What Teaching Databases Taught Us about Researching Databases

Extended Talk Abstract

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

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

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

Hanze Meng
hanze.meng@duke.edu
Duke University
Durham, North Carolina, USA

Zhengjie Miao
zhengjie@sfu.ca
Simon Fraser University
Burnaby, Canada

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

Kristin Stephens-Martinez
ksm@cs.duke.edu
Duke University
Durham, North Carolina, USA

ABSTRACT

Declarative querying is a cornerstone of the success and longevity of database systems, yet it is challenging for novice learners accustomed to different coding paradigms. The transition is further hampered by a lack of query debugging tools compared to the plethora available for programming languages. The paper samples several systems that we build at Duke University to help students learn and debug database queries. These systems have not only helped scale up teaching and improve learning, but also inspired interesting research on databases. Furthermore, with the rise of generative AI, we argue that there is a heightened need for skills in scrutinizing and debugging AI-generated queries, and we outline several ongoing and future work directions aimed at addressing this challenge.

KEYWORDS

Query Debugging, Query Verification, Database Education, SQL, Relational Algebra

ACM Reference Format:
Jun Yang, Amir Gilad, Yihao Hu, Hanze Meng, Zhengjie Miao, Sudeepa Roy, and Kristin Stephens-Martinez. 2024. What Teaching Databases Taught Us about Researching Databases: Extended Talk Abstract. In *3rd International Workshop on Data Systems Education: Bridging education practice with education research (DataEd '24), June 09–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3663649.3664375

1 INTRODUCTION

Relational querying was invented more than half of a century ago [5]. While other programming languages wax and wane, SQL has remained among the most popular languages according to the latest Stack Overflow Developer Survey [15]. As database researchers, we take great pride in our tradition of declarative querying. This tradition, which enables automatic optimization, sets database systems apart from others with more imperative style of programming. It is arguably the most important feature underpinning the success and longevity of SQL databases.

However, declarative querying is very much an acquired taste. Many students struggle to learn to write and debug relational algebra or SQL, because most of them first learned to program in more imperative styles. Besides adjusting to a very different style, students also face a lack of tools for learning and debugging relational queries. Most programming languages have integrated development environments with sophisticated debugging support, but we have yet to see a true debugger for SQL (not counting syntax highlighters, autocompleters, or the systems we are currently building). One challenge lies in the declarative nature of SQL itself, for it is not even clear how to debug a declarative specification. While we can trace through and debug a query execution plan, this plan, because of automatic optimization, may bear little or no resemblance to the original query. Hence, insights gained through tracing the execution may not be helpful with fixing semantic errors in the query.

There is also a sense of urgency in addressing the challenges of debugging SQL, or in general, an implementation against a declarative specification. In recent years, Generative AI has become better at coding SQL as well as other languages by leaps and bounds. Looking forward, rather than writing low-level code that requires mastering idiosyncrasies of specific languages, developing high-level specifications may become a more important task for humans. On the other hand, despite continued improvement in AI code generation, there is still no reliable method for vetting the semantic correctness of complex queries and code. This problem could be exacerbated in the future, when fewer people might have the same level of mastery and experience with specific languages as today. While this future may not sound palatable, it is a possibility that we should not risk overlooking and fail to prepare for. Given the

DataEd '24, June 09–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0678-3/24/06
https://doi.org/10.1145/3663649.3664375

declarative tradition of our database community, it stands to reason that we should be at the forefront in tackling the challenge of how to prepare our next generation for the future.

In this short paper accompanying a keynote talk at DataEd 2024, we outline several systems developed at Duke University to address the challenges outlined above. Collectively they belong to a project we call *HNRQ: Helping Novices Learn and Debug Relational Queries*. We will start with the classroom setting and discuss how to use small examples to help students see wrong queries (Section 2), and how to offer actionable hints to help students fix wrong queries (Section 3). We will then consider a more general setting where a correct solution in SQL may not be known in advance, and discuss our design for a SQL debugger and a vision for verifying query correctness without assuming knowledge of SQL (Section 4). We conclude in Section 5 with acknowledgments.

