How Database Theory Helps Teach Relational Queries in Database Education

Sudeepa Roy
Duke University, Durham, NC, USA
RelationalAI, Berkeley, CA, USA
(Visiting Scientist)

Amir Gilad
Hebrew University of Jerusalem, Israel

Yihao Hu
Duke University, Durham, NC, USA

Hanze Meng
Duke University, Durham, NC, USA

Zhengjie Miao
Simon Fraser University, Burnaby, Canada

Kristin Stephens-Martinez
Duke University, Durham NC, USA

Jun Yang
Duke University, Durham, NC, USA

Abstract

Data analytics skills have become an indispensable part of any education that seeks to prepare its students for the modern workforce. Essential in this skill set is the ability to work with structured relational data. Relational queries are based on logic and may be declarative in nature, posing new challenges to novices and students. Manual teaching resources being limited and enrollment growing rapidly, automated tools that help students debug queries and explain errors are potential game-changers in database education. We present a suite of tools built on the foundations of database theory that has been used by over 1600 students in database classes at Duke University, showcasing a high-impact application of database theory in database education.

2012 ACM Subject Classification Theory of computation → Database theory; Information systems → Data management systems; Information systems → Structured Query Language

Keywords and phrases Query Debugging, SQL, Relational Algebra, Relational Calculus, Database Education, Boolean Provenance

Digital Object Identifier 10.4230/LIPIcs.ICDT.2024.2

Category Invited Talk

Funding This work was partially supported by NSF awards IIS-2008107 and IIS-1552538.

Acknowledgements The authors are grateful to graduate and undergraduate students Alexander Bendeck, Stephen Chen, Tiangang Chen, Jeremy Cohen, Kevin Day, James Leong, Qiulin Li, James Lim, Jeffrey Luo, Allen Pan, Aanya Sanghavi, Sharan Sokhi, Aparimeya Taneja, and Zachary Zheng who contributed to building the tracing tool for SQL queries. This work was done when Zhengjie Miao (part of his PhD dissertation research) and Amir Gilad were at Duke University.

Extended Abstract

In a world where decisions are increasingly driven by data, data analytics skills have become an indispensable part of any education that seeks to prepare its students for the modern workforce, in particular, in the multi-billion dollar and rapidly-growing data analytics industry [7]. Essential in this skill set is the ability to work with *structured or relational data* in tabular form – such data can be queried directly to yield useful insights, or transformed into other representations for additional analysis, model training, or visualization. The standard "tools of trade" for manipulating structured data include the venerable and ubiquitous SQL language as well as popular data manipulation libraries, e.g., `dplyr` for R, `DataFrame` for Python `pandas`, and *Spark*. Despite differences in syntax, they are all fundamentally based on the *relational model* and rooted in relational query languages such as *Relational Calculus* (RC, similar to First Order Logic (FOL)) and *Relational Algebra* (RA). Learning and debugging relational queries, however, pose challenges to novices [2]. Even computer science students with programming background are often not used to thinking in terms of logic (e.g., when writing SQL queries) or functional programming (e.g., when writing queries using operators that resemble RA). In stark contrast to the plethora of educational tools for teaching traditional programming, there is a glaring lack of tools for helping novices learn and debug relational queries. The problem becomes more critical in a classroom setting, especially given the rapid growth of enrollment in database classes, and limited availability of manual help from instructors and teaching assistants in assisting students debug their queries. This motivates the need to build automated query debugging tools for database education that can give students succinct but comprehensive information about the mistakes they have made, and ideally also offer them advice or hints on how to resolve the mistakes without giving out the entire correct query.

