Understanding Queries by Conditional Instances

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

Zhengjie Miao*
Duke University
zjmiao@cs.duke.edu

Sudeepa Roy
Duke University
sudeepa@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

A powerful way to understand a complex query is by observing how it operates on data instances. However, *specific* database instances are not ideal for such observations: they often include large amounts of superfluous details that are not only irrelevant to understanding the query but also cause cognitive overload; and one specific database may not be enough. Given a relational query, is it possible to provide a simple and *generic* "representative" instance that (1) illustrates how the query can be satisfied, (2) summarizes all specific instances that would satisfy the query in the same way by abstracting away unnecessary details? Furthermore, is it possible to find a collection of such representative instances that together completely characterize all possible ways in which the query can be satisfied? This paper takes initial steps towards answering these questions. We design what these representative instances look like, define what they stand for, and formalize what it means for them to satisfy a query in "all possible ways." We argue that this problem is undecidable for general domain relational calculus queries, and develop practical algorithms for computing a minimum collection of such instances subject to other constraints. We evaluate the efficiency of our approach experimentally, and show its effectiveness in helping users debug relational queries through a user study.

1 INTRODUCTION

A powerful way to understand a complex query is by observing how it operates on data instances. A further in-depth approach may also consider the provenance of the query [10, 17, 28] as an indicator how different combinations of tuples in the database satisfy the query and generate each result. Another approach [41] finds a minimal satisfying instance of the query. However, these characterizations are highly dependent on the given database instance, even when provenance is employed. In addition, such database instances can contain many details that divert attention from the query features themselves. For example, some queries are satisfied by an empty instance, but there may be other satisfying instances that are not trivial. Thus, some parts of the query may be ignored since they are not satisfied by the instance. Furthermore, the evaluation on an instance leads to a satisfaction of specific combination of query atoms, but a different combination of atoms that is not satisfied by the specific database instance may reveal new insights. While there are approaches to find a query solution without a given database instance [15], they provide only one way to satisfy a query, thereby again possibly missing different paths toward satisfying the query.

On the other hand, a single query can have infinitely many satisfying instances so showing all of them will not just be confusing, it may be impossible. Therefore, to understand all possible solutions to a query, we study the question of whether it is possible to provide a simple and *generic* "representative" instance that (1)

illustrates how the query can be satisfied, and (2) summarizes all specific instances that would satisfy the query in the same way by abstracting away unnecessary details. Further, we ask how we can find a collection of such representative instances that together completely characterize all possible ways to satisfy the query.

To answer these questions, we propose a novel approach of understanding queries based on *conditional instances* or *c-instances*, by adapting the notion of *c-tables* [32] from the literature on incomplete databases, which are abstract database instances comprising variables (labeled nulls) along with a condition on those variables. Thus, each c-instance can be considered a representative of all grounded instances that replace its variables with constants that satisfy the conditions they are involved in. However, it may be impossible to capture all satisfying instances with a single c-instance. Therefore, we use the idea of *coverage*, borrowed from the field of software validation [5, 42, 43], where it has been well-studied in the context of software testing. For example, a test suite is said to *cover* a function if the function is invoked during the test. This idea can be abstracted to program flows, where an edge/branch in the control-flow graph (see [4] for details) is said to be covered if the edge/branch has been executed. For our use, when given a query Q and a satisfying c-instance $\mathcal{I}$, the atoms and conditions of Q that are satisfied by all ground instances that $\mathcal{I}$ represents are said to be covered by $\mathcal{I}$. We intend to find a set of c-instances such that for every grounded instance that satisfies Q with some coverage C, we have a c-instance that satisfies Q with the same coverage C.

The idea of providing a compact representation of all instances that satisfy a query is appealing not just from a theoretical perspective, but also for multiple practical reasons.

- First, this approach can be used for explaining why a given (wrong) query is different than another (correct) one, as studied in works on counterexamples [15, 41]. In this scenario, we are given a query Q_1 and another query Q_2, the output is an instance K such that $Q_1(K) \neq Q_2(K)$ (i.e., K is a satisfying instance for $Q_1 - Q_2$ or $Q_2 - Q_1$). In an educational setting, such instances would help instructors and students understand why a query is wrong and debug it, without revealing the correct query to students.
- Second, developers and data scientists who work with complex queries can this approach to explore how various parts of the queries can be triggered by different data, or to help them debug or refine these queries. For example, if there are no instances that can trigger some part of a query, it may be possible to simplify the query to remove "dead code" that logically contradicts other necessary conditions in the query.
- Third, this approach offers a method for generating a suite of test instances for a complex query such that together they "exercise" all parts of the query. In the field of synthetic data generation, previous works have proposed different approaches to generating data for testing workload queries [9, 37, 47]. Using our approach,

---

*Both authors contributed equally to this research.

name	addr
Eve Edwards	32767 Magic Way

(a) Drinker relation

name	brewer
American Pale Ale	Sierra Nevada

(b) Beer relation

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

(c) Bar relation

drinker	beer
Eve Edwards	American Pale Ale

(d) Likes relation

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 relation

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

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

(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, $\lrcorner$ denotes the space symbol.

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

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

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

(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.

given a workload query Q, we can generate a set of instances to provide coverage testing for all parts of Q; furthermore, given a set of workload queries, we can generate test instances where a given subset of queries are satisfied but others are not. These generated instances can be used for automated, comprehensive testing of queries.

We illustrate the first use case above with an example below.