Disclaimers. There is considerable content overlap with [14], particularly in Section 2, as both [14] and this paper are extended abstracts for invited talks summarizing collaborative work under the HNRQ project. This paper is meant to provide an overview of our work along with some high-level intuitions; for technical details as well as discussion of related work, please refer to our previously published papers, cited in this paper.

2 RATEST & CINSGEN: UNDERSTANDING QUERIES THROUGH EXAMPLES

Every database teacher has faced student questions along these lines: "Why did you mark my query wrong?" Or, in an era when we have turned increasingly to automation to cope with exploding enrollment in computer science classes: "Why did my query fail the autograder?" Consider the following database schema about beers, drinkers, bars, and their relationships:

```
Beer(name, brewer), Drinker(name, address), Bar(name, address),
Likes(drinker, beer), Serves(bar, beer, price),
Frequents(drinker, bar, times_a_week)
```

Suppose we ask students to write a query to find, for each beer Eve likes, the bar(s) serving it at the highest price. Here is a *reference query* $Q^\star$ that answers this question correctly:

```
SELECT DISTINCT S1.beer, S1.bar -- Q*
FROM Likes L, Serves S1
WHERE L.drinker LIKE 'Eve %'
  AND L.beer = S1.beer AND NOT EXISTS (SELECT * FROM Serves S2
        WHERE S2.beer = S1.beer AND S2.price > S1.price);
```

A student may write the following incorrect query:

```
SELECT DISTINCT S1.beer, S1.bar -- Q
FROM Likes L, Serves S1, Serves S2
WHERE L.drinker LIKE 'Eve %'
  AND L.beer = S1.beer AND S1.beer = S2.beer AND S1.price >
        S2.price;
```

For each beer Eve likes, this incorrect query Q finds the bar *not* serving it at the *lowest* price. For untrained eyes, spotting this mistake is challenging. An autograder can detect that $Q(D) \neq Q^\star(D)$ for some test database instance D, but debugging on these instances, even moderately sized ones, can be overwhelming for students. Imagine what experienced teachers would do: instead of using the whole test instance D, they would choose a small example to illustrate the mistakes: e.g., the instance in Table 1, with only 9 rows total, is enough to show that $Q \not\equiv Q^\star$.

How to derive such small counterexample instances automatically is at heart of the RATest [12, 13] system we have developed. Given two queries Q and $Q^\star$ and a database instance D where $Q(D) \neq Q^\star(D)$, we want to find a small instance $D' \subseteq D$ such that $Q(D') \neq Q^\star(D')$.[1] A practical way of solving this problem is to pick a row t in the symmetric difference between $Q(D)$ and $Q^\star(D)$, and find a small $D' \subseteq D$ such that t remains in the symmetric difference between $Q(D')$ and $Q^\star(D')$. To this end, we trace the "provenance" [7] for t back to base table rows, and find the minimum set of such rows needed to support the derivation of t, which reduces to solving a *min-ones satisfiability* problem. The problem complexity depends on the size of the provenance expression and is heavily influenced by the type of queries. With some performance optimizations, such as aggressively pushing down selections induced by the choice of t, RATest is able to find small counterexample instances for most queries at interactive speed.

However, aggregation still posed particularly interesting challenges. We had to change our problem definition for practicality—not just to achieve acceptable running time but also to have bounded-size counterexamples. For instance, consider the following queries:

```
SELECT beer, AVG(price) FROM Serves GROUP BY beer; -- Q1
SELECT beer, AVG(price) FROM Serves GROUP BY beer -- Q2
  HAVING COUNT(*) > 1000;
```

For Q_1, insisting the exact result row t be returned by Q_1 would force RATest to find a subset of price values with the same exact average as the overall, a daunting task that may not yield any proper subset. For Q_2, its `HAVING` condition would force any instance to have at least 1000 rows. Therefore, to address the issues exposed by the above examples, RATest does not attempt to reproduce the same row t in the difference between two query results; rather, it simply ensures that the queries return different results for t's group. To further enable smaller counterexamples, RATest may *parameterize* queries differently: e.g., instead of constructing a counterexample for `HAVING COUNT(*)` $> k$ where $k = 1000$ as in the original query, we can do so for $k = 3$, assuming that the query is written in a way that generalizes to different k values.