Building tools for verifying, debugging, and fixing relational queries requires foundational research in database theory. Consider a scenario in a classroom setting where students are learning to write relational queries and suppose a student has submitted a query Q for a question they have been asked to solve. Since there are various tools for checking the correctness of the syntax of a query, we can assume that Q is syntactically correct. Suppose the instructor has a correct query Q_0 as reference for the same question. Since there can be multiple equivalent ways of writing the same query, ideally we want to check whether Q and Q_0 are equivalent. However, unfortunately, *"query equivalence"* testing is undecidable in general, for non-monotone RC or FOL queries involving universal quantifiers ($\forall$), for *Datalog* with recursion, and for practical query languages such as SQL with support for even just integer arithmetic [24, 25, 1, 5]. The problem is decidable but intractable for non-recursive monotone queries [6, 23], but that does not give a solution for verifying student submissions for general queries they need to learn in practice. "Eyeballing" errors manually by instructors and teaching assistants is difficult, especially for subtle mistakes, and does not scale in large classes. Therefore, the standard practice is to run both the student query Q and the reference query Q_0 on some test database instances D and compare their results, which is often done by "*autograding*" tools like GradeScope [11]. If the results differ, i.e., $Q(D) \neq Q_0(D)$, we know Q is wrong. Note that this does not test query equivalence, i.e., there is no guarantee that Q is correct if the results agree. For practical purposes, we resort to complex test instances that attempt to exercise conceivable corner cases in order to increase the chances of catching wrong queries.

Merely marking Q as wrong, however, does not guide students toward a correct solution. As a first step, *how do we explain to students "why" their query is wrong?* One option for the instructor to explain the errors in the student query Q is to show the test database instance D for which we know $Q(D) \neq Q_0(D)$ as a "*counterexample*", together with $Q(D)$ and $Q_0(D)$ (without revealing Q_0 itself). This approach may not work since D tends to be large and complex by design. For example, in the database courses at Duke University, one assignment is based on the real DBLP database with millions of rows; another assignment uses synthetic test databases, and we needed tens of thousands of rows in order to catch most of the errors that were manually found [17]. Showing millions, thousands, or even just dozens of database rows can overwhelm the student, especially when the student is learning to think about the solution logic as well as the query syntax and semantics for the first time. Further, revealing a test instance D in its entirety encourages the behavior of tweaking one's query just to pass

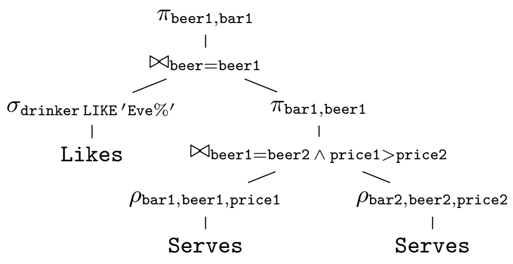
$\pi_{\texttt{beer1,bar1}}$
$\bowtie_{\texttt{beer=beer1}}$
$\sigma_{\texttt{drinker LIKE 'Eve\%'}}$
Likes
$\pi_{\texttt{bar1,beer1}}$
$\bowtie_{\texttt{beer1=beer2} \wedge \texttt{price1>price2}}$
$\rho_{\texttt{bar1,beer1,price1}}$
Serves
$\rho_{\texttt{bar2,beer2,price2}}$
Serves

Figure 1 Wrong query Q.

each particular test, which is not conducive to learning. Hence, we need to develop solutions to help students better understand what is wrong, and make it scale for a large number of students and easily customizable for different exercises across classrooms. The problem becomes more challenging when we consider different classes of relational queries, as the solutions for the procedural RA queries and declarative SQL queries may be very different.

At Duke University, we have been working on a project called *HNRQ: Helping Novices Learn and Debug Relational Queries* [27], where we are building a suite of tools for debugging queries. Furthermore, we are using these tools in our large undergraduate and graduate database classes to provide automated and scalable help to students debug their own queries. We are evaluating the effectiveness of these tools on learning by running user studies and surveys employing techniques from CS Education in consultation with the Institutional Review Board (IRB) at our university. These tools are built upon foundations and techniques from database theory, whereas they have simple user-friendly interactive graphical interfaces targeted towards students who are learning to write relational queries. This research direction showcase a practical application of database theory to help students learn to write relational queries and has a direct impact on any data science curriculum. Building query debugging tools has applications beyond the educational setting, e.g., to help debug database queries that fail regression tests commonly used by the software industry.

In the rest of the paper, we give a brief overview of our query debugging tools, and some ongoing and future research directions. We skip discussions of related work in this extended abstract; a detailed discussion of related work can be found in our research papers [17, 10, 16], in the articles in a Data Engineering Bulletin Special Issue on "*Widening the Impact of Data Engineering through Innovations in Education, Interfaces, and Features*" that Roy and Yang co-edited [22], and in the recent workshops "*Data Systems Education (DataEd)*" [21].