EXAMPLE 1. *Consider the database K_0 shown in Figure 1 containing information about drinkers (Drinker table), beers (Beer table), bars (Bar table), which beer does a drinker like (Likes table), and which bar serves which beer (Serves table). Also consider the queries Q_A and Q_B written in Domain Relational Calculus (DRC) in Figures 2a and 2b, respectively. The correct query Q_A returns a list of bars that serve the*

most expensive beer liked by any drinker whose first name is 'Eve', whereas Q_B is a very similar query that 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, Figure 1 gives the minimum counterexample K_0 for the difference between Q_A and Q_B [41]. In particular, Q_B returns the tuples (Restaurante Raffaele, American Pale Ale) and (Tadim, American Pale Ale) while Q_A only returns the latter tuple.

Now consider the more general counterexample as a c-instance (defined in the next section) showing the differences between the queries $Q_B - Q_A$ 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, it also specifies the 'minimal' condition for which Q_B differs from Q_A (the global condition). The ground instance in Figure 1 contains specific values that may confuse the user and divert attention from the core differences. This is one of the c-instances in our universal solution that includes three c-instances. Each of the c-instances captures a facet of the difference between Q_B and Q_A.

Our contributions. Our contributions are summarized below.

- We propose a framework for characterizing all query answers using c-instances using the notion of *coverage* and a *universal solution* that captures different ways a given DRC query can be satisfied.
- We argue that deciding whether a universal solution or even any satisfying c-instance exists is undecidable for general DRC queries, by giving a reduction from the finite satisfiability problem for First Order Logic formulas [50]. However, for the class of conjunctive queries with negation $CQ^{\neg}$, the universal solution can be found in poly-time in the query size.
- Since the problem in general is undecidable, we give two practical algorithms for finding a minimal set of satisfying c-instances. The first algorithm runs an exhaustive search subject to a size limit on the c-instances inspired by the chase procedure from Data Exchange [21, 22]. Then, we provide a more efficient chase algorithm that may return a smaller set of satisfying c-instances.
- We experimentally show scalability and quality of the c-instances returned by our algorithms varying different parameters. Our experiments use a real collection of wrong queries submitted by students of an undergraduate database course as well as queries over the TPC-H schema. Finally, we provide a comprehensive user study and a case study that show the usefulness of our approach.

2 RELATED WORK

Test data generation. QAGen [9] was among the first systems that focused on data generation in a query-aware fashion. The system aimed at testing the performance of a database management system given a database schema, one parametric Conjunctive Query, and a collection of constraints on each operator. MyBenchmark [37] extends [9] by generating a *set* of database instances

that approximately satisfies the cardinality constraints from a set of query results. HYDRA [47] uses a *declarative approach* that allows for the generation of a database summary that can be used for dynamically generating data for query execution. Cosette [15], which targets checking SQL equivalence without any test instances, encodes SQL queries to constraints using symbolic execution, and uses a constraint solver to find one counterexample that differentiates two input queries. RATest [41] proposes an instance-based counterexample for distinguishing two queries, where the emphasis is on the cardinality of the generated counterexample. The main differences between our work and [41] are that (1) we provide abstract instances with variables and conditions to pinpoint the source of error while comparing a wrong query against a correct query, (2) we only use the query and the schema to generate the counterexample and *do not need a database instance* to provides a counterexample, whereas [41] requires a database instance to output one sub-instance as the counterexample, (3) [41] provides one grounded instance where the correct and wrong queries differ, while we generate *a set of c-instances* aims to show all possible ways two queries can differ. XData [11] generates test data by covering different types of query errors that can commonly occur. Qex [52] is a tool for generating input relations and parameter values for a parameterized SQL query and aims at unit testing of SQL queries. It also generates one instance that satisfies the query and does not support nested queries and set operations, and thus does not support the full class of DRC queries. Olston et. al. [44] studied the problem of generating small example data for dataflow programs to help users understand the behavior of their programs.

Explanations for query results using provenance Data provenance has been studied from many aspects [6, 10, 13, 24–26, 28, 48]. A multitude of approaches have used provenance for *query answers explanations*. One approach [46] is based on tuple interventions in the database in the goal finding the influencing tuples over the query answers. Another [40] quantifies the responsibility of each input tuples to the query answer, an idea inspired by causality [45]. Shapley values [49] have also been used in this measure the contribution of each input tuple to the query result [36]. Natural Language (NL) explanations have also been proposed [18] when users employ an NLIDB such as [35]. The problem of explaining why a certain tuple is not in the query answer, also referred to as 'why-not provenance', has been studied using two approaches; instance-based [29–31, 34] where explanations are (missing) input tuples, and query-based [12, 19, 51] where explanations are based on query predicates or operators. Our approach suggests a query characterization that is independent of a specific database instance and thus also its provenance.

Coverage in software testing Test coverage is used to measure the percentage of a given software that is executed during the tests. Intuitively, the higher the test coverage, the lower the likelihood of the software containing bugs and unforeseen errors [39, 42, 53]. Different criteria for coverage have been proposed [43, 53], e.g., function coverage (checking the percentage of functions in the program that are executed during testing) and branch coverage (checking the percentage of branches, i.e., decisions that have true and false outcomes, in the program that are executed during testing). These ideas have certainly influenced our model.

Chase in schema mappings The chase procedure was originally suggested in the context of database dependencies [3, 38] and later used for generating schema mappings [14, 21]. The latter application aims to map one database schema to another, by using tuple-generating and equality-generating dependencies. These dependencies are then used in a chase procedure to generate the mapping between the schemas. Previous work has explored the complexity of the chase procedure and the types of solutions it is able to generate [20, 23, 27]. Our notion of a universal solution is therefore inspired by the notion of universal solution in the schema mapping problem [21]. Recent work has proposed an abstraction of schema mappings [7], which allows reusing meta schema mappings. In this paper, the c-instances can be thought of as "meta-instances" that can be mapped to concrete ones (e.g., [41]) as needed.