Since its first deployment in a large undergraduate database course at Duke University in Fall 2018, RATest has become a staple in our database curriculum, benefiting more than 1,600 students to date. Our user study [12] showed that RATest improved student performance on challenging home problems, and an overwhelming majority of users found the tool helpful in understanding their mistakes and and debugging queries. In retrospect, we attribute RATest's success to how well it fits the learning setting. Importantly, RATest's counterexamples do not reveal solution queries or full test database instances, which would have introduced incentives to game the system. For students struggling to understand why

[1] At this point, it is instructive to compare RATest with the seminal work of Cosette [4], which can also generate small counterexample instances to show that two queries are not equivalent. However, Cosette does not assume the knowledge of a large instance that already differentiates the queries. This feature is a double-edged sword: on one hand, not relying on any test instances makes Cosette more powerful than RATest; on the other hand, Cosette must deal with the generally *undecidable* problem of testing query equivalence [1], which necessarily implies that Cosette may fail to find counterexamples for some queries. In contrast, RATest's assumption of a known large counterexample D immediately makes the problem decidable, which allows us to support more queries that may arise in practice. Also, if D a real-world instance, counterexamples constructed out of D will have meaningful concrete values that are easier for students to understand.

Drinker

name	address
Eve Edwards	32767 Magic Way

Likes

drinker	beer
Eve Edwards	American Pale Ale

Beer

name	brewer
American Pale Ale	Sierra Nevada

Bar

name	address
Restaurant Memory	1276 Evans Estate
Tadim	082 Julia Underpass
Restaurante Raffaele	7357 Dalton Walks

Serves

bar	beer	price
Restaurant Memory	American Pale Ale	2.25
Tadim	American Pale Ale	2.75
Restaurante Raffaele	American Pale Ale	3.50

→

Result of $Q^\star$

beer	bar
American Pale Ale	Restaurante Raffaele

Result of Q

beer	bar
American Pale Ale	Restaurante Raffaele
American Pale Ale	Tadim

Table 1: Concrete example illustrating $Q \not\equiv Q^\star$.

Drinker

name	address
$drinker_1$	⋆

Likes

drinker	beer
$drinker_1$	$beer_1$

Beer

name	brewer
$beer_1$	⋆

Bar

name	address
bar_1	⋆
bar_2	⋆
bar_3	⋆

Serves

bar	beer	price
bar_1	$beer_1$	$price_1$
bar_2	$beer_1$	$price_2$
bar_3	$beer_1$	$price_3$

Global condition

$drinker_1$ LIKE 'Eve %'
AND $price_1 < price_2 < price_3$

→

Result of $Q^\star$

beer	bar
$beer_1$	bar_3

Result of Q

beer	bar
$beer_1$	bar_3
$beer_1$	bar_2

Table 2: Abstract example illustrating $Q \not\equiv Q^\star$.

their queries fail autograder tests, RATest offers small, illustrative examples that are *tailored* towards their mistakes, and the supply of these examples is seemingly infinite—they can keep debugging until their queries are correct. It would be impossible to achieve such a tailored learning experience with manual effort, given the oversize and understaffed classes nowadays in our discipline.

Concrete vs. Abstract Examples. Despite its success, we wanted to improve RATest further. First, the concrete instances contain so many details to the point of being distracting: e.g., in Table 1, drinker and bar addresses do not contribute to understanding in any way; the specific serving prices do not matter either, so long as they are ordered such that there exists a price between the highest and the lowest to differentiate $Q^\star$ and Q. An idea we had is to *abstract* an example to hide unnecessary details and highlight its properties required for an counterexample. This was one of the motivations for our follow-up system called CINSGEN [6, 10].