Explaining Wrong Queries with Small Counterexamples

In our first tool called **RATest** [17, 18], we focused on explaining wrong RA queries adapting the idea of using a test instance D where $Q(D) \neq Q_0(D)$. Instead of showing the entire test database instance D, the key idea behind RATest is to show a small subinstance $D' \subseteq D$ (still conforming to all database constraints like keys and foreign keys) such that $Q(D') \neq Q_0(D')$, i.e., D' is a small counterexample still able to illustrate the difference between Q and Q_0.

▶ **Example 1.** Consider the popular Drinker database with information about bars and bar-goers as follows (keys are underlined):

Drinker(<u>name</u>, address), Bar(<u>name</u>, address), Beer(<u>name</u>, brewer),
Frequents(<u>drinker</u>, <u>bar</u>, times_a_week), Likes(<u>drinker</u>, <u>beer</u>), Serves(<u>bar</u>, <u>beer</u>, price).

`name`	`addr`
Eve Edwards	32767 Magic Way

(a) Drinker.

`name`	`brewer`
American Pale Ale	Sierra Nevada

(b) Beer.

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

(c) Bar.

`drinker`	`beer`
Eve Edwards	American Pale Ale

(d) Likes.

`bar`	`beer`	`price`
Restaurant Memory	American Pale Ale	2.25
Restaurante Raffaele	American Pale Ale	2.75
Tadim	American Pale Ale	3.5

(e) Serves.

Figure 2 A small counterexample returned by RATest.

Students are asked to write the following query in RA: "*for each beer liked by any drinker with first name* `Eve`*, find the bars that serve this beer at the highest price.*" A common incorrect query Q is shown in Figure 1 that considers not-lowest price instead of highest price among other errors. The test database instance D for this database used in our database classes contains thousands of tuples and would not be useful to the student. RATest, on the other hand, is able to find a remarkably small counterexample in Figure 2 automatically to illustrate why Q is wrong. RATest also shows that Q returns both $\langle$`American Pale Ale`, `Restaurante Raffaele`$\rangle$ and $\langle$`American Pale Ale`, `Tadim`$\rangle$ on this small instance, whereas the correct result should contain only $\langle$`American Pale Ale`, `Tadim`$\rangle$. RATest further allows the student to trace the execution of Q over the counterexample along the RA query plan. The correct query Q_0 itself is never revealed.

In the backend, to simplify the problem of finding a small $D' \subseteq D$ further for efficiency such that $Q(D) \neq Q_0(D)$, we choose a result tuple $t \in Q_0(D) \setminus Q(D)$ (or $t \in Q(D) \setminus Q_0(D)$), and try to find a small instance $D' \subseteq D$ such that still $t \in Q_0(D') \setminus Q(D')$. If Q and Q_0 are **monotone**, we can solve this problem efficiently in polynomial time in the size of data (i.e., data complexity [26]). The problem becomes more interesting and challenging when Q_0 or Q is intrinsically **non-monotone**, then a non-answer can become an answer in a smaller database instance. In [17], we discussed both the data complexity and combined complexity (when both data size and query size are parameters) for different query classes: as soon as queries involve projection, join, and difference operations, even the data complexity becomes NP-hard. Nevertheless, in [17] we provided practical solutions for general RA queries with the difference operation. The intuitive idea is to compute the provenance [13] as a Boolean formula $\phi = \phi_1 \wedge \neg\phi_2$ for the tuple of interest, say $t \in (Q_0 - Q)(D)$, where ϕ_1, ϕ_2 denote its provenance or lineage [12] in $Q_0(D), Q(D)$ respectively (joint usage of two tuples by join $\bowtie$ is captured with $\wedge$, and alternative usage by projection π or union $\cup$ is captured by $\vee$). Then we find a minimum satisfying solution of ϕ by setting the smallest number of variables to true (*min-ones satisfiability problem*), which can be tackled by *SMT* (*satisfiability modulo theories*) solvers. We use an automatic rewrite procedure to convert the queries into SQL that will compute not only their results but also provenance expressions for the results tuples. We apply optimizations such as pushing down selections for interactive performance.

Suppose `Eve` *likes beer B,*
bar X_1 serves beer B at price P_1,
bar X_2 serves beer B at price P_2,
bar X_3 serves beer B at price P_3,
and $P_2 < P_1 < P_3$;
then your query would incorrectly return $\langle$ beer B, bar X_1 $\rangle$.