3 MODEL FOR QUERY CHARACTERIZATION

3.1 Databases and Relational Calculus

First we review and define domains and Domain Relational Calculus which will be used for express queries in this paper. A database schema $\mathbf{R}$ is a collection $(R_1, ..., R_r)$ of relation schemas. Each R_i is defined over a set of attributes denoted by $\texttt{Attr}(R_i)$. For each attribute $\texttt{A} \in \texttt{Attr}(R_i)$, its **domain** is a set of (possibly infinite) **constants** and is denoted as $\textsc{Dom}(\texttt{A})$, and $\textsc{Dom} = \cup_\texttt{A}\textsc{Dom}(\texttt{A})$; for simplicity, we will frequently use $\textsc{Dom}$ instead of $\textsc{Dom}(\texttt{A})$ with the implicit assumption that the constants are from the right domain $\textsc{Dom}(\texttt{A})$. Two relations can share the same attribute $\texttt{A}$; we use $R_i.\texttt{A}$ to explicitly denote an attribute $\texttt{A} \in \texttt{Attr}(R_i)$. Further, two attributes may share the same domain (e.g., when they share the same name or are related by foreign key constraints). A **ground instance** (or simply an **instance** when it is clear from the context) is a (possibly empty) finite set of tuples with constant attribute values that conform to the schema and corresponding domains. In addition, we allow standard constraints like key constraints, foreign key constraints, and functional dependencies in our framework.

DRC queries and tree representation. We next review the definition of Domain Relational Calculus (DRC) [33] and use it to define queries and syntax trees. It has been shown that DRC is equivalent to Relational Algebra [16], which provides the theoretical foundation to query languages such as SQL.

DEFINITION 1 (DRC QUERIES). *Given a schema* $\mathbf{R}$, *a* **DRC query** Q *has the form* $Q = \{(x_1, x_2, ..., x_p) \mid P_Q(x_1...x_p)\}$ *where* P_Q *is a standard first order logic (FOL) formula [1] involving relation names* $R_1, \cdots, R_r$, *constants from* $\textsc{Dom}$, *a set of* **query variables** $\mathcal{V}_Q$ *for attribute values, quantifiers* $\exists, \forall$, *operators* $\neg, =, >, \geq, <, \leq, \neq, LIKE$ *etc., and connectives* $\wedge, \vee$. *Here,* $\mathcal{V}_Q^{out} = \{x_1, \cdots, x_p\} \subseteq \mathcal{V}_Q$ *denote* **output variables** *of the query* Q, *which can be an empty set for a Boolean query. The output variables are free variables in* P_Q; *the remaining variables in* P *are quantified under* $\forall$ *or* $\exists$.

The formula P_Q *is built up 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* $y_i \in \mathcal{V}_Q \cup \textsc{Dom}$ *is a query variable or a constant, and (2) conditions* x_1 *op* x_2 *or* x_1 *op* c, *where* $x_1, x_2 \in \mathcal{V}_Q, c \in \textsc{Dom}$, *and op is a binary operator.*

A ground instance D *is said to* **satisfy** *a DRC query* Q *(denoted by* $D \models Q$) *if* $Q(D) \neq \emptyset$ *(for a Boolean query,* $Q(D) = \{\{\}\}$ *or* true), *i.e.,*

there is a **satisfying assignment** $\alpha : \mathcal{V}_Q^{out} \rightarrow D_{OM}$ *of the output variables of Q to the constants in D such that* P_Q *evaluates to true.*

We make a few assumptions without loss of generality: (1) in the FOL formula P_Q, all negations appear in DRC atoms (which can be achieved by repeated applications of standard equivalences like $\neg(\forall x P(x)) = \exists x(\neg P(x))$ and De Morgan's laws like $\neg(a \vee b) = \neg a \wedge \neg b$, etc.); (2) the DRC queries are **safe** or domain independent [1], i.e., any variable y_i that appears in a negated relation $\neg R(y_1..., y_k)$ also appear under a positive relation, e.g., queries like $\{x : \neg R(x)\}$ are not allowed; and (3) each quantified variable is unique in P_Q (which can be achieved by renaming).

EXAMPLE 2. *The queries* Q_A *and* Q_B *are shown in Figure 2, whereas Figure 3 gives the difference query* $Q_B - Q_A$ *that identifies beers at a non-lowest price but also not at the highest price. In Figure 3, the output variables are* x_1, b_1 *and the FOL formula is specified on the right hand side by renaming the variables in* Q_A, Q_B *in Figure 2. The ground instance K in Figure 1 satisfies* $Q_B - Q_A$ *in Figure 3, since there is a satisfying assignment* α *from the output variables* x_1, b_1 *in the query* $Q_B - Q_A$, *i.e.,* $\alpha(x_1) =$ *"Restaurant Raffaele",* $\alpha(b_1) =$ *"American Pale Ale", such that the formula in the query is satisfied when* $d_1 =$ *"Eve_Edwards",* $p_1 = 2.75$, $x_2 =$ *"Restaurant Memory",* $p_2 = 2.25$, *therefore the first part of the FOL formula from* Q_B *is true. The second part of the FOL formula with* $\wedge$ *is also true for all* d_2, p_3: *while the first two disjuncts under* $\neg$ *evaluates to false,* $\neg Serves(x_1, b_1, p_3) = true$ *for* $p_3 = 2.25, 3.5$, *and for* $p_3 = 2.75$ *the fourth disjunct is satisfied with* $x_3 =$ *"Tadim" and* $p_4 = 3.5$.

