Characterizing and Verifying Queries Via CinsGen

Hanze Meng, Duke University, hm222@cs.duke.edu

Zhengjie Miao*, Megagon Labs, zhengjie@megagon.ai

Amir Gilad, Duke University, agilad@cs.duke.edu

Sudeepa Roy, Duke University, sudeepa@cs.duke.edu

Jun Yang, Duke University, junyang@cs.duke.edu

ABSTRACT

Example database instances can be very helpful in understanding complex queries. Different examples may illustrate alternative situations in which answers emerge in the query results and can be useful for testing. Examples can also help reveal semantic differences between queries that are supposed to be equivalent, e.g., when students try to understand how their queries behave differently from a reference solution, or when programmers try to pinpoint mistakes inadvertently introduced by rewrites meant to improve readability or performance. In this paper, we propose to demonstrate CinsGen, a system that can characterize queries and help distinguish between two queries. Given a query, CinsGen generates minimal conditional instances (c-instances) that satisfy it. In turn, each c-instance is a generalization of multiple database instances, yielding a compact representation. Thus, using CinsGen enables users to obtain a comprehensive and compact view of all scenarios that satisfy a specified query, allowing for query characterization or distinction between two queries.

CCS CONCEPTS

• Theory of computation → Incomplete, inconsistent, and uncertain databases; **• Information systems →** *Relational database query languages*; **Database utilities and tools**.

KEYWORDS

database usability, incomplete databases

ACM Reference Format:

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-Companion '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3555041.3589721

name	addr
Eve Edwards	32767 Magic Way

(a) Drinker relation

name	brewer
American Pale Ale	Sierra Nevada

(b) Beer relation

name	addr
Land & Leute	1276 Evans Estate
Tadim	082 Julia Underpass
Algarve	7357 Dalton Walks

(c) Bar relation

drinker	beer
Eve Edwards	American Pale Ale

(d) Likes relation

bar	beer	price
Land & Leute	American Pale Ale	2.25
Algarve	American Pale Ale	2.75
Tadim	American Pale Ale	3.5

(e) Serves relation

bar	beer
Algarve	American Pale Ale
Tadim	American Pale Ale

(f) Result of Q_B

bar	beer
Tadim	American Pale Ale

(g) Result of Q_A

Figure 1: Database instance K_0 of the Beers dataset. We assume natural foreign key constraints from Serves and Likes to Drinker, Bar, Beer.

$$Q_A = \{(x_1, b_1) \mid \exists d_1, p_1 (\text{Serves}(x_1, b_1, p_1) \wedge \text{Likes}(d_1, b_1) \wedge d_1 \text{ LIKE 'Eve\textvisiblespace\%'} \wedge \forall x_2, p_2 (\neg \text{Serves}(x_2, b_1, p_2) \vee p_1 \geq p_2)\}$$

(a) *Query Q_A: for each beer liked by any drinker whose first name is Eve, find the bars that serve this beer at the* highest *price*

$$Q_B = \{(x_1, b_1) \mid \exists d_1, p_1 (\exists x_2, p_2 (\text{Serves}(x_1, b_1, p_1) \wedge \text{Likes}(d_1, b_1) \wedge d_1 \text{ LIKE } 'Eve\%' \wedge \text{Serves}(x_2, b_1, p_2) \wedge p_1 > p_2)\}$$

(b) *Query Q_B which is similar to Q_A but does not use the difference operator and instead, find beers served at a* non-lowest *price*

Figure 2: Correct query Q_A and incorrect query Q_B. Note that the formula in Q_A has a space after 'Eve' whereas Q_B does not. Here and later, ␣ denotes the space symbol.

1 INTRODUCTION

Data analytics is indispensable in today's technological environment, making the ability to query database management systems (DBMS) one of the core skills in various fields. The need for tools to support DBMS users in understanding database queries by examining how the query executes on certain database instances has been extensively explored by the database community [3, 5, 8]. A substantial part of these focuses on the provenance of the query results, based on which the tools provide users with different combinations of input tuples in the database and illustrate how the input tuples satisfy the query.