Figure 3 Intended generalization of the counterexample in Figure 2.

More challenges arise once we consider extended RA queries with **aggregation, grouping, and** `HAVING` (i.e., a selection after a group-by aggregation), for which the previous approach may not yield a small counterexample, because its aggregate (say `SUM`) value generally depends on all member tuples in the input group corresponding to t, and because of predicates like `HAVING Count(*) > 1000` that seek to output a database with at least 1000 tuples. Our solution is to *parameterize* the queries and allow a counterexample to differentiate two queries on *some* setting of the parameters (which may be different from the original setting) and constructing the provenance for aggregates using the approach by Amsterdamer et al. [4].

We built RATest as a web-based teaching tool, deployed it in an undergraduate database class at Duke university in Fall 2018 with about 170 students, and conducted a detailed *user study* with 10 homework problems where RATest was available only for a subset of the problems. We collected usage patterns on RATest as well as how students eventually scored on the homework problems. The usage of RATest was high in the class and more for harder queries, and RATest seemed to have helped students solve harder queries as well as other similar hard queries. We also collected 134 anonymous responses from the students and the feedback was largely positive. 69.4% of the respondents agreed or strongly agreed that the 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. Open-ended comments were overwhelmingly positive, 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."

Since Fall 2018, we have used RATest regularly in graduate and undergraduate courses at Duke with continued extensive use and positive feedback from students; till date RATest has been used **by more than 1600 students** at Duke University.

Explaining Wrong Queries with Abstract Conditional Instances

Since RATest was successful both in terms of research and practical uses in a classroom setting, the next step was improving this tool in terms of usability, generality, and deployability. For instance, in Example 1 and Figure 1, it can be noted that Q makes multiple mistakes. The counterexample in Figure 2 shows the mistake that for a given beer, Q actually finds bars that do not serve it at the lowest price (a monotone query), as opposed to bars that serve it at the highest price (requires a non-monotone query). The other mistake, where the predicate "`LIKE 'Eve%'`" may incorrectly pick up a drinker whose first name is `Evelyn`, is not illustrated in this counterexample. While this omission might be helpful for some students who can focus on one mistake at a time, it is also useful to show all the mistakes in the explanation for why Q is wrong. Further, Figure 2 does not "pinpoint" the mistake that the error is due to the presence of three distinct price values of the same beer, due to the

name	addr
d_1	$*$

(a) Drinker relation.

name	addr
x_1	$*$
x_2	$*$
x_3	$*$

(b) Bar.

bar	beer	price
x_1	b_1	p_1
x_2	b_1	p_2
x_3	b_1	p_3

(c) Serves.

name	brewer
b_1	$*$

(d) Beer.

drinker	beer
d_1	b_1

(e) Likes.

$\boxed{d_1 \text{ LIKE 'Eve␣\%' } \wedge p_1 > p_2 \wedge p_2 > p_3}$

(f) Global condition.

Figure 4 An abstract conditional instance generated by CINSGEN generalizing the concrete small instance in Figure 2 and capturing the intuitive explanation in Figure 3.

presence of redundant information in the form of attribute values in several other tuples in other tables. Ideally, we want an explanation as given in Figure 3, which would explain one major error why the query Q in Figure 1 is wrong. Moreover, the intrinsic limitation of an instance-based explanation starting with a test instance D is that it may not detect a wrong query. Further, in some cases, a good test instance D may be unavailable, especially when new queries are created on a new database schema.

Our next tool named **CINSGEN** [10, 15]) focused on addressing these issues. In [10] we considered queries in the form of Relational Calculus (the prototype [15] provided a procedure to convert SQL queries without aggregates to RC). The broad goal in CINSGEN was to understand all possible solutions to a query Q' by a set of "*generic*" and "*representative*" instances that (1) illustrate different ways the query Q' can be satisfied, and (2) summarize all specific instances that would satisfy the query in the same way by abstracting away unnecessary details. To formalize this, we develop the concept *conditional instances* or *c-instances* by adapting the notion of *c-tables* by Imilienski and Lipski [14] for incomplete databases, which are abstract database instances comprising variables (labeled nulls) along with a condition on those variables. An example c-instance capturing the intuitive condition in Figure 3 and generalizing the concrete small instance in Figure 2 is shown in Figure 4. Another c-instance (not shown) will illustrate the second error of the first name being "Eve" vs. the first name starting with "Eve". Thus, each c-instance can be considered a representative of all grounded instances that replace its variables with constants that satisfy the conditions that they are involved in. Since it may be hard to capture all satisfying instances with abstract c-instances (e.g., they can be unbounded in size), we use the idea of *coverage* from the software validation field [19, 20, 3], covering different ways a query can be satisfied. Since now we are essentially testing equivalence of two first order logic queries, the problem of finding such conditional instances in general is **undecidable** by a reduction from the *finite satisfiability problem* and Trakhtenbrot's Theorem [25]. Hence we developed practical algorithms inspired by the "*chase*" procedure by Fagin et al. [8, 9] with a user-specified input stopping condition (on the number of steps or time) to generate such instances. Hence, if CINSGEN does not return any c-instances, the query Q may still be wrong and not equivalent to Q_0. Our user study with undergraduate and graduate students shows that although both RATest [17] and CINSGEN [10] help students detect errors, conditional instances by CINSGEN help students detect multiple errors in wrong queries unlike concrete instances provided by RATest.

Tracing Outputs and Errors in Declarative SQL Queries

Part of convincing a student why a relational query Q is wrong is to help the student understand the semantics of Q and its behavior of on an input instance D. Even if D is small, it may not be obvious why Q produces $Q(D)$ as its result, especially for students who may have misunderstandings about how certain query constructs work. A useful way of debugging Q would be to "trace" its execution over D. For RA (or other operator-based relational languages), a straightforward approach, which we have used successfully in [17, 18], is to show the algebraic query expression tree and the intermediate results produced by each operator when executed in a bottom-up fashion. However, for the practical "declarative" language of SQL, which draws inspiration from logic, it is not even clear what "tracing" means. For instance, for a correlated subquery, we cannot talk about its result without the context from which it derives its variable bindings. Using the schema in Example 1, the following simple query has a correlated subquery:

```
SELECT address
FROM Drinker
WHERE EXISTS
(SELECT * FROM Frequents WHERE drinker = Drinker.name)
```

Clearly, the result of the subquery depends on the particular `Drinker.name` value, which comes from the outer query over `Drinker`. Hence, tracing intermediate results in a bottom-up fashion does not work. In practice, with or without correlated subqueries, SQL queries are almost never executed as the way they were written because of query optimization. Instead, we must be able to trace SQL at a *logical* level, in a way consistent with how a query is originally written, without requiring any additional knowledge of relational algebra or its mapping from SQL. Toward explaining how answers are generated from a SQL query, we developed a tracing tool named **I-REX** [16] that provides a novel interactive interface that allows users to trace query evaluation in a way faithful to how the query is written originally. For instance, I-Rex explains how the results are semantically generated (and non-answers are filtered out) by first computing the cross product of tables in the FROM clause, applying the predicates in the WHERE clause in a logical tree form, grouping intermediate results in the GROUP BY clause, applying predicate in the HAVING clause, and executing the subqueries in a "context" with specific variable bindings provided during the evaluation of its outer queries. In the backend, I-REX extends provenance support for SQL in non-trivial ways to work with various query constructs. This tool is currently being deployed in our classes and evaluations by user studies are being performed.

Ongoing and Future Work

When a student understands that their query is wrong, the next natural step is to provide some "*hints*" to fix their queries. While it is possible to quickly suggest students to restructure their approaches if the wrong queries are too far off from the solution, many wrong queries contain subtle mistakes which require potentially long time for examining and resolving errors, and instructors need to come up with good hints for fixing errors without showing the solution. Further, for wrong queries with smaller mistakes (e.g., missing a predicate in the WHERE clause), it is important to show the incremental changes that have to be made on the wrong query instead of suggesting a correct query that is drastically different from the student query. Toward this goal, we are developing a tool that takes two non-equivalent SQL queries (a correct query and a wrong query), pinpoints the parts of the wrong query that causes "*semantic*" differences between the queries, and also provides hints for direct edits to make it equivalent to the correct query. Our approach works for simple single-block SQL queries (without nested sub-queries, NULLs, constraints, etc.), and it will be interesting to develop methods for larger classes of SQL queries (e.g., two non-equivalent queries can become equivalent in the presence of certain integrity constraints).

A much more challenging and open-ended research direction is designing and building query debugging solutions in the realm of Large language Model (LLM)-based tools such as ChatGPT. These tools can return queries in standard relational query languages like SQL, but may produce incorrect solutions for complex queries. While questions like whether and how such tools should be allowed in database classes are being discussed in the community in various panels and workshops, the form of query debugging may be very different in the presence of LLMs, and may lead to a novel direction of research spanning multiple areas such as database theory, natural language processing, and programming languages.

Another important aspect in building query debugging tools is evaluating their effect on learning – e.g., whether they help students learn to write a query when such tools are not available, whether they inspire students to seek help more from instructors, or whether the students merely use them as a shortcut for getting their class assignments completed in less time. While we have been conducting extensive user studies and surveys in the large database classes at our university at different stages of development of these tools, there are restrictions on how such studies can be performed in classroom settings due to several compliance and ethical concerns. For instance, the gold standard of inferring causal conclusions by "*Randomized Controlled Trials*" to test whether these tools help students learn to write queries may not be feasible in an active class. Continued discussions and collaborations across universities will lead to fundamental database research as well as effective scalable solutions potentially revolutionizing database and data science education.

References

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

2 Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. A quantitative study of the relative difficulty for novices of writing seven different types of sql queries. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 201–206, New York, NY, USA, 2015. ACM. `doi:10.1145/2729094.2742620`.

3 Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

4 Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011. `doi:10.1145/1989284.1989302`.

5 Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database Theory*. Open source at `https://github.com/pdm-book/community`, 2022.

6 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 77–90, 1977. `doi:10.1145/800105.803397`.

7 Data Analytics Market Share, Market Research Future. `https://www.marketresearchfuture.com/reports/data-analytics-market-1689`.

8 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003. `doi:10.1007/3-540-36285-1_14`.

9 Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. In *PODS*, pages 90–101, 2003. `doi:10.1145/773153.773163`.

10 Amir Gilad, Zhengjie Miao, Sudeepa Roy, and Jun Yang. Understanding queries by conditional instances. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12–17, 2022*, pages 355–368. ACM, 2022. `doi:10.1145/3514221.3517898`.

11 Gradescope. `https://www.gradescope.com/`.

12 Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007. `doi:10.1145/1265530.1265535`.

13 Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007. `doi:10.1145/1265530.1265535`.

14 Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984. `doi:10.1145/1634.1886`.

15 Hanze Meng, Zhengjie Miao, Amir Gilad, Sudeepa Roy, and Jun Yang. Characterizing and verifying queries via CINSGEN. In Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia, editors, *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 143–146. ACM, 2023. `doi:10.1145/3555041.3589721`.

16 Zhengjie Miao, Tiangang Chen, Alexander Bendeck, Kevin Day, Sudeepa Roy, and Jun Yang. I-rex: An interactive relational query explainer for SQL. *Proc. VLDB Endow.*, 13(12):2997–3000, 2020. `doi:10.14778/3415478.3415528`.

17 Zhengjie Miao, Sudeepa Roy, and Jun Yang. Explaining wrong queries using small examples. In *SIGMOD*, pages 503–520, 2019. `doi:10.1145/3299869.3319866`.

18 Zhengjie Miao, Sudeepa Roy, and Jun Yang. Ratest: Explaining wrong relational queries using small examples. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1961–1964. ACM, 2019. `doi:10.1145/3299869.3320236`.

19 Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963. `doi:10.1145/366246.366248`.

20 Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley, 2004.

21 International Workshop on Data Systems Education (DataEd) 2023.

22 Sudeepa Roy and Jun Yang. Letter from the special issue editor. *IEEE Data Eng. Bull.*, 45(3):2–3, 2022. URL: `http://sites.computer.org/debull/A22sept/p2.pdf`.

23 Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, oct 1980. `doi:10.1145/322217.322221`.

24 Oded Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, feb 1993. `doi:10.1016/0743-1066(93)90040-N`.

25 Boris Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. *Proceedings of the USSR Academy of Sciences*, 70(4):569–572, 1950.

26 Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 137–146, 1982. `doi:10.1145/800070.802186`.

27 Project website:. `https://dukedb-hnrq.github.io/`.