Table 2 shows the abstract counterexample produced by CINSGEN, which generalizes the concrete one in Table 1. This abstract example is a *conditional instance* (or *c-instance* for short), which was inspired by *c-tables* for incomplete databases [9]. In a c-instance, concrete values are replaced with named variables (such as $price_1$) or "don't-care" (⋆), and there is a global condition involving the named variables. Each c-instance conceptually represents all possible concrete instances where named variables are instantiated with specific values that satisfy the global condition. It is not difficult to see from Table 2 that c-instances are more compact and informative than concrete instances.

CINSGEN in fact tackles the more general problem of constructing c-instances to "satisfy" (i.e., generate result rows for) a given query. Applied in the setting of illustrating mistakes in Q relative to $Q^\star$, it constructs c-instances to satisfy the symmetric difference between Q and $Q^\star$. Additionally, CINSGEN aims to enumerate all ways to satisfy a query (motivated by the notion of *coverage* in software testing), allowing it to construct c-instances to illustrate different mistakes in Q, without assuming any known test instance.[2] Our user study of CINSGEN [6] showed that it further enhanced students' ability to find bugs in queries beyond RATest. However, interestingly, when asked whether they preferred concrete instances (RATest) versus abstract instances (CINSGEN), more preferred concrete instances. This preference is unfortunate but understandable. Luckily, it also appears to be *conditionable*: the graduate students involved the study were more comfortable with abstract instances than the undergraduate students. We shall revisit this point later in Section 4.

3 QR-HINT: FROM UNDERSTANDING TO FIXING QUERIES

Even if students understand why their queries are wrong, it is not always clear how to fix their queries. Therefore, we have built the Qr-Hint [8] system to automatically suggest fixes. As an example, given the same database schema in Section 2, suppose we want to rank, for each beer Amy likes, the bars she frequents serving this beer according to the serving price. Assuming no ties for simplicity, here is the (correct) reference query $Q^\star$:

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

Here is a student's query Q, which is incorrect:

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

Some mistakes are easy to spot: e.g., Q misses table `Frequents` in its join. Other mistakes are tricky. For example, it is tempting to suggest changing > to <= in Q's `s1.price > s2.price` to match $Q^\star$, but this suggestion is wrong: a careful inspection of Q would reveal that it intends table aliases `s1` and `s2` to play the roles of `S2` and `S1` (note the reverse order) in $Q^\star$, so the correct suggestion should be changing > to >=. Finally, many syntactic differences between Q and $Q^\star$ do not lead to semantic differences. For example, it is fine for Q to check Amy in `WHERE` instead of `HAVING`, and not to include `drinker` in its `GROUP BY`.

[2] Now, without this assumption, we are in the same territory as Cosette [4]: because query equivalence is undecidable in general, CINSGEN cannot guarantee that it will find a c-instance counterexample.

The above example highlights several challenges. Because there are many ways to write a correct query that are syntactically different but logically equivalent, we cannot rely on syntax alone to suggest fixes. On the other hand, good teachers will not ignore how the wrong query is written in suggesting fixes—"let's go through your query and fix it" offers a better learning experience than "forget it, just do what the solution does." Furthermore, in most cases, there is no easy way to say that a part of a query is "definitely" wrong. For example, suppose that a correct query contains the predicate `A>=0 AND A+B>0`, while a wrong query says `A>0 AND A+B>0`. We cannot simply call `A>0` wrong, because it is still part of another correct query `(A>0 AND A+B>0) OR (A=0 AND B>0)`. This observation prompted us to shift our perspective/approach from finding wrong parts of a query to a more constructive one of finding *repairs* to make a query correct.

These insights led us to the following (idealized) problem formulation. Given a query Q not equivalent to the reference query $Q^\star$, consider all queries logically equivalent to $Q^\star$ but possibly very different syntactically. Among these, we would like to pick $Q^{\star\prime} \equiv Q^\star$ whose syntactic edit distance (as a measure for repair cost) to Q is minimized. Unfortunately, this idealized formulation entails solving the query equivalence problem, which is undecidable in general as discussed also in Section 2. Furthermore, for a wrong query with multiple issues such as Q above, this formulation offers no guidance on how to provide hints: students often get confused when asked to come up with multiple repairs that need to work together as a whole to make Q correct.

In the end, we developed a practical yet still principled approach for Qr-Hint, based on the idea of hinting and fixing a query step by step in *stages*. Stages naturally correspond to components of a single-block SQL query (including grouping and aggregation). Each stage has a "viability check" that the working query Q must pass before advancing to the next, which sets an explainable goal for the student to achieve. Qr-Hint guarantees that following its hints for a given stage will lead to a query passing this check. Furthermore, if Q passes a stage's viability check, there must exist some correct query (i.e., equivalent to $Q^\star$) in the end that agrees with Q up to and including this stage; in other words, Qr-Hint ensures steady progress towards the goal, so the student never has to revisit an earlier stage. While we cannot guarantee the overall optimality of hints and fixes because of reduction from query equivalence testing, Qr-Hint can provide some local optimality guarantees for its repair algorithm within each stage.

As an example, for the example $(Q, Q^\star)$ pair in this section, a teacher can leverage Qr-Hint as follows. First, in the `FROM` stage, whose goal is to ensure Q gets data from the right multiset of tables, Qr-Hint finds that `Frequents` needs to be added to `FROM`; based on this information, the teacher may ask the student to double-check Q's `FROM` tables to see what other piece of information is required.[3] Assuming that the students adds the correct table and join conditions, Qr-Hint proceeds to the `WHERE` stage, whose goal is to ensure Q considers the correct subset of of input row combinations for further grouping and aggregation. There, Qr-Hint finds the repair `s1.price>s2.price` $\rightsquigarrow$ `s1.price>=s2.price`; based on this repair, the teacher may highlight `s1.price>s2.price` for the student to investigate further. Notably, even though hinting happens one stage at a time, Qr-Hint's repair algorithm does look *ahead* of the current stage, so it can avoid false alarms such as Q's extra Amy check in `WHERE` as discussed earlier.

Readers can refer to [8] for details on each stage of Qr-Hint, but we will briefly describe the `WHERE` stage to illustrate some technical challenges. Suppose we have already established the correct mapping among the `FROM` tables in Q and $Q^\star$ and "unified" their `WHERE` predicates P and $P^\star$ to use same variables for the same columns. Our goal is to find the smallest repair to P to make it logically equivalent to $P^\star$. For simplicity, consider the following sub-problem. Let P be represented as a syntax tree whose internal nodes are logical operators ($\wedge, \vee, \neg$) and leaves are atomic predicates. Suppose we are allowed to replace only one subtree of P with a new subtree x such the resulting tree is equivalent to $P^\star$. What is the smallest (in size) such x?