DEFINITION 2 (SYNTAX TREE OF QUERY). *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.*
(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, since in the formula P_Q *all negations appear in the DRC atoms, all negations in the syntax tree appear in the leaves; we do not use separate nodes for negation.*

Given a DRC query Q, we can have a unique syntax tree following the order of quantifiers in Q (e.g., for $\exists xy$, $\exists y$ appears as the child of $\exists x$, and a fixed order of associative connectives with appropriate parentheses, e.g., $(p_1 \vee p_2 \vee p_3)$ is always assumed to be $((p_1 \vee p_2) \vee p_3)$. However, two equivalent DRC queries may have different syntax trees, e.g., $\{x \mid R(x)\}$ and $\{x \mid (R(x) \wedge T(x)) \vee (R(x) \wedge \neg T(x))\}$. The syntax tree for the query $Q_B - Q_A$ from Figure 3 is shown in Figure 5 (to save space, we put multiple quantifiers with variables in the same node). The special treatment of the negation operator $\neg$ is for the sake of convenience in our algorithms.

3.2 Conditional Instances or C-Instances

We next give the definition of a *c-instance* adapting the concepts of *v-tables* and *c-tables* from the literature [32]. We distinguish between query variables whose domain is denoted by $\mathcal{V}$ (Definition 1), and variables in the c-instances whose domain is denoted by $\mathcal{L}$; we refer to the latter as **labeled nulls** (called *marked nulls* in [32]) for clarity. The c-instances involve conditions using **atomic conditions**, which are either (1) an atom of the form $[x \ op \ c] \ (\neg[x \ op \ c])$
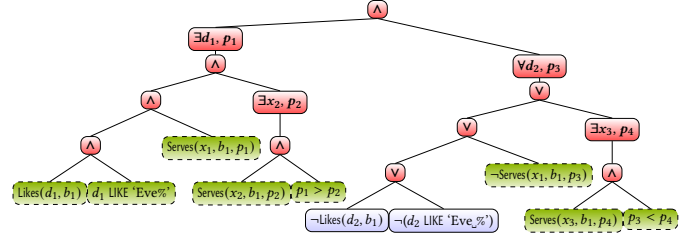

Figure 5: Syntax tree of $Q_B - Q_A$ **in Example 2. Atoms covered by the c-instance in Figure 4 are in green dashed boxes.**
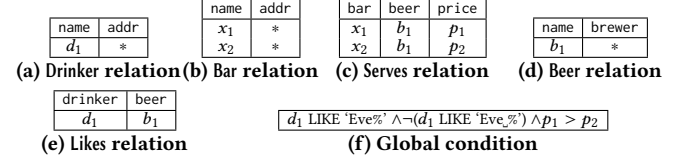

Figure 6: C-instance $\mathcal{I}_1$ **that satisfies** $Q_B - Q_A$.

or $[x \ op \ y](\neg[x \ op \ y])$ where x and y are labeled nulls in $\mathcal{L}$, c is a constant in $D_{OM}(x)$, and $op \in \{<, >, \leq, \geq, =, \neq, LIKE, ...\}$ 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.

DEFINITION 3 (CONDITIONAL INSTANCE OR C-INSTANCE). *A v-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* $A \in Attr(R_i)$, $t[A]$ *is either a constant from* $D_{OM}(A)$ *or is a labeled null from* $\mathcal{L}$.

*A **c-instance*** $\mathcal{I}$ *of* $\mathbf{R}$ *is a tuple of the form* $(\{T_1, \ldots, T_r\}, \phi)$, *where for each* $i \in [1, r]$, T_i *is a v-table with schema* R_i, *and* ϕ *is a conjunction of atomic conditions, which is associated with the c-instance, denoted as the **global condition**.*

Our definition of c-instances is slightly different from those found in previous literature [32], as we only associate the instance with a global condition, while there are no local conditions associated with a single tuple or even a single table in the instance. Table-level conditions might still appear in the global condition as a conjunct that contains labeled nulls from a single table, e.g. in Figure 4, the condition $p_1 > p_2$ is only relevant to the *Serves* table.

A conditional table (c-table) is a special case of a c-instance when there is only one relation in the instance, hence we only discuss c-instances in the rest of this paper. Note that we also allow for labeled nulls that do not affect the evaluation of ϕ, and are not needed for joins between tables. These are called "don't care" labeled nulls and are denoted by $*$ for simplicity instead of having unique names.

EXAMPLE 3. *Consider the c-instance* $\mathcal{I}_0$ *shown in Figure 4. The single tuple in the drinker table says that the name of the drinker is* d_1, *its address is a "don't care" token, and the condition says that the name* d_1 *must start with "Eve". The condition also enforces an order on the prices.* b_1 *cannot be replaced with* $*$ *because all three beers in Serves must be the same.*

C-instances define a set of possible worlds, each defined by a *mapping* (see below) to the labeled nulls in the c-instance.

DEFINITION 4 (MAPPING FOR C-INSTANCES). *Given a c-instance* $\mathcal{I} = (T_1, \ldots, T_r, \phi)$ *over a schema* $\mathbf{R}$, *a **mapping*** $\mu : \mathcal{L} \rightarrow D_{OM}$ *maps the labeled nulls* $\mathcal{L}$ *in* $\mathcal{I}$ *to their respective domains.*
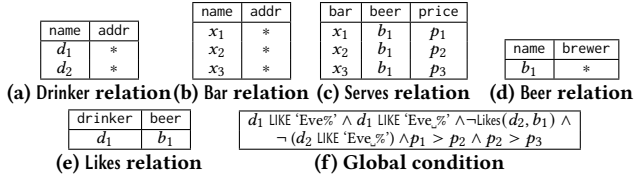
name	addr
d_1	*
d_2	*

(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%' $\wedge$ d_1 LIKE 'Eve_%' $\wedge \neg$Likes$(d_2, b_1) \wedge$ $\neg (d_2$ LIKE 'Eve_%'$) \wedge p_1 > p_2 \wedge p_2 > p_3$

(f) Global condition

Figure 7: C-instance $\mathcal{I}_2$ that satisfies $Q_B - Q_A$. It is not minimal since one of the Serves tuples can be removed without changing the coverage.

Extending μ to the c-instance $\mathcal{I}$, we denote $\mu(\mathcal{I})$ as the ground instance $\mu(\mathcal{I}) = \{\mu(x) : x$ is a labeled null in T_i, for $i \in [1, r]\}$. Let $\mathcal{L}' \subseteq \mathcal{L}$ be the labeled nulls appearing in ϕ of $\mathcal{I}$. Then, $\phi_\mu = \phi(\mu(\mathcal{L}'))$ yields the evaluation of ϕ (True or False) with the mappings given by μ.

DEFINITION 5 (POSSIBLE WORLDS AND CONSISTENCY FOR C-INSTANCES). *Given a c-instance $\mathcal{I} = (\mathsf{T}_1, \ldots, \mathsf{T}_r, \phi)$ of schema* **R***, the* **set of possible worlds** *for $\mathcal{I}$ is*

$$PWD(\mathcal{I}) = \{\mu(\mathcal{I}) : \mu \text{ is a mapping} \wedge \phi_\mu = True\}.$$

A c-instance $\mathcal{I}$ is said to be **consistent**, *denoted by IsConsistent($\mathcal{I}$) if $PWD(\mathcal{I}) \neq \emptyset$.*

Note that ground instances in $PWD(\mathcal{I})$ cannot contain extra tuples that are not in the c-instance $\mathcal{I}$. They may, however, have fewer tuples than the c-instance mapped to them because we consider set semantics, where a mapping may map two tuples with labeled nulls to the same tuple with constant values.

EXAMPLE 4. *One possible world of the c-instance in Figure 4 is the ground instance in Figure 1. In this ground instance, all labeled nulls have a mapping such that the global condition is satisfied.*

3.3 Query Characterization by C-Instances

We next give definitions for our framework of query characterization through c-instances. The intuition is that the c-instances should be *sound*, i.e., they should correctly capture a query, as well as *"complete"* in terms of *different ways* a query can be satisfied by ground instances, which requires more careful considerations using *"coverage"* of ground and c-instances as we discuss below.

DEFINITION 6. *[Satisfying c-instances] Given a query Q, a c-instance $\mathcal{I}$ is said to* **satisfy** Q *(denoted $\mathcal{I} \models Q$) if $\mathcal{I}$ is consistent (Definition 5) and for every ground instance D in $PWD(\mathcal{I})$, $D \models Q$.*

EXAMPLE 5. *Consider the query $Q_B - Q_A$ in Figure 3, and the c-instance $\mathcal{I}_0$ shown in Figure 4. $\mathcal{I}_0 \models Q_B - Q_A$ since every mapping of constants from the domain of its labeled nulls that satisfies the global condition will generate a ground instance D such that $D \models Q_B - Q_A$; $D = K_0$ in Figure 1 is an example.*

Coverage. Given a DRC query, the *Coverage* of a (ground or c-) instance captures different parts of a DRC query, that are satisfied by the instance, and helps us define the *completeness* of a set of c-instances. An inductive definition of coverage is given below.

DEFINITION 7. *[Coverage of ground instances] Given a DRC query syntax tree Q, a ground instance K such that $K \models Q$, and a satisfying assignment $\alpha : \mathcal{V}_Q^{out} \to \text{DOM}$ of the output variables of Q, the*

coverage cov(Q, K, α) of Q by K under α *identifies a subset of the DRC atoms (leaves) of the query syntax tree recursively top-down by extending α to all free variables in a subtree as follows:*

(1) *If Q consists of a single DRC atom (here all variables in Q are free), then cov(Q, K, α) contains the DRC atom of Q if:*
   (a) *$Q = R(x_1, \ldots, x_k)$ and $R(\alpha(x_1), \ldots, \alpha(x_k))$ appears in K, or*
   (b) *$Q = \neg R(x_1, \ldots, x_k)$ and $R(\alpha(x_1), \ldots, \alpha(x_k))$ is not in K, or*
   (c) *$Q = x$ op y and $\alpha(x)$ op $\alpha(y)$ evaluates to True; otherwise cov$(Q, K, \alpha) = \emptyset$.*
(2) *If $Q = Q_1 \wedge Q_2$ or $Q = Q_1 \vee Q_2$ (here the sets of free variables in Q, Q_1, Q_2 are the same), then cov$(Q, K, \alpha) = cov(Q_1, K, \alpha) \cup cov(Q_2, K, \alpha)$.*
(3) *If $Q = \exists x Q'(x)$ or $Q = \forall x Q'(x)$ (here x is a new free variable in Q'), then*
   cov$(Q, K, \alpha) = \cup_{c \in \text{DOM}_K} cov(Q', K, \alpha \cup \{x \to c\})$, where DOM_K denotes the constants appearing in K.

The **coverage of K for Q** *is defined as cov$(Q, K) = \bigcup_\alpha cov(Q, K, \alpha)$.*

Intuitively, the coverage cov(Q, K) is the set of atoms and conditions of Q that can be covered by a ground instance K, eventually leading to a satisfying assignment of the output variables of Q. Therefore, we use union to combine the coverages in Definition 7 for all cases, since we are interested in all possible ways to satisfy a query. Since α is a satisfying assignment of the output variables, the coverages of Q' in case (3) in Definition 7 for both $\exists, \forall$, and for both Q_1, Q_2 for a $\wedge$ node and for at least one of them for a $\vee$ node in case (2) above is non-empty. For universal quantifiers, it is worth noting that when the quantified variable takes different constants, different branches of the inner query (Q') may be satisfied, and thus provide different coverages, and to take all of them into account, we again employ union.

EXAMPLE 6. *The only satisfying assignment to the difference query $Q_B - Q_A$ depicted in Figure 3 w.r.t. the ground instance shown in Figure 1 is given by the assignment $\alpha(x_1) = $ "Restaurant Raffaele", $\alpha(b_1) = $ "American Pale Ale" described in Example 2. By applying the recursive top-down process implied by Definition 7, the DRC atoms of the query covered by this assignment are the leaves colored in green in Figure 5. Note that different assignments of a $\forall$ variable can cover different leaves, e.g., for $\forall p_3$ node in the right subtree, $p_3 = 2.25, 3.5$ covers the node $\neg Serves(x_1, b_1, p_3)$ atom whereas $p_3 = 2.75$ covers $Serves(x_3, b_1, p_4)$ and $p_3 < p_4$ atoms as discussed in Example 2.*

DEFINITION 8. *[Coverage of satisfying c-instances] Given a DRC query Q and a c-instance $\mathcal{I}$ such that $\mathcal{I} \models Q$, the* **coverage of $\mathcal{I}$ for** Q *is defined as cov$(Q, \mathcal{I}) = \bigcap_{K \in PWD(\mathcal{I})} cov(Q, K)$.*

Since $PWD(\mathcal{I})$ can contain ground instances with different coverages, the coverage of $\mathcal{I}$ is defined as the common coverage of all possible worlds. Therefore, the coverage of a c-instance $\mathcal{I}$ is always a (not necessarily strict) subset of any ground instance in $PWD(\mathcal{I})$.

EXAMPLE 7. *Reconsider the syntax tree of the query $Q_B - Q_A$ shown in Figure 5. Every ground instance in $PWD(\mathcal{I}_0)$ ($\mathcal{I}_0$ is depicted in Figure 4) has to have exactly three Serves tuples due to the global conditions (recall that the mapping from a c-instance to each ground instance in PWD has to be onto). Thus, the coverage of the c-instance $\mathcal{I}_0$ depicted in Figure 4 is exactly the coverage of the ground instance in Figure 1/Example 6 and highlighted in green with dashed frames*

in the tree shown in Figure 5. The c-instance $\mathcal{I}_2$ in Figure 7 covers all DRC atoms in the query $Q_B - Q_A$. To see this, note that there are two drinker variables d_1, d_2 in $\mathcal{I}_2$ such that $\neg \text{Likes}(d_2, b_1)$ as well as $\neg(d_2$ LIKE 'Eve_%') hold in all ground instances in $PWD(\mathcal{I}_2)$. Therefore, for the above satisfying assignments of output variables, when the $\forall d_2$ in the right subtree iterates over the constants corresponding to d_2, the two remaining uncovered leaves are covered as well.

Minimality of a c-instance. We now define minimality w.r.t. the coverage of a c-instance. In the next definition, we denote by $|\mathcal{I}|$ the **size of a c-instance** $|\mathcal{I}|$, defined as the total number of tuples and atomic conditions of $\mathcal{I}$. For instance, $|\mathcal{I}_0| = 12$ in Figure 4 (9 tuples and 3 atomic conditions).

DEFINITION 9 (MINIMAL SATISFYING C-INSTANCE). *Given a DRC query Q, a c-instance $\mathcal{I}$ with coverage C satisfying Q is **minimal** if for every other satisfying c-instance $\mathcal{I}'$ that has coverage C, it holds that $|\mathcal{I}| \leq |\mathcal{I}'|$.*

EXAMPLE 8. *Following Example 7, the c-instance $\mathcal{I}_0$ in Figure 4 is a minimal satisfying c-instance assuming natural foreign key constraints from Serves, Likes to Drinker, Beer, Bar, since any other c-instance with the same coverage has a larger size. In particular, $\mathcal{I}_1$ of smaller size (=10) in Figure 6 does not have the same coverage, since in the $\forall$ nodes of right subtree when d_2 (in query) = d_1 (in $\mathcal{I}_1$) and p_3 (in query) = p_1 (in $\mathcal{I}_1$), the two rightmost leaves Serves(x_3, b_1, p_4) and $p_3 < p_4$ are not covered by any ground instance in $PWD(\mathcal{I}_1)$.*

Minimality ensures that we do not include redundant tuples or conditions in our c-instances. While we adopt the above simple notion of minimality, it is ensured by a post-processing step in our algorithms, so any other reasonable form of minimality can also be used in this framework.

Universal solution. Our framework for finding solutions for a query Q is given in terms of a set of minimal c-instances $\mathcal{I} \models Q$, which immediately ensures soundness of our solutions, since any output c-instance is guaranteed to satisfy the query. Conversely, the notion of completeness is more challenging since there can be satisfying ground instances of unbounded size with redundant tuples that do not affect the query answer. The notion of coverage helps us define a notion of completeness using a *universal solution*, which ensures that for all satisfying ground instances of a certain coverage, a c-instance with the same coverage is included.

DEFINITION 10 (MINIMAL C-SOLUTION AND UNIVERSAL SOLUTION). *Let Q be a DRC query over a schema $\mathbf{R}$ and domain DOM. A **minimal c-solution** of Q is a set of minimal c-instances of Q, $\mathcal{S}_{\mathcal{I}} = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$, such that for all $\mathcal{I}_i$, $\mathcal{I}_i \models Q$, and for any two $\mathcal{I}_i, \mathcal{I}_j$ where $i \neq j$, $cov(Q, \mathcal{I}_i) \neq cov(Q, \mathcal{I}_j)$.*

*A **universal solution** of Q is a minimal c-solution $\mathcal{S}_{\mathcal{I}} = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$ such that (1) if there exists a ground instance K, where $K \models Q$, with coverage C, there is a c-instance $\mathcal{I}_i \in \mathcal{S}_{\mathcal{I}}$ with coverage $C = C_i$, (2) if we remove any c-instance from $\mathcal{S}_{\mathcal{I}}$, condition (1) does not hold.*

Note that for the universal solution, we do not require that $K \in PWD(\mathcal{I}_i)$ since K may contain more tuples than $\mathcal{I}_i$ and thus may not be part of the set $PWD(\mathcal{I}_i)$.

EXAMPLE 9. *Reconsider the $Q_B - Q_A$ shown in Figure 3. The set $\{\mathcal{I}_0, \mathcal{I}_1\}$ is a minimal c-solution for $Q_B - Q_A$ since they have different*

coverages and both satisfy $Q_B - Q_A$. Two of the three c-instances in the universal solution are $\mathcal{I}_0, \mathcal{I}_1$ shown in Figures 4, 6, 7, respectively (the rest are shown in Section 5.2 in the case study).

3.4 Computability of the Universal Solution

PROPOSITION 3.1. *The computability and complexity of finding a universal solution is as follows:*

(1) *Finding a universal solution is poly-time in the size of the query for $CQ^\neg$ (and therefore also for CQs), where $CQ^\neg$ is the class of conjunctive queries with negation that includes operators $\exists$, $\wedge$, $\neg$ for individual atoms and conditions.*

(2) *Checking whether a universal solution exists (or even any minimal c-solution exist) is undecidable for general DRC queries that may include the operators $\forall, \exists, \vee, \wedge,$ and $\neg$.*

PROOF. (1) The universal solution of $q \in CQ^\neg$ is a single c-instance comprising all relational atoms $R(x_1, \cdots, x_k)$ of the query, and a global condition that is the conjunction of all comparisons (x op y, x op c with or without negation) and negated relational atoms $\neg R(x_1, \cdots, x_k)$ in the query. This implies a poly-time complexity in the size of the query.

(2) Finding whether a universal solution exists is an undecidable problem for general DRC queries due to a reduction from the *finite satisfiability problem in first order logic* (FOL) that is known to be undecidable by the Trakhtenbrot's Theorem [50]. An FOL sentence ϕ is finitely satisfiable if there exists a finite ground instance K such that ϕ is true over K, which is true if and only if there is a satisfying c-instance $\mathcal{I}$. Hence the universal solution for Q is non-empty if and only if the global condition ϕ_Q is finitely satisfiable, which is undecidable when Q is a general DRC query. $\square$

Note that the inclusion of $\forall$ operators and arbitrary positioning of $\neg$ make general DRCs harder than $CQ^\neg$. In $CQ^\neg$, negations are only allowed in front of relational atoms and conditions subject to standard 'safety' constraints [2]. The query that returns all beers that are not liked by some drinker: $\{(b) \mid \exists x, d, a \, (\text{Beer}(b, x) \wedge \text{Drinker}(d, a) \wedge \neg\text{Likes}(d, b))\}$ in the DRC form, and in the equivalent (safe) Datalog with negation form: $Q(b) : - \text{Beer}(b, x), \neg\text{Likes}(d, b), \text{Drinker}(d, a)$, is an example of $CQ^\neg$.

Since the problem of finding a universal solution for general DRC queries is undecidable, in Section 4 we give an algorithm that builds an exhaustive minimal c-solution up to a certain limit on the size of the c-instances to ensure halting by checking all possible assignments of variables. Further, we give a more efficient algorithm in Section 4.3 that relaxes the requirement of generating all possible c-instances by providing a subset of satisfying c-instances.

4 ALGORITHM FOR MINIMAL C-SOLUTION

In this section we show how to compute an exhaustive set of satisfying c-instances up to a size limit for a DRC query Q by adapting ideas from the *chase* procedure [21, 22].

4.1 Basic Notions and Overview

Given a query syntax tree Q (Definition 2), our algorithm constructs an exhaustive set of c-instances in the minimal c-solution by recursively extending each c-instance in multiple ways. To explore the different options of extending each c-instance, our algorithm takes

a similar approach to that of the chase procedure, that was originally proposed for database dependencies [3, 38]. It constructs the different options for c-instances using a breadth first search (BFS) procedure, thereby implicitly generating a chase tree [8]. However, unlike the classic chase algorithm that directly adds or modifies tuples, our procedure converts the syntax tree into a conjunction of atoms and then maps the atoms in the conjunction to tuples and conditions which are added to the c-instance. Each quantifier and connector in the tree triggers a tailored recursive call. While creating the c-instances, the algorithm keeps track of the mappings between the query variables and the labeled nulls in the c-instances. We next formally define this homomorphism between query and c-instance; since we build the homomorphism in steps, we define it as a partial function.

DEFINITION 11 (HOMOMORPHISM BETWEEN QUERY AND C-INSTANCE). *Given a query Q and a c-instance $\mathcal{I}$ on the same schema and domain DOM, with query variables $\mathcal{V}_Q$ and labeled nulls $\mathcal{L}_{\mathcal{I}}$, and constants $C_Q, C_{\mathcal{I}} \subseteq$ DOM respectively, a **homomorphism** h **from** Q **to** $\mathcal{I}$ is a partial function $h : \mathcal{V}_Q \cup C_Q \rightarrow \mathcal{L}_{\mathcal{I}} \cup C_{\mathcal{I}}$ such that, (1) for each constant $c \in C_Q$, $h(c) = c$, and (2) for an atom $a = R(x_1, ..., x_k)$ in Q, if all of $h(x_1), \cdots, h(x_k)$ are defined, $h(a) = R(h(x_1), ..., h(x_k))$ is in relation R in $\mathcal{I}$.*

As opposed to assignments from queries to ground instances, homomorphisms to c-instances are not restricted to output variables but can also map quantified variables to labeled nulls. This allows the algorithms to consider multiple different homomorphisms of variables to different labeled nulls. For universally quantified variables $\forall x$, the algorithm keeps track of multiple homomorphisms even within the same c-instance.

We also abuse terminology as follows. Given a query Q and a c-instance $\mathcal{I}$ such that there is a homomorphism h from Q to $\mathcal{I}$, we refer to the **domain of a query variable in** Q as the set of labeled nulls with the same domain in $\mathcal{I}$ according to h, i.e., the labeled nulls in the same attribute or in corresponding attributes in different tables that share the same domain (e.g., by foreign key dependencies).

The procedure starts by mapping the free variables of the query to fresh labeled nulls in the c-instance. Then, it performs a BFS where for each c-instance in the queue, it expands the homomorphism and the c-instance using a recursive procedure. For a syntax tree with no quantifiers, the recursive procedure adds its atoms as tuples to the c-instance, ensuring that the variables in the atoms are converted to their labeled null counterparts according to the homomorphism. Otherwise, it handles the syntax tree based on its root: each quantifier ($\forall, \exists$) or connective ($\land, \lor$) is handled separately.

To ensure the minimality of each c-instance in the obtained set and the minimality of the set itself (Definitions 9 and 10), we use a post-processing procedure that checks the coverage of each c-instance, and for any coverage, it keeps a single c-instance with minimum size (breaking ties arbitrarily).

4.2 Exhaustive Chase for C-Instances

Algorithm 1 and Algorithm 2 form the main body of our 'chase' procedure. The procedure starts by calling Algorithm 1 (TREE-CHASE-BFS) on the schema $\mathbf{R}$, the entire query Q, an empty instance $\mathcal{I}_0$, and an empty mapping h_0. In addition, the size bound *limit* sets the

---

Algorithm 1 Tree-Chase-BFS

TREE-CHASE-BFS($\mathbf{R}, Q, h_0, \mathcal{I}_0, limit$)

    Input: $\mathbf{R}$: the database schema; Q: a DRC query;
    $\mathcal{I}_0$: a c-instance of schema of $\mathbf{R}$; h_0: a mapping;
    limit: the maximum number of tuples and conditions in the c-instance;
    Output: A list of satisfying c-instances for Q.
1   res = [ ], queue = an empty queue
2   **for** $x \in FreeVar(Q)$
3   Create a fresh variable x' in the domain of x
4      $\mathcal{I}_0.domain(x) = \mathcal{I}_0.domain(x) \cup \{x'\}$
5      $h_0 = h_0 \cup \{x \rightarrow x'\}$
6   queue.push($\mathcal{I}_0$)
7   visited = $\emptyset$
8   **while** ¬ queue.isEmpty()
9   I = queue.pop()
10   **if** $I \in$ visited or $|I| > limit$
11   continue
12   visited = visited $\cup \{I\}$
13   **if** TREE-SAT($Q, I, \emptyset$) and IsConsistent(I)
14        $res.append(I)$
15   continue
16     $Ilist$ = TREE-CHASE($\mathbf{R}, Q, I, h_0, limit$)
17   **for** $J \in Ilist$
18   **if** IsConsistent(J) and $|J| \leq limit$ and $J \notin visited$
19   queue.push(J)
20 **return** res

---

maximum number of tuples and atomic conditions in the global condition allowed in the c-instance, and is meant to ensure halting of the algorithm since finding a satisfying c-instance for a general DRC query is undecidable (Proposition 3.1).

Breadth-first search. First, to initialize the instance $\mathcal{I}_0$ and the mapping h_0 from free variables in the query to labeled nulls and constants in $\mathcal{I}_0$, for each free variable x in Q, Algorithm 1 will create a new labeled null and add it to the domain of x in $\mathcal{I}_0$ and update h_0 (Line 2-5). Then, the algorithm runs in a Breadth-first search manner: $\mathcal{I}_0$ is initially added to the empty queue; every time the algorithm takes the c-instance from the head of the queue, checks whether the instance has already been generated and its size does not exceeds *limit* (Line 10). The procedure for checking $I \in visited$ takes into account renaming of variables; it first compares certain properties of the c-instances (e.g., number of tuples, size of conditions etc.) and filters out candidates that cannot be equivalent to I, and then it checks all possible mappings to previously generated c-instances. It also checks (Line 13) (1) whether $I \models Q$ by the TREE-SAT procedure, and (2) whether it is consistent, i.e., $PWD(I) \neq \emptyset$, (we use an SMT solver in our implementation). It then runs the recursive procedure on the current c-instance (Line 16) and adds each one of the resulting c-instances to the queue (Lines 17–19).

Recursive generation of c-instances. The recursive procedure Algorithm 2 (TREE-CHASE) