like.*

Students must use basic relational algebra; in particular, they were not allowed to use aggregation. Problems (g) and (i) are more challenging than others: (g) involves non-trivial uses of self-join and difference; (i) involves two uses of difference.

We released RATEST a week in advance of the homework due date. We collected usage patterns on RATEST, as well as how students eventually scored on the homework problems. Ideally, we wanted to answer the following questions: (i) Did students who used RATEST do better than those who did not? (ii) For students who used RATEST, how did they do on problems with and without RATEST's help? We should note upfront that we expected no simple answers to these questions, as scores could be impacted by a variety of factors, including the inherent difficulty of a question itself, individual students' abilities and motivation, as well as the learning effect (where one gets better at writing queries in general after more exercises). Therefore, to supplement quantitative analysis of usage patterns and scores, we also gave an optional, anonymous questionnaire to all students after the homework due date.

Quantitative Analysis of Usage Patterns and Scores. Before exploring the impact of RATEST on student scores, let us examine some basic usage statistics, summarized in Figure 8. Overall, 137 students (more than 80% of the class) attempted a total of 3,146 submissions to RATEST. The sheer volume of the usage speaks to the demand for tools like RATEST, and the sustained usage (across problems) suggests that the students found RATEST useful. It is also worth noting that the number of attempts

Did the student use RATEST?		No	Yes
Problem (b)	# of students	67	102
	Mean score	100.00	100.00
	Std. dev.	0.00	0.00
Problem (d)	# of students	76	93
	Mean score	100.00	100.00
	Std. dev.	0.00	0.00
Problem (e)	# of students	69	100
	Mean score	99.03	99.67
	Std. dev.	5.63	3.33
Problem (g)	# of students	70	99
	Mean score	92.38	97.98
	Std. dev.	26.11	14.14
Problem (i)	# of students	49	120
	Mean score	89.80	94.40
	Std. dev.	30.58	19.00

Table 4: Comparison of performance between students who did not use RATEST and those who did, on problems for which RATEST was available.

reflects problem difficulty; for example, (i), the most difficult problem, took far more attempts than other problems. We also note that while RATEST helped the vast of majority of its users get the correct queries in the end; some users never did. We observed in the usage log some unintended uses of RATEST: e.g., one student made more than a hundred incorrect attempts on a problem, most of which contained basic errors (such as syntax); apparently, RATEST was used to just try queries out as opposed to debugging queries after they failed the auto-grader. Such outliers explain the phenomenon shown in Figure 8 where the overall average # of attempts were much higher than the average # before a correct attempt.

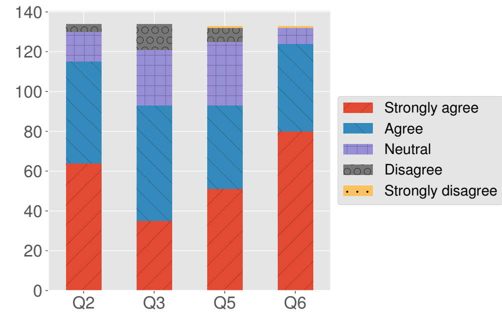
Next, we examine how the use of RATEST helps improve student scores. Table 4 compares the scores achieved by students who did not use RATEST versus those who did, on problem for which we made RATEST available. For simple problems such as (b), (d), and (e), there is no little or no difference at all, because nearly everybody got perfect scores with or without help from RATEST. However, for more difficult problems, (g) and (i), students who used RATEST had a clear advantage, with average scores improved from 92.38 to 97.98 and from 89.80 to 94.40, respectively. Of course, within the constraints of our user study, it is still difficult to conclude how much of this improvement comes from RATEST itself; it is conceivable that students who opted to use RATEST were

Problem	# of users		average # of attempts	
	total	who got a correct answer eventually	over all users	before a correct answer
(b)	102	93	4.08	1.79
(d)	93	93	3.12	1.57
(e)	100	95	5.24	3.45
(g)	99	91	5.90	3.76
(i)	120	94	11.10	7.46

Figure 8: Statistics on RATest usage.

Did the student use RATest for (i)?	No	Yes	Time of the first use (before due date)			
			5-7 days	3-4 days	2 days	1 day
# of students	49	120	45	30	16	29
Mean score on (i)	89.80	94.40	97.14	99.05	91.96	86.70
Std. dev.	30.58	19.00	15.41	5.22	25.54	26.16
Mean score on (h)	88.34	93.57	96.83	95.24	95.54	85.71
Std. dev.	31.77	20.86	14.89	18.12	17.86	30.06
Mean score on (j)	85.46	85.42	96.67	90.00	82.81	64.66
Std. dev.	34.17	34.39	16.51	30.51	37.33	47.02

Figure 9: Comparison of performance on (h) and (j) between students whether they used RATest for (i) or not.



Q2: RATest helped me tell whether my queries were correct.

Q3: The small example RATest provided helped me understand and fix the bug.

Q5: Compared with autograder, RATest is more helpful in debugging.

Q6: I'd like to use tools like RATest again for similar assignments in the future.

Figure 10: Results of student feedback.

simply more diligent and therefore would generally perform better than others. While we cannot definitively attribute all improvement in student performance to RATest, we next provide some evidence that it did help in a significant way.

Here, we zoom in on the three most difficult problems, (h), (i), and (j); RATest was only made available for (i). Problem (h) (*find all drinkers who frequent only those bars that serve **some** beers they like*) is quite similar to (i) (the difference being "some beers" vs. "only beers"). Problem (j) (*find all (bar1, bar2) pairs where the set of beer served at bar1 is a proper subset of those served at bar2*) on other hand requires very different solution strategy. Between those who did not use RATest for (i) and those who did, Figure 9 (focus on the first three columns and ignore the rest for now) compares their scores on (h) and (j). We see that for the similar problem (h), those who used RATest on (i) significantly improved their scores on (h), with a degree comparable to the improvement on (i). For the dissimilar problem (j), those who used RATest no (i) showed no improvement in their scores on (j)—the two score distributions are practically the same. We make two observations here. First, it is clear that not all improvement in student performance can be explained by student "diligence" alone; otherwise we would have seen higher performance on (j) for students who used RATest on (i). Second, a learning effect seems to exist: using RATest for one problem can help with a similar problem: (i) helps (h).

Figure 9, in its last four columns, also breaks down the statistics by when a student started to work on Problem (i). Not surprisingly, we see that "procrastinators" (those who started very close to the due date) performed clearly worse than others. If somebody started to work on (i) using RATest only the day before the homework was due, this individual would be expected to perform even worse than an "average" student who opted not to use RATest at all, especially for the last problem. It would have been nice if we can similarly break down the statistics for students who opted not to use RATest at all, but it was not possible in that case to know when they started to work on the problems. We could only conjecture that a similar trend might exist for procrastinators, so using RATest did not hurt any individual's performance.

Results of Anonymous Questionnaire. We collected 134 valid responses to our anonymous questionnaire; Figure 10 summarizes these responses. The feedback was largely positive. For instance, 69.4% of the respondents agree or strongly agree that the explanation by counterexamples helped them understand or fix the bug in their queries, and 93.2% would like to use similar tools in the future for assignments on querying databases. We also asked students which problems they found RATest to be most helpful (multiple choices were allowed): 58% voted for (g) and 94% voted for (i), which were indeed the most challenging ones. We also solicited open-ended comments on RATest. These comments were overwhelming positive and reinforces our conclusions from the quantitative analysis, e.g.:

- *"It was incredibly useful debugging edge cases in the larger dataset not provided in our sample dataset with behavior not explicitly described in the problem set."*
- *"Overall, very helpful and would like to see similar testers for future assignments."*
- *"I liked how it gave us a concise example showing what we did wrong."*

Summary. Overall, the conclusion of our user study is positive. Students who used RATest did better, and their improvement cannot be attributed all to merely the fact that

they opted to use an additional tool—RATest did add real value. Also, using RATest on one problem could also help with another problem, provided that the problems are similar. Finally, most students found RATest very useful and would like to use similar systems in the future.

6 RELATED AND FUTURE WORK

Test data generation. Cosette [12], which targets at deciding SQL equivalence without any test instances, encodes SQL queries to constraints using symbolic execution, and uses a constraint solver to find counterexamples that differentiates two input queries. The main difference of RATest with Cosette is in the use of the given test data instance D. While the reliance on D has its own issues, it also has a number of advantages. First, since RATest outputs a subinstance of D as the counterexample, it produces tuples that preserve the context and semantic of the input schema. Cosette returns counterexamples of arbitrary integer values, which may be harder for people to read and perceive. Second, since RATest computes provenance from actual input tuples, it is oblivious to complex or even black-box conditions in the input query. In contrast, Cosette only handles integer domains and queries that can be encoded into symbolic expressions. Technically, Cosette uses incremental solving to dynamically increase the size of each symbolic relation, thus it returns counterexamples with least number of distinct tuples, but the total number of tuples is not minimized. Further, Cosette does not handle database constraints explicitly and when there are selection predicates comparing aggregate values with numbers, i.e., the last challenge we discussed in Section 3.1, Cosette may fail to return a counterexample in minutes (we found such a case when there is a "having count(distinct column) > 2" in our modified TPC-H Q21). However, we note that some of our strategies like encoding integrity constraints into symbolic constraints and parameterizing aggregate queries can also be applied to Cosette. More detailed comparison can be found in the full version [32].

XData [10] generates test data by covering different types of query mutants of the standard query, without looking into wrong queries. Qex [46] is a tool for generating input relations and parameter values for a parameterized SQL query that also uses the SMT solver Z3, and aims at unit testing of SQL queries. It does not support nested queries and set operations and hence it cannot work for our problem because of our use of difference. Olston et al. [36] studied the problem of generating small example data for dataflow programs to help users understand the behavior of their dataflow programs.

Provenance and witness. Data provenance has primarily been studied for non-aggregate queries: Buneman et al. [9] defined why-provenance of an output tuple in the result set, which they call the witness basis. Green et al. [19] introduced how-provenance with the general framework of *provenance semiring*. Sarma et al. [41] gave algorithms for computing how-provenance over various RA operators in the Trio system. Amsterdamer et al. [2] extended the *provenance semiring* framework [19] to support aggregate queries. Besides these theoretical works, there are systems that capture different forms of provenance [3, 17, 18, 27, 38, 42]. However, to the best of our knowledge, no prior work considered `SWP/SCP`, and there are no systems available that support provenance for general SPJUD and aggregate queries.

Missing answers and why-not questions. The problem of explaining "why a certain tuple is not in the query answer" has been studied using two approaches: instance-based [21–23, 29] where explanations are (missing) input tuples, and query-based [11, 44] where explanations are based on query predicates or operators. Since we are trying to answer "why a tuple t is in the result of $(Q_1 - Q_2)(D)$", the problem "why t is not in $Q_2(D)$" and the instance-based approach are quite related to our problem. However, it cannot be directly applied to solve our problem: on one hand, existing works only consider monotone queries; on the other hand, the instance-based solution provides input tuples whose insertion will make the missing tuple t appear in the query result of Q_2, which does not help differentiate the two queries.

Teaching or grading tool for programming. Due to popularity of students taking programming-related courses, teaching and grading tools for programming assignments that automatically generate feedback for submissions are receiving a lot of attention [20, 26, 37]. In the database community, Chandra et al. built XData [10] that can be used for grading by generating multiple test cases for different query mutants, as well as giving immediate feedback to student. The latter is similar to our RATest tool. Jiang and Nandi [25, 34] designed and prototyped interactive electronic textbook to help students get rapid feedbacks from querying the database with novel interaction techniques.

Explanations for query answers. Explanations based on tuples in the provenance has been recently studied by Wu-Madden [47] and Roy-Suciu [40]. These works take an aggregate query and a user question as input, find tuples whose removal will change the answer in the opposite direction, and returns a compact summary as explanations.

Future work. Building user-friendly tools to help students or programmers learn and debug database queries is an interesting research direction. In particular, building a similar tool with the full functionality of SQL queries is a challenging open problem.

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. On the Limitations of Provenance for Queries with Difference. In *TaPP*.

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

[3] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. 2014. A generic provenance middleware for database queries, updates, and transactions. In *TaPP*.

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, et al. 2011. CVC4. In *CAV '11*, Vol. 6806. Springer, 171–177.

[5] 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.

[6] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343.

[7] Armin Biere. [n. d.]. CaDiCaL: Simplified Satisfiability Solver. https://github.com/arminbiere/cadical. [Online; accessed 24-Oct-2018].

[8] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ-an optimizing SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 194–199.

[9] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *International conference on database theory*. Springer, 316–330.

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

[11] Adriane Chapman and H. V. Jagadish. 2009. Why not?. In *SIGMOD*. 523–534.

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

[13] Sara Cohen, Yehoshua Sagiv, and Werner Nutt. 2005. Equivalences among aggregate queries with negation. *ACM Transactions on Computational Logic (TOCL)* 6, 2 (2005), 328–360.

[14] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www.tcp.org/hspec.html* 21 (2008), 592–603.

[15] 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.

[16] Michael R Garey, David S. Johnson, and Larry Stockmeyer. 1976. Some simplified NP-complete graph problems. *Theoretical computer science* 1, 3 (1976), 237–267.

[17] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*. 174–185.

[18] Todd J Green, Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. 2007. Update exchange with mappings and provenance. In *PVLDB*. 675–686.

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

[20] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*. 1345–1351.

[21] Melanie Herschel and Mauricio A. Hernández. 2010. Explaining Missing Answers to SPJUA Queries. *PVLDB* 3, 1 (2010), 185–196.

[22] Melanie Herschel, Mauricio A. Hernández, and Wang Chiew Tan. 2009. Artemis: A System for Analyzing Missing Answers. *PVLDB* 2, 2 (2009), 1550–1553.

[23] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. 2008. On the provenance of non-answers to queries over extracted data. *PVLDB* 1, 1 (2008), 736–747.

[24] Tomasz Imieliński and Witold Lipski, Jr. [n. d.]. Incomplete Information in Relational Databases. *J. ACM* 31, 4 ([n. d.]), 761–791.

[25] Lilong Jiang and Arnab Nandi. 2015. Designing interactive query interfaces to teach database systems in the classroom. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. 1479–1482.

[26] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *SIGSOFT*. 739–750.

[27] Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. 2010. Querying data provenance. In *SIGMOD*. 951–962.

[28] Stefan Kratsch, Dániel Marx, and Magnus Wahlström. 2010. Parameterized complexity and kernelizability of max ones and exact ones problems. In *MFCS*. 489–500.

[29] Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. 2017. A SQL-Middleware Unifying Why and Why-Not Provenance for First-Order Queries. In *ICDE*. 485–496.

[30] Michael Ley and Schloss Dagstuhl. 2018. DBLP database. https://dblp.uni-trier.de/xml/. (2018).

[31] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 607–618.

[32] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. *Arxiv* (2019). https://arxiv.org/abs/1904.04467

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

[34] Arnab Nandi. 2015. Breathing Life into Database Textbooks. In *CIDR*.

[35] Werner Nutt, Yehoshus Sagiv, and Sara Shurin. 1998. Deciding equivalences among aggregate queries. In *PODS*. 214–223.

[36] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. 2009. Generating example data for dataflow programs. In *SIGMOD*. 245–256.

[37] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 92–97.

[38] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained lineage at interactive speed. *PVLDB* 11, 6 (2018), 719–732.

[39] Sudeepa Roy, Vittorio Perduca, and Val Tannen. 2011. Faster query answering in probabilistic databases using read-once functions. In *ICDT*. 232–243.

[40] Sudeepa Roy and Dan Suciu. 2014. A formal approach to finding explanations for database queries. In *SIGMOD*. 1579–1590.

[41] Anish Das Sarma, Martin Theobald, and Jennifer Widom. 2008. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*. IEEE, 1023–1032.

[42] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: provenance and probability management in postgreSQL. *PVLDB* 11, 12 (2018), 2034–2037.

[43] Niklas Sörensson and Niklas Eén. 2009. Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *SAT* (2009), 31.

[44] Quoc Trung Tran and Chee-Yong Chan. 2010. How to ConQueR Why-not Questions. In *SIGMOD*. 15–26.

[45] Moshe Y Vardi. 1982. The complexity of relational query languages. In *STOC*. 137–146.

[46] Margus Veanes, Nikolai Tillmann, and Jonathan De Halleux. 2010. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 425–446.

[47] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *PVLDB* 6, 8 (2013), 553–564.

[48] Jun Yang. 2018. RA (radb): A relational algebra interpreter over relational databases. https://github.com/junyang/radb.

A PROOFS FROM SECTION 2.2

We will give the proofs of theorems in Table 1 in this section.

A.1 SJ and SPU Queries

Given $t \in Q_1(D) \setminus Q_2(D)$, the poly-time algorithm for SJ and SPU queries involve finding a smallest witness of t in D for Q_1, and using the fact that since Q_2 is monotone and $t \notin Q_2(D)$, therefore, $\forall D' \subseteq D$, $t \notin Q_2(D')$.

Theorem 1. *The SWP for two SJ queries is poly-time solvable in combined complexity.*

Proof. Let $R_1, ..., R_k$ be all relations that participate in the SJ query Q_1. For each relation R_i, $i \in [1, k]$, there must exist exactly one tuple $t_i = t.R_i$ (the R_i component of t), which is part of the witness of t (under set semantic). Since each t_i must satisfy all selection conditions for t to appear in $Q_1(D)$, the set $D_t = \{t_i | i \in [1, k]\}$ ensures that $t \in Q_1(D_t)$, and is also minimal. Since Q_2 is monotone and $t \notin Q_2(D)$, we have $t \notin Q_2(D_t)$; hence $t \in (Q_1 - Q_2)(D_t)$. The running time to find D_t is polynomial in k, giving polynomial combined complexity. □

When projection is allowed, an output tuple may have multiple minimal witnesses, and we pick any one of them.

Theorem 2. *The SWP for two SPU queries is polynomial-time solvable in combined complexity.*

Proof. For an SPU query Q_1, if $t \in Q_1(D)$, at least one tuple t' in one of the input relations must satisfy the selection condition (if any), and its projected attribute values must match that of t. The smallest witness D_t only consists of only t'. Since Q_2 is monotone and $t \notin Q_2(D)$, we have $t \notin Q_2(D_t)$. The running time to find $D_t = \{t'\}$ is polynomial in the size of the Q_1 and the input database. □

A.2 PJ Queries

For queries involving both projection and join, we show that it is NP-hard in query complexity to find the smallest witness, even when the query can be evaluated in poly-time.

Theorem 3. *The SWP for two PJ queries is NP-hard in query complexity.*

Proof. We prove the theorem by a reduction from the vertex cover problem with vertex degree at most 3, which is known to be NP-complete [16] and is defined as follows: Given an undirected graph $G(V, E)$ with vertex set V and edge set E, where the degree of every vertex is at most 3, decide whether there exists a vertex cover C of at most p vertices such that each edge in E is adjacent to at least one vertex in the set.

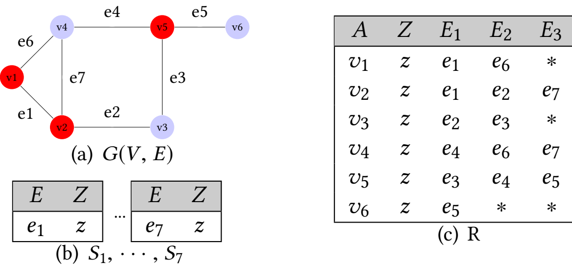
Construction. Given $G(V, E)$, suppose $V = \{v_1, ..., v_n\}$, and $E = \{e_1, \cdots, e_m\}$. We encode each vertex as a tuple in the relation $R(A, Z, E_1, E_2, E_3)$. For each vertex $v_i \in V$, R contains a tuple $t_i = (v_i, z, e_{i_1}, e_{i_2}, e_{i_3})$, where $e_{i_1}, e_{i_2}, e_{i_3}$ are identifiers of edges adjacent to v_i, $i_1 < i_2 < i_3$. If the degree of v_i is less than 3, the identifiers are replaced by a null symbol "$*$". The attribute $Z = z$ is a constant for all tuples. In addition to R, we have m relations $S_1, ..., S_m$. Each S_i, $i \in [1, m]$, has schema $S_i(E, Z)$. For the edge $e_i \in E$, S_i contains a single tuple (e_i, z). Let $D = (R, S_1, ..., S_m)$ be the database instance.

Next, we construct Q_1 involving PJ that consist of m subqueries as follows: For all $i \in [1, m]$, let $q_i = \pi_Z(R \bowtie_{R.E_1=S_i.E \vee R.E_2=S_i.E \vee R.E_3=S_i.E} S_i)$, which operates on S_i and R, checks for match of $R.E_1, R.E_2$, or $R.E_3$ with $S_i.E$, and then projects on to Z. Then we construct $Q_1 = q_1 \bowtie q_2 \bowtie \ldots \bowtie q_m]$ using natural join on Z. All queries q_i and Q_1 have a single attribute Z. Note that, initially, $q_i(D) = \{(z)\}$ for all $i \in [1, m]$, and therefore $Q_1(D) = \{(z)\}$ as well. The query Q_2 also outputs the attribute Z, but not the tuple $\{(z)\}$. We set $Q_2 = \pi_Z(R \bowtie_{R.Z \neq S_1.Z} S_1)$ (the choice of S_1 is arbitrary), and therefore $Q_2(D) = \{\}$ is empty. The tuple t for which we want to find the smallest witness in $(Q_1 - Q_2)(D)$ is (z). In other words, the goal is to find a subinstance $D' = (R', S'_1, ..., S'_m)$, $R' \subseteq R, S'_1 \subseteq S_1, ..., S'_m \subseteq S_m$, such that $(z) \in Q_1(D) \setminus Q_2(D)$.

Below we argue that G *has a vertex cover of size* $\leq p$, *if and only if the SWP instance above has a witness* D' *of size* $\leq p + m$ *where* m *is the number of edges in* G.

The "Only If" direction. Suppose we are given a vertex cover C with at most p vertices in G. We construct $R'_i = \{t_j \mid v_j \in C\}$, and $S'_i = S_i$ for all $i \in [1, m]$. Since $|C| \leq p$, $|D'| \leq p + m$ since each S_i contains a single tuple. Since C is a vertex cover, for all edge $e_i = (v_j, v_\ell) \in E$, either $v_j \in C$ or $v_\ell \in C$. Suppose without loss of generality (wlog.) $v_j \in C$. Then wlog. assume $t_j = (v_j, z, e_i, e', e'')$ where e', e'' are other two adjacent edges on v_j (they could be $*$ as well if the degree of v_j is < 3). The tuple t_j and the tuple $S_i(e_i, z)$ will satisfy the join condition of q_i (irrespective of the position of e_i in t_i), and the projection will output (z). Since C is a vertex cover, for all $i \in [1, m]$, $q_i(D') = \{(z)\}$. Therefore, $Q_1(D') = \{(z)\}$. $Q_2(D')$ remains empty. Hence $(z) \in Q_1(D') \setminus Q_2(D')$ Therefore, D' is a witness of (z) of size at most $p + m$.

The "If" direction. For the opposite direction, consider a witness $D' = (R', S'_1, ..., S'_m)$ where $R' \subseteq R, S'_1 \subseteq S_1, ..., S'_m \subseteq S_m$, $|R'| + |S'_1| + \ldots + |S'_m| \leq p + m$, such that $(z) \in Q_1(D') \setminus Q_2(D')$, i.e., $(z) \in Q_1(D')$. We construct $C = \{v_i \mid t_i \in R'\}$. Note that if $(z) \in Q_1(D')$, (z) must be in the result of all subqueries $q_i(D')$, $i \in [1, m]$. And $q_i(D')$ returns (z) if and only if (a) S'_i is not empty (i.e., $S'_i = S_i$ since S_i had only one tuple), and (b) if $e_i = (v_j, v_\ell)$, at least one of t_j or t_ℓ must appear in R' to satisfy the join condition in q_i; otherwise q_i



(a) $G(V, E)$

E	Z
e_1	z

...

E	Z
e_7	z

(b) $S_1, \cdots, S_7$

A	Z	E_1	E_2	E_3
v_1	z	e_1	e_6	*
v_2	z	e_1	e_2	e_7
v_3	z	e_2	e_3	*
v_4	z	e_4	e_6	e_7
v_5	z	e_3	e_4	e_5
v_6	z	e_5	*	*

(c) R

Figure 11: An example reduction for Theorem 3

returns an empty result and thus Q_1 returns an empty result. Therefore, all S'_i must be equal to S_i, $|S'_i| = 1$. Then we have $|S'_1| + ... + |S'_m| = m$. Since $|D'| \le p + m$, $|R'| \le p$, and thus we get a vertex cover C of size at most p.

An example reduction is shown in Figure 11. □

A.3 JU Queries

Theorem 4. *The SWP for two JU queries is NP-hard in query complexity.*

Proof. We reduce from the vertex cover problem.

Construction. Suppose $V = \{v_1, ..., v_n\}$ and $E = \{e_1, \cdots, e_m\}$ in the input graph $G(V, E)$ in the vertex cover problem. For each vertex v_i in G, there is a relation $R_i(Z)$ which consists of a single tuple (z). For each edge $e_i = (v_j, v_\ell) \in E$, we construct a query $q_i = R_j \cup R_\ell$. Then we construct a query $Q_1 = q_1 \bowtie \cdots \bowtie q_m$, where the join is a natural join on Z. We construct $Q_2 = R_1 \bowtie_{R_1.Z \neq R_2.Z} R_2$ (the choice of R_1, R_2 is arbitrary). Hence $D = (R_1, \cdots, R_n)$, $Q_1(D) = \{(z)\}$, and $Q_2(D) = \{\}$. The output tuple $(z) \in Q_1(D) \setminus Q_2(D)$, and the goal is to find a witness $D' = (R'_1, \cdots, R'_n)$ for (z) where $R'_i \subseteq R_i$ for all $i \in [1, n]$.

We show that *there exists a vertex cover C in G of size $\le p$ if and only if there is a witness D' for (z) of size $\le p$.*

The "Only If" direction. Consider a vertex cover C of G such that $|C| \le p$. If $v_i \in C$, then $R'_i = \{(z)\}$, otherwise $R'_i = \{\}$. Since C is a vertex cover, all edges must be covered. For an edge $e_i = (v_j, v_\ell)$, suppose wlog. $v_j \in C$. Hence the subquery $q_i = R_j \cup R_\ell$ returns (z) on D'. Therefore, $Q_1(D') = (z), Q_2(D') = \{\}, (z) \in Q_1(D') \setminus Q_2(D')$, i.e., D' is a witness for (z), and $|D'| = |C| \le p$.

The "If" direction. Consider any witness $D' = (R'_1, ..., R'_n)$ where $R'_1 \subseteq R_1, ..., R'_n \subseteq R_n$ and $|R'_1| + ... + |R'_n| \le p$, such that $(z) \in Q_1(D') \setminus Q_2(D')$, i.e., $(z) \in Q_1(D')$. Since R_i had only one tuple (z), either R'_i has (z) or it is empty. If tuple $(z) \in R'_i$, then we add vertex v_i to a set C. If (z) is in the result of $Q_1(D')$, (z) must be in the result of all subqueries $q_i(D')$ for all $i \in [1, m]$. For $e_i = (v_j, v_\ell)$, $q_i(D')$ returns (z) if and only if at least one of R'_j and R'_ℓ is not empty; otherwise q_i returns an empty result and thus Q_1 returns an empty result. Therefore, for each edge $e_i \in E$, at least one of its adjacent vertices v_j or v_ℓ must exist in C. Hence C is a vertex cover, and $|C| = |D'| \le p$. □

On the other hand, the following theorem shows that if all unions appear after all joins (which we call JU* queries), then the SWP can be solved in poly-time in combined complexity.

Theorem 5. *The SWP for two JU* queries is polynomial time solvable in combined complexity.*

Proof. Let $t \in Q_1(D) \setminus Q_2(D)$. According to Theorem 1, the SWP for SJ queries is polynomial time solvable in combined complexity. Hence, we look for the smallest witness of t in join-only part of Q_1, and choose the one with smallest number of tuples. The running time is polynomial in both $n = |D|$ and the size of the query. □

A.4 Size-Bounded SPJU Queries

Theorem 6 shows that if the SPJU queries are of bounded size (i.e. if we consider data complexity), there is a polynomial time algorithm for SWP. We prove this theorem using Proposition A.1, which is intuitive and known (e.g., [39]). We use *m-DNF* to refer to a DNF where each minterm has at most m literals.

Proposition A.1. *Given an SPJU query Q, a database instance D, and an output tuple $t \in Q(D)$, the how-provenance of t in $Q(D)$ can be transformed into a $k+1$-DNF in polynomial time when Q is of bounded size, where k is the number of join operations in Q.*

Theorem 6. *The SWP for two SPJU queries is polynomial-time solvable in data complexity.*

Proof. Let t be an output tuple in $Q_1(D) \setminus Q_2(D)$. Since Q_2 is monotone, $t \notin Q_2(D')$ for any $D' \subseteq D$. According to Proposition A.1, we can compute the how-provenance $\mathtt{Prv}_{Q_1 - Q_2, D}(t)$ in DNF in poly-time in data complexity. Then we scan the DNF to find the minterm with least number of literals, and this minterm represents the smallest witness for t in $Q_1(D) - Q_2(D)$. The literals in this clause are the identifiers of tuples in the smallest witness. □

E.g., if $\mathtt{Prv}(t) = a + bc$, then a forms the smallest witness.

A.5 Queries Involving Difference

Before discussing general SPJUD queries, let's focus on one special class of SPJUD queries where all differences appear after all SPJU operators (which we call SPJUD* queries). More formally, we define this class using formal grammar: $Q \rightarrow q^+ | Q - Q$, where q^+ is a terminal that represents SPJU queries. For instance, queries Q_1 and Q_2 in Example 1 are SPJUD* queries. The following theorem shows that the SWP can be solved in poly-time for SPJUD* queries.

Theorem 7. *The SWP for two SPJUD* queries is polynomial-time solvable in data complexity.*

Proof. Let t be an output tuple in $Q_1(D) \setminus Q_2(D)$. Since Q_1 and Q_2 are SPJUD* queries that can be written as nested differences of queries like $q_1 - q_2 - (q_3 - (q_4 - q_5)) - \ldots$, where all q_i-s are SPJU queries, $Q_1 - Q_2$ is also an SPJUD* query. The output tuple t must be either in or not in the result of each q_i. We find the smallest witness by enumerating the minimal witnesses of t w.r.t. every q_i and D. If t is in the result of $q_i(D)$, let w_i be the set of minimal witnesses of t w.r.t. q_i and D. Then we pick one element from every $w_i \cup \{\emptyset\}$, and construct w as the union of all witnesses or the empty set we picked. We evaluate Q_1 and Q_2 on w to see whether it is a witness for t, and record the w of the smallest size. We finish this procedure until we enumerate all combinations.

This procedure will return the smallest witness because: (i) if $t \notin q_i(D)$, t will also not be in $q_i(w)$ for any $w \subseteq D$ due to monotonicity, so we don't need to consider such q_i-s; (ii) Assume that w' is a smallest witness of t w.r.t. $Q_1 - Q_2$ and D, for all q_i where $t \in q_i(w')$, w' must be a superset of a minimal witness of t w.r.t. q_i and D. Hence w' must be the union of minimal witnesses of t w.r.t. these q_i-s and D; otherwise, if w' is a strict superset of the union of minimal witnesses of t, we can always remove tuples not belong to any minimal witness of t w.r.t. q_i-s and D from w', without affecting t to be in or not in any q_i, which contradicts the assumption that w' is a smallest witness. Therefore a smallest witness of t w.r.t. $Q_1 - Q_2$ and D must be union of minimal witness of t w.r.t. q_i and D, and thus it must be enumerated during the enumeration procedure.

The time complexity of entire enumeration process is $O(\Pi_i m^{k_i}) = O(m^{kd})$, where d is the number of difference operators, m is the max size of relations, k_i is the max complexity of each SPJU query q_i (i.e., the number of joins in q_i is $k_i - 1$), $k = \max_i k_i$ and d is the number of q_i-s. When queries are of bounded sizes, i.e., if d and k are fixed, the SWP for two SPJUD queries that can be written as nested differences of SPJU queries is polynomial-time solvable. □

SWP is NP-hard in general even for bounded-size queries.

Theorem 8. *The SWP for two SPJUD queries Q_1 and Q_2 is NP-hard in data complexity.*

Proof. We again give a reduction from the vertex cover problem with vertex degree at most 3 (see Theorem 3).

Construction. Suppose in $G = (V, E)$, $V = \{v_1, ..., v_n\}$, $E = \{e_1, \cdots, e_m\}$. We construct two relations $R(A, Z, E_1, E_2, E_3)$ and $S(B, C, Z)$. For each vertex $v_i \in V$, R contains a tuple $t_i = (v_i, z, e_{i_1}, e_{i_2}, e_{i_3})$, where $e_{i_1}, e_{i_2}, e_{i_3}$ are the identifiers of edges adjacent to v_i, $i_1 < i_2 < i_3$. If the degree of v_i is less than 3, the identifiers are replaced by a null symbol "$*$". Here z is a constant. For each edge $e_i \in E$, S contains a tuple $(e_i, e_{(i\%m)+1}, z)$, where $e_{(i\%m)+1}$ is the identifier of the next edge in the edge list (the next edge of e_m is e_1). Let $D = (R, S)$ be the database instance.

Next, we construct an SPJUD query that consists of several subqueries as follows: Let q_1 (on S) = $\pi_Z(S)$; q_2 (on S) = $\pi_{B,Z}(S)$; q_3 (on R, S)=$\pi_{S.C,S.Z}(S \bowtie_{S.C=E_1 \vee S.C=E_2 \vee S.C=E_3} R)$. Then we construct $Q_1 = q_1$, hence $Q_1(D) = \{(z)\}$. We also construct $Q_2 = \pi_Z(q_2 - q_3)$ (assume C in q_3 is renamed to B). For edge $e_i = (v_j, v_\ell)$, the edge e_i appears for both tuples t_j, t_ℓ (in one of E_1, E_2, E_3 attributes), and therefore, (e_i, z) appears in the result of $q_3(D)$ for every $i \in [1, m]$. Hence $q_3(D) = \pi_{B,Z}(S)$. So $q_2(D) \setminus q_3(D) = \emptyset$. Then $(Q_1 - Q_2)(D) = \{(z)\}$, and the goal is to find the smallest witness for (z). For the vertex cover instance in Figure 11(a), R will be as given in Figures 11(c), and S will contain tuples $\{(e_1, e_2, z), (e_2, e_3, z), \cdots (e_7, e_1, z)\}$.

We now show that *there exists a vertex cover C of size at most p in the graph G if and only if there is a witness $D' = (R', S')$ where $|R'| + |S'| \le p + m$.*

The "Only If" direction. Suppose we are given a vertex cover C of G with at most k vertices. Construct $R' = \{t_i \mid v_i \in C\}$, and $S' = S$. $Q_1(D) = Q_1(D') = \{(z)\}$ since S is unchanged. Similarly, $q_2(D') = \pi_{B,Z}(S)$ is unchanged. Since C is a vertex cover, for every edge $e_i = (v_j, v_\ell)$ either t_i or t_ℓ is in R'; hence $q_3(D') = q_3(D)$, i.e., each (e_i, z), $i \in [1, m]$ appears in $q_3(D')$. Then $Q_1 - Q_2$ outputs (z) on D', $|R'| = |C| \le p$, $|S'| = |S| = m$, and we get a witness of at most $p + m$ tuples.

The "If" direction. Consider any witness $D' = (R', S')$ where $R' \subseteq R, S' \subseteq S, |R'| + |S'| \le p + m$, such that $(z) \in Q_1(D') \setminus Q_2(D')$. We construct $C = \{v_i \mid t_i \in R'\}$. Since (z) is in $Q_1(D') \setminus Q_2(D')$, (z) must be in the result of $q_1(S')$, and not in the result of $q_2(S') - q_3(R', S')$, hence S' must contain at least one tuple. Therefore, $q_2(S')$ outputs at least one tuple (e_i, z) since S' is not empty. In turn, $q_3(R', S')$ must output all tuples in $q_2(S')$ to make $q_2(S') - q_3(R', S')$ empty. (a) We argue that $S' = S$. Suppose S' contains at least one tuple, say wlog, (e_1, e_2, z). Then to remove (e_1, z) from $q_2(S') \setminus q_3(R', S')$, $q_3(R', S')$ must contain (e_1, z), which can generate only from $S(e_m, e_1, z)$. Hence $(e_m, e_1, z) \in S'$. In turn, $(e_m, z) \in q_2(S')$. To remove it, we need $S(e_{m-1}, e_m, z)$ in S'. Continuing this argument (by induction), we get $S = S'$. (b) Consider any tuple, say wlog., (e_1, e_2, z) in S'. Then to remove (e_1, z) from $q_2(S') \setminus q_3(R', S')$, not only the tuple $(e_m, e_1, z) \in S'$, it also has to satisfy the join condition with R. This will hold only if for one of the end points v_j, v_ℓ of $e_1 = (v_j, v_\ell)$, $t_j \in R'$ or $t_\ell \in R'$. This should hold for all edges, and therefore the set C we constructed is a vertex cover. Since $|S'| = |S| = m$, $|R'| = |C| \le p$, therefore, we get a vertex cover in G of size at most p.

The queries we constructed during the reduction are all of bounded size, therefore the SWP for two SPJUD queries is NP-hard in data complexity. □