Although existing provenance-based tools are shown to be effective in explaining how the given query generates certain outputs (often used in query debugging), these tools are highly dependent on the given database instances. Hence, such tools may lead users to focus on specific details in the given database instance but fail to yield a general picture of the query features, i.e., what, in general, leads to the satisfaction of the query. In particular, instances that are not given may reveal other ways to satisfy the query.

Even if one can have an ideal test instance and can use existing tools to find multiple different database instances, there can be infinitely many database instances that satisfy the given query or pinpoint issues in the query. In this case, the DBMS user would

*Part of the work was done when the author was a Ph.D. student at Duke University.

SIGMOD-Companion '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 978-1-4503-9507-6/23/06…$15.00
https://doi.org/10.1145/3555041.3589721

$$Q_B - Q_A = \{(x_1, b_1) \mid \exists d_1, p_1 (\exists x_2, p_2 (\text{Serves}(x_1, b_1, p_1) \wedge \text{Likes}(d_1, b_1) \wedge d_1 \text{ LIKE } 'Eve\%' \wedge \text{Serves}(x_2, b_1, p_2) \wedge p_1 > p_2) \wedge \forall d_2, p_3 (\neg \text{Likes}(d_2, b_1) \vee \neg (d_2 \text{ LIKE 'Eve\_\%'}) \vee \neg \text{Serves}(x_1, b_1, p_3) \vee (\exists x_3, p_4 (\text{Serves}(x_3, b_1, p_4) \wedge p_3 < p_4)))\}$$

Figure 3: The difference query $Q_B - Q_A$ from Figure 2.

name	addr
d_1	*

(a) Drinker relation

name	addr
x_1	*
x_2	*
x_3	*

(b) Bar relation

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

(c) Serves relation

name	brewer
b_1	*

(d) Beer relation

drinker	beer
d_1	b_1

(e) Likes relation

d_1 LIKE 'Eve%' $\wedge p_1 > p_2 \wedge p_2 > p_3$

(f) Global condition

Figure 4: C-instance $\mathcal{I}_0$ that satisfies $Q_B - Q_A$ and generalizes the counterexample K_0 in Figure 1.

expect to see "clusters" of these instances instead of seeing all of the instances.

To this end, we propose CinsGen[1], a system that generates a set of conditional instances or c-instances that satisfy a given query. We adapt the notion of c-tables [10] from incomplete databases. Such instances can contain variables instead of only constants and assert logical conditions involving those variables. Thus, each c-instance can be considered as a representative of all grounded instances that replace its variables with constants satisfying the conditions they are involved in. We also use the idea of *coverage* from software validation [2] to capture different ways that database instances satisfy the query. When a DBMS user examines how their query executes, they will find that a specific ground instance satisfying a certain subset of the query parts is sufficient to satisfy the query. Therefore, we refer to the subset of the query atoms as the coverage of the ground instance. CinsGen can provide a compact representation of all satisfying instances without relying on a specific database instance.

Example 1. *Consider the database K_0 shown in Figure 1 containing information about drinkers (Drinker), beers (Beer), bars (Bar), which beer does a drinker like (Likes), and which bar serves which beer (Serves). Suppose that a student is asked to write a query to find the bars that serve the most expensive beer liked by any drinker whose first name is Eve. A correct solution Q_A written in Domain Relational Calculus (DRC) is shown in Figure 2a, while the student may write a very similar but different query Q_B (in Figure 2b), which chooses bars serving beers not at the lowest price and only requires first names to have a prefix of 'Eve'. Figure 3 shows the formula for $Q_B - Q_A$ but is not easily understandable and does not clearly show the difference between the queries. In this case, using provenance-based tools and a reasonable test database instance, we can find the minimum counterexample K_0 (shown in Figure 1) for the difference between Q_A and Q_B [11]. In particular, Q_B returns the tuples with non-lowest prices, (Algarve, American Pale Ale) and (Tadim, American Pale Ale), while Q_A only returns the latter tuple – the bar with the highest price. Notice that the actual price and other values in K_0 are unimportant – as long as there exist three different prices in the database, the Q_B would return the bar with non-lowest and non-highest prices. Now consider the more general counterexample as a* c-instance *showing the differences between the queries Q_B and Q_A in Figure 4. This c-instance, $\mathcal{I}_0$, shows abstract tuples with variables instead of constants (* are 'don't care' variables) and a condition that the variables must satisfy (there should be a drinker whose name is 'Eve' with a space after and the order of the prices in Serves table should be $p_1 > p_2 > p_3$). Thus, $\mathcal{I}_0$ not only generalizes the counterexample in Figure 1 (i.e., there exists an assignment to the variables that results in the instance in Figure 1 and satisfies the global condition), but, also specifies the 'minimal' condition for which Q_B differs from Q_A (the global condition). K_0 in Figure 1 contains specific values that may confuse the user and divert attention from the core differences.*

[1] The research paper that developed the approach used by CinsGen appeared in SIGMOD 2022 [7].

Extensions of [7] for usability. While our algorithms are designed to work with DRC queries and our output is in the form of c-instance, in our implementation, we make CinsGen more accessible and its results more easily understandable. In particular, we recognize that writing queries in DRC may be out of reach for most users. We have, therefore, added a novel translation component that allows CinsGen to get SQL queries and automatically convert them into DRC, which is the input to our algorithms. The translation component takes as input the query plan generated by I-Rex [9], creates a distinct variable for each column reference in the query plan, and constructs DRC tree nodes according to specific rules by recursively tracing down the query plan. Another feature added to CinsGen is the instantiation of c-instances. Now, users are able to choose a c-instance that was generated by our algorithm, instantiate it with values from the appropriate domains, and get a concrete database instance that satisfies the query. CinsGen further evaluates the query over this instance and presents the results, making the c-instances easier to understand and interpret.

We will demonstrate CinsGen with real-world datasets and allow conference participants to explore different queries, the c-instances generated from them, and the resulting concrete instances that satisfy their queries. Thus, participants will experience an additional tool for characterizing complex queries and distinguishing between similar queries.

2 TECHNICAL BACKGROUND

We consider queries in Domain Relational Calculus (DRC), which is equivalent to Relational Algebra [4].

Given a schema $\mathbf{R}$, a DRC query Q is expressed as $Q = \{(x_1, x_2, ..., x_p) \mid \mathrm{P}_Q(x_1, ...x_p)\}$ where each x_i represents a query variablethat can only be assigned of variables or constants in its domain, P_Q is a standard first order logic (FOL) formula [1] involving relation names, constants, and domain variables. The formula P_Q is built from DRC atoms of the following forms: (1) $R(y_1..., y_k)$ or $\neg R(y_1..., y_k)$, where $R \in \mathbf{R}$ is a relation, and each y_i is a query variable or a constant, and (2) conditions x_1 *op* x_2 or x_1 *op* c, where x_1, x_2 are variables in the query, c is a constant in the domain, and *op* is a binary operator such as $=, >, \geq, <, \leq, \neq$, *LIKE*.

C-instance. We give the definition of a c-instance adapting the concepts of c-tables from the literature [10]. A conditional table (*c-table*) with a relational schema $R_i \in \mathbf{R}$ is a table T_i in which for each tuple $t \in T_i$ and each attribute $\mathsf{A} \in \mathtt{Attr}(R_i)$, $t[\mathsf{A}]$ is either a constant from its active domain $\text{Dom}(\mathsf{A})$ or is a labeled null. A