When written out, the problem appears to be one that solves for an unknown formula x given a logical statement involving x, e.g.: $A{=}1 \vee \cdots \vee (x \wedge B{>}2) \vee \cdots \Leftrightarrow P^\star$. Interestingly, it turns out that there are only two possibilities. One is that no x can make the statement true; i.e., the repair site is not feasible. The other possibility is a continuous "range" of solutions: any x such that $l_x \Rightarrow x \Rightarrow u_x$ will do, where l_x and u_x, which we call *lower* and *upper bounds* of x, are some logical formulae involving atomic predicates in P and $P^\star$. We have devised efficient procedures for testing whether a repair site is feasible, and if yes, computing the appropriate lower and upper bounds; see [8]. Then, the problem reduces to finding the simplest x "sandwiched" between two logical formulae. Figure 1 provides some geometric intuition of this optimization problem on a very simple example. Here, we would like to solve for x in $q \wedge x \Leftrightarrow P^\star$, where all predicates are over columns A and B. Geometrically, in two-dimensional space of all possible (A, B) values, each of the inequality atomic predicate involving either A or B corresponds to a halfspace, and p, $P^\star$, and x all can be seen as polyhedra formed by these halfspaces. The lower bound l_x (the least possible solution for x) in this case is the same as $P^\star$, the rectangle in the center, while the upper bound u_x (the most relaxed solution for x) is the $\dashv$ shape with a thick outline representing $P^\star \vee \neg p$. Any x between the lower and upper bounds is a solution because the portion of x outside $P^\star$ gets "clipped" by q in $q \wedge x$. We look for the simplest (i.e., fewest-sided) polyhedron sandwiched between l_x and u_x, which is the three-sided rectangular shape with a highlighter-style outline. Now, think of x as a Boolean function which variables representing satisfaction of individual atomic predicates. Each partition of the space induced by the halfspaces corresponding to these predicates (3×3 grid shown in Figure 1) is a minterm. Since $l_x \Rightarrow x \Rightarrow u_x$, x must set all partitions inside l_x to be true and all partitions outside u_x to be false; partitions in between are *don't-cares* (either true or false). Hence, the problem of finding the simplest x reduces to standard Boolean logic minimization with don't-cares, which Qr-Hint solves using the standard ESPRESSO tool [3].

We tested the coverage of Qr-Hint using real student queries from a database class at Duke University as well as queries crafted to cover additional common SQL issues identified in the literature [2]. Among the student queries, 11% contained SQL features that Qr-Hint currently does not support, such as outer-joins and general

[3] It may be feasible to use Generative AI to convert repairs to hints in natural language, a direction that we are still exploring.

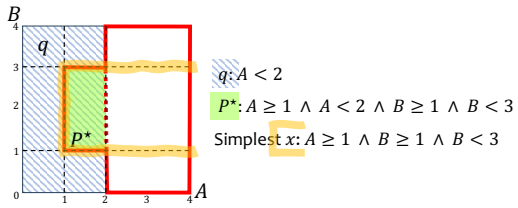

Figure 1: Example illustrating the geometric intuition for finding the smallest repair for x such that $q \wedge x \Leftrightarrow P^\star$.

subqueries, but Qr-Hint handled the rest perfectly, which covered the ten most common issues identified in [2]. We also stress-tested the speed of Qr-Hint using wrong queries generated by injecting errors into `WHERE` (which is the most expensive stage) of the TPC-H [16] benchmark queries. Running on a desktop, Qr-Hint was able to find single-site repairs under ten seconds with surgical precision. Finding optimal repairs involving two sites took two to three minutes, which leaves room for improvement but is already helpful to the teaching staff in helping students. For queries with three or more injected errors in `WHERE`, Qr-Hint was able to find reasonably good but sometimes non-optimal repairs under three minutes. While we have yet to deploy Qr-Hint in a classroom, our user study with 15 students showed that its hints generally helped debugging and are on par with (if not better than) hints manually produced by graduate teaching assistants. Details of the evaluation can be found in [8].

4 I-REX & D-DOXA: DEBUGGING WITHOUT A REFERENCE QUERY

While the systems in the previous sections are effective in helping students in a classroom setting, an assumption they make is the knowledge of a reference query $Q^\star$, which already correctly computes the intended result. Outside the classroom setting, however, we may not have such a luxury. The more general setting is when a user knows what the query should do, which oftentimes is or can be specified in natural language, but there is no $Q^\star$ specified in SQL or another formal language. The user may write a potentially incorrect query Q or use Generative AI to do so from a natural language specification. In either case, we are faced with the challenge of debugging Q without the help of a reference query $Q^\star$.

One ongoing project we are pursuing is a SQL debugger called I-Rex, an earlier version of which was demonstrated in [11] and deployed in our classrooms. As discussed in Section 1, if we care about correctness (rather than performance) of a query Q, debugging Q's execution plan is not helpful, as this plan may bear little resemblance to how Q is written. The key idea behind I-Rex is to provide an *illusion* of executing Q in a "literal" fashion based on how it is written. Specifically, a `FROM` clause corresponds to nested loops over the tables in the same order, with the table aliases acting as loop variables that iterate over rows from each table in a deterministic order. A conditional statement in the inner loop checks the `WHERE` condition. Uses of subqueries and common table expressions (`WITH`) can be seen as invoking functions returning tables, and correlated subqueries are simply parameterized functions that are invoked in the inner loop with specific parameter values derived from loop variables. Grouping, aggregation and projection, and ordering are treated as additional processing steps after `FROM` and `WHERE`. The execution is reproducible by design, down to processing order.

This literal approach to query execution makes it easy for the user to map execution to the query specification, and to convert insights gained in observing execution to fixes on the query. Because our execution approach conceptually employs traditional programming constructs such as loops, conditionals, and functions, it is friendly to users familiar with imperative programming but new to declarative querying. It also allows I-Rex to include features analogous to those found in popular debuggers of imperative languages. For example, users can step through query execution (and "into" correlated subqueries), and observe what complex expressions evaluate to for particular combinations of input rows (and values of correlated references). At the same time, the declarative nature of SQL allows I-Rex to provide novel debugging features more powerful than traditional debuggers. For example, users can fast-forward and fast-rewind execution at will; they can trace lineage among rows forward and backward in input, intermediate result, and final result tables; they can "pin" particular rows in these tables for investigation, which conceptually restricts execution an "input subspace" relevant only to the pinned rows.

One challenge in realizing I-Rex's literal approach to query execution is that it is far more costly than optimized execution plans. Our current I-Rex prototype only works on small database instances, but we are actively developing a new, scalable version of I-Rex for debugging on massive databases. The key insight is that at any point in time, the user is observing only a small portion of the overall execution. Thanks to advanced features such as fast forwarding/rewinding, lineage tracing, and pinning, most debugging sessions will likely not require observing the entire execution. Thus, while overall execution is inefficient, I-Rex aims to produce traces for just the *given portions* of the overall execution efficiently. We hope to roll out this new version of I-Rex in 2025.

Verifying Queries Without Knowing SQL. Finally, as a first step towards debugging SQL written by Generative AI, we have recently embarked on a system called D-Doxa that helps a user verify if a SQL query is semantically correct. We assume a setting different from I-Rex: the user has limited or no knowledge of SQL (but still understands how data is represented using tables). The user has in mind what the correct query should do and can describe it informally in natural language. Suppose a Generative AI tool offers a SQL query Q based on the user's prompts. D-Doxa then asks a series of questions to the user, with the goal of determining, in the end, whether Q correctly realizes what the user has in mind. The simplest form of question would be to present a concrete database instance to the user, such as the one in Table 1, and ask the user whether a particular result row of Q is indeed correct. Unfortunately, while concrete instances are good at showing that a query is *incorrect*, they are less effective (and impossible without some assumption on the form of correct queries) in proving that a query is correct, i.e., it returns the desired result for *all* possible database instances. D-Doxa's idea is to use abstract instances instead, such as the c-instance in Table 2. A user response on an abstract instance immediately generalizes to all concrete instances conforming to it,

which makes it possible to verify Q's correctness with fewer rounds of interactions.

As discussed at the end of Section 2, however, we are aware that more students prefer concrete instances to abstract ones. As we have done in CINSGEN [10], D-Doxa plans to combine both concrete and abstract instances: asking the user to confirm query result on an abstract instance in general, but still providing concrete instances as an aid. We are also hopeful that students will be trained to be more comfortable with abstract instances, as we adapt our curriculum to focus more on abstract reasoning and other skills that may become more valuable in the era of Generative AI.

5 CONCLUSION

In this extended talk abstract, we have presented some of our projects at Duke University motivated by how to teach databases more effectively and at scale. It is worth noting that besides pedagogical questions, this line of work has also opened up many interesting research questions with the field of databases itself (which this abstract focuses on). We argue that education-inspired research deserves more attention from the wider database research community, as the rise of Generative AI has called in question what and how to teach future researchers and practitioners. For those of us in academia, instead of lamenting that we do not have access to real "customer pain points" to inform our research agenda, we should recognize the unique opportunity we have to understand the needs of our students and make direct, consequential impacts. These students are our real users, and as our future generation, they are too important a group to ignore.

Acknowledgments. We thank the U.S. National Science Foundation for supporting *HNRQ: Helping Novices Learn and Debug Relational Queries* (IIS-2008107). Some of this work was done when Zhengjie Miao and Amir Gilad were at Duke University. Besides the authors, many graduate, undergraduate, and high school students have worked on the projects discussed in this abstract, including Alexander Bendeck, Tiangang Chen, Zian (Stephen) Chen, Jeremy Cohen, Kevin Day, Kushagra Ghosh, James Leong, Qiulin Li, James Lim, Jeffrey Luo, Allen Pan, Aanya Sanghavi, Sharan Sokhi, Aparimeya Taneja, and Zachary Zheng. We also thank all students of CompSci 316 and 516 at Duke University who have put up with our relentless experimentation and many not-so-stable releases over the years.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley. http://webdam.inria.fr/Alice/

[2] Stefan Brass and Christian Goldberg. 2006. Semantic errors in SQL queries: A quite complete list. *J. Syst. Softw.* 79, 5 (2006), 630–644. https://doi.org/10.1016/J.JSS.2005.06.028

[3] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. 1984. *Logic Minimization Algorithms for VLSI Synthesis.* The Kluwer International Series in Engineering and Computer Science, Vol. 2. Springer. https://doi.org/10.1007/978-1-4613-2821-6

[4] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[5] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[6] Amir Gilad, Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2022. Understanding Queries by Conditional Instances. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 355–368. https://doi.org/10.1145/3514221.3517898

[7] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, Leonid Libkin (Ed.). ACM, 31–40. https://doi.org/10.1145/1265530.1265535

[8] Yihao Hu, Amir Gilad, Kristin Stephens-Martinez, Sudeepa Roy, and Jun Yang. 2024. Qr-Hint: Actionable Hints Towards Correcting Wrong SQL Queries. In *Proceedings of the 2024 ACM SIGMOD International Conference o n Management of Data.* Santiago, Chile.

[9] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (1984), 761–791. https://doi.org/10.1145/1634.1886

[10] Hanze Meng, Zhengjie Miao, Amir Gilad, Sudeepa Roy, and Jun Yang. 2023. Characterizing and Verifying Queries Via CINSGEN. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 143–146. https://doi.org/10.1145/3555041.3589721

[11] Zhengjie Miao, Tiangang Chen, Alexander Bendeck, Kevin Day, Sudeepa Roy, and Jun Yang. 2020. I-Rex: An Interactive Relational Query Explainer for SQL. *Proc. VLDB Endow.* 13, 12 (2020), 2997–3000. https://doi.org/10.14778/3415478.3415528

[12] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 503–520. https://doi.org/10.1145/3299869.3319866

[13] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. RATest: Explaining Wrong Relational Queries Using Small Examples. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1961–1964. https://doi.org/10.1145/3299869.3320236

[14] Sudeepa Roy, Amir Gilad, Yihao Hu, Hanze Meng, Zhengjie Miao, Kristin Stephens-Martinez, and Jun Yang. 2024. How Database Theory Helps Teach Relational Queries in Database Education (Invited Talk). In *27th International Conference on Database Theory, ICDT 2024, March 25-28, 2024, Paestum, Italy (LIPIcs, Vol. 290)*, Graham Cormode and Michael Shekelyan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:9. https://doi.org/10.4230/LIPICS.ICDT.2024.2

[15] Stack Overflow. [n. d.]. 2023 Developer Survey. https://survey.stackoverflow.co/2023/#technology-programming-scripting-and-markup-languages-all-respondents

[16] Transaction Processing Performance Council. [n. d.]. The TPC-H Benchmark. http://www.tpc.org/tpch