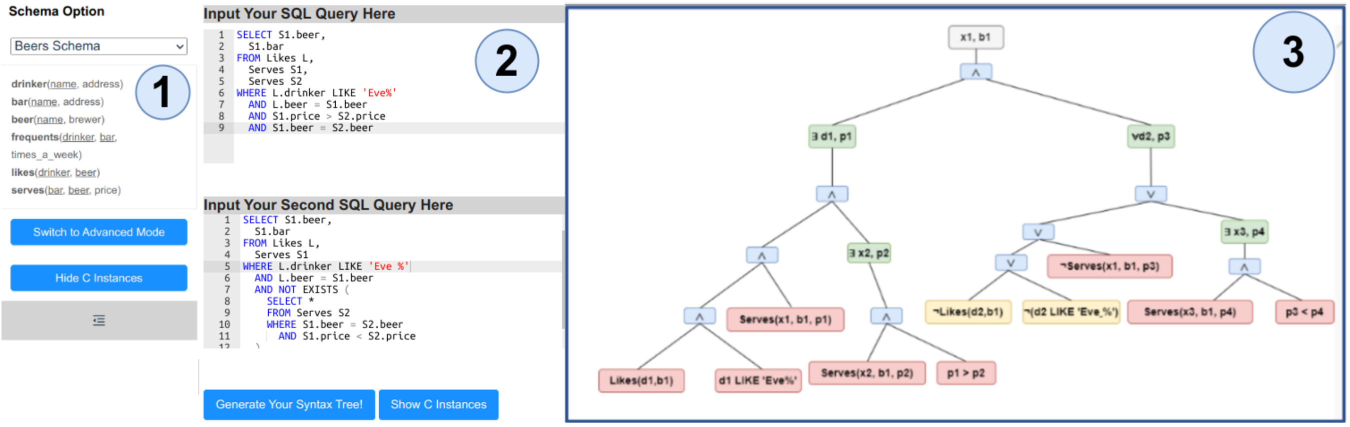

Figure 5: The input screen of CinsGen showing our running example.

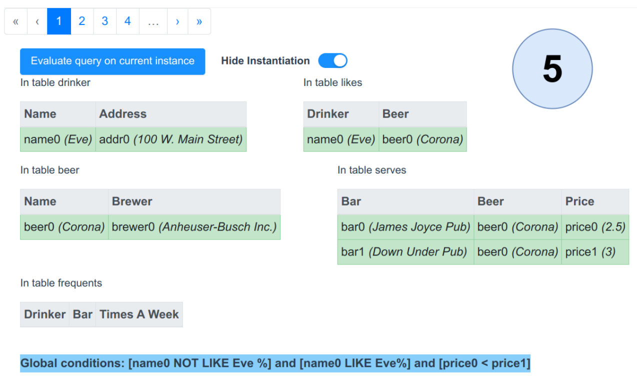

Figure 6: Instantiation screen.

c-instance $\mathcal{I}$ of **R** is a tuple of the form $(\{\mathsf{T}_1, \ldots, \mathsf{T}_r\}, \phi)$, where for each $i \in [1, r]$, T_i is a c-table with schema R_i, and the global condition ϕ is a conjunction of atomic conditions associated with the c-instance. The atomic conditions in the c-instance are either (1) an atom of the form $[x\ op\ c]$ $(\neg[x\ op\ c])$ or $[x\ op\ y](\neg[x\ op\ y])$ where x and y are labeled nulls, c is a constant in the active domain, and op is a binary operator, or (2) a condition of the form $\neg R(x_1, \ldots, x_k)$ where R is a relation on k attributes.

Query syntax tree. A syntax tree of a query Q is tree for the FOL formula P_Q satisfying the following rules: (1) each leaf node is a DRC atom, and (2) each internal node is either a quantifier with a single variable (e.g., $\forall x$ and $\exists x$) with a single child, or a connective ($\wedge$ and $\vee$) with two children. Further, all negations in the syntax tree appear in the leaves; we do not use separate nodes for negation. Figure 5 shows the syntax tree of the difference query in Figure 3.

Coverage. Given a query Q, In this work, we want to find c-instances instead of ground instances that satisfy Q. To measure how a database instance satisfies a query or how it distinguishes two queries, we propose to use the subset of query atoms satisfied when evaluating the queries on the instance, which we call the *"coverage"* of an instance. Intuitively, the coverage $\mathrm{cov}(Q, \mathcal{I})$ is the set of atoms and conditions of Q that can be covered by any ground instance of the c-instance $\mathcal{I}$, eventually leading to the satisfaction of Q. In the syntax tree of Q, the coverage can be seen as the subset of leaves that are satisfied by $\mathcal{I}$. The coverage of $\mathcal{I}_0$ (Figure 4) is shown by the red leaves in Figure 5.

Query characterization. With the notion of coverage, for a query Q and a given set of leaves L, the *query characterization* problem is to find a set of c-instances $\mathcal{S}_\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$, such that for all $\mathcal{I}_i$, $\mathcal{I}_i$ satisfies Q, $\mathcal{I}_i$ is minimal (no other satisfying c-instances with fewer tuples/conditions have the same coverage), and each $\mathcal{I}_i$ covers a subset of L in the syntax tree. Ideally, the solution $\mathcal{S}_\mathcal{I}$ should be *complete*, i.e., for any satisfying grounded instance K with coverage C such that $\mathrm{C} \cap L$ is a maximum subset of L that can be covered, there is a $\mathcal{I}_i \in \mathcal{S}_\mathcal{I}$ with $\mathrm{C} = \mathrm{cov}(Q, \mathcal{I}_i)$. Also, the ideal solution should have *no redundancy*, i.e., for any two $\mathcal{I}_i, \mathcal{I}_j$ where $i \neq j$, $\mathrm{cov}(Q, \mathcal{I}_i) \neq \mathrm{cov}(Q, \mathcal{I}_j)$. Intuitively, these c-instances comprise a minimal set to characterize all possible ways that Q is satisfied.

3 SYSTEM IMPLEMENTATION

The interface of CinsGen is implemented in Flask where the optional database is stored in PostgreSQL. The algorithms used to translate SQL to DRC and generate the c-instances are implemented in Python 3.7 and use an SMT solver [6].

Translating SQL to DRC. To translate the queries written by the user from SQL to DRC, we first employ the I-Rex system [9] to obtain a JSON file containing the query plan in a specific format. This representation is an internal intermediate step in I-Rex. Then, the query plan is parsed, and the DRC syntax tree and query are built in a bottom-up fashion, starting from the atoms and conditions and moving to quantifiers ($\forall, \exists$) and connectors ($\wedge, \vee$). Meanwhile, it creates a distinct variable for each column reference in the query plan and keeps track of the variables bounded by existential and universal quantifiers respectively.

Building C-instances. Next, we compute the set of satisfying c-instances for the query for a given coverage. In [7], we show that this problem is undecidable. So, inspired by the chase procedure in data exchange, we provide search-based heuristics to build such c-instances. At a high level, our algorithm tries to "map" the leaf atoms and conditions in the DRC tree to tuples and conditions being added to the c-instances. It keeps adding tuples and conditions by repeatedly traversing the tree and enumerating possible assignments of quantified variables until the resulting c-instance is consistent and satisfies the query (checked using an SMT solver). In particular, handling $\vee$ and $\forall$ nodes in the tree increases the complexity. For a tree rooted at a $\vee$ node ($Q = Q_1 \vee Q_2$), the algorithm reduces it into three conjunctive trees by considering $Q_1 \wedge Q_2$, $Q_1 \wedge \neg Q_2$, and $\neg Q_1 \wedge Q_2$. For each reduced case, the algorithm may obtain a set of c-instances and will return all of them as the result. For a tree rooted at a $\forall$ node, the algorithm maps the quantified variable x to different labeled nulls and constants and merges all resulting c-instances into one single c-instance.

Instantiating c-instances. The resulting c-instances given by our algorithm may contain labeled nulls that are denoted with identifiers that are combinations of letters and numbers. To provide the users with a more tangible result, CinsGen also has the option to instantiate c-instances with concrete values. To achieve this, CinsGen loads the domain of each attribute in the dataset (it can also discovers the active domain from a loaded database instance). Using this data, CinsGen employs an SMT solver to find a valid assignment to the labeled nulls in the c-instance. If there is no source of user-provided active domain or there are no available values in the database that lead to a valid assignment, CinsGen can use the solver to generate values satisfying the conditions.

4 DEMONSTRATION SCENARIO

Our demonstration will employ the Beers dataset, a sample of which is shown in Figure 1, and the DBLP dataset. We will begin with an initial explanation of the input screen, the different options for dataset selection, and the use of the query input boxes. We will then give a detailed example of running the various steps in CinsGen using Example 1.

Step 1: Dataset selection. Users start by choosing one of the pre-loaded databases in CinsGen (Beers and DBLP) and familiarizing themselves with the schema of the selected database (displayed on the left side of the screen in Figure 5, with keys in each table underscored).

Step 2: Query formulation. Next, users will utilize the query fields in Figure 5 to formulate their query in SQL. Our algorithms in CinsGen will automatically translate the query to DRC (see Section 3). Additionally, users can provide a second query as a reference query that they wish to distinguish from the first one. CinsGen will then find c-instances to differentiate them. As mentioned in Example 1, this scenario is particularly useful when users want to examine two similar queries that may be equivalent, or in a classroom setting when TAs wish to check a student query and give the students instances for which the queries differ.

Step 3: Choice of covered nodes in the syntax tree. Upon clicking the "Generate Your Syntax Tree!" button (located at the bottom of the screen shown in Figure 5), users will see the syntax tree of their query (if a single query was given), or the syntax tree of the difference query (if two queries were given). The user can then examine the structure of the query, which can be crucial for novice users like students to understand their queries. Moreover, for more experienced users such as instructors and TAs, CinsGen provides an advanced mode: in the view of the DRC syntax tree, the users can annotate the leaves that they want to be covered by the c-instances simply by clicking on them. Besides offering users a flexible interface to explore how their query evaluates, this feature narrows down the search space of CinsGen in the c-instance generation process. As a result, CinsGen will only generate c-instances that satisfy a maximum size subset of the annotated atoms in the leaves.

Step 4: C-instance generation. When clicking the "Show C Instances" button in Figure 5, CinsGen will generate the requested c-instances if leaf nodes were selected in the previous step, or run an exhaustive search to find all satisfying c-instance if no leaf node was selected. The resulting c-instances will be displayed below the query field on the user interface of CinsGen, as depicted in Figure 6. In this view, users can review the generated c-instances one by one by navigating through the pagination row using the arrows in the top left corner of Figure 6.

Step 5: Instance instantiation and evaluation. To provide a more concrete view of the c-instances for standard users such as students, CinsGen will generate concrete values for each labeled null in the c-instance. Specifically, CinsGen uses the domains of the different attributes in the database to complement the identifiers in the c-instance (e.g., name0 and beer0 in Figure 6) with values from the domain (e.g., Eve and Corona in Figure 6) in a way that ensures the assignment is consistent and satisfy the global condition, as explained in Section 3. However, more experienced users can choose not to instantiate the c-instance in the advanced mode if they prefer to examine c-instances without concrete values. Furthermore, the results of evaluating the query (or both user-input queries in case two queries were given) over this instance will also be shown to the user, explicitly indicating whether the instance satisfies the given query or can distinguish between the two given queries.

Users can then further interact with CinsGen by modifying their initial query, adding a second query if one was not provided, annotating different leaves in the syntax tree, and choosing a different c-instance to instantiate.

ACKNOWLEDGMENTS

This work was partially supported by the NSF grants IIS-2008107 and IIS-2147061.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*.
[3] 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.
[4] Edgar F Codd et al. 1972. *Relational completeness of data base sublanguages*.
[5] Yingwei Cui and Jennifer Widom. 2003. Lineage tracing for general data warehouse transformations. *The VLDB Journal—The International Journal on Very Large Data Bases* 12, 1 (2003), 41–58.
[6] 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.
[7] Amir Gilad, Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2022. Understanding Queries by Conditional Instances. In *SIGMOD*. 355–368.
[8] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. 31–40.
[9] Yihao Hu, Zhengjie Miao, Zhiming Leong, Haechan Lim, Zachary Zheng, Sudeepa Roy, Kristin Stephens-Martinez, and Jun Yang. 2022. I-Rex: An Interactive Relational Query Debugger for SQL. In *SIGCSE*. 1180.
[10] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (1984), 761–791.
[11] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *SIGMOD*. 503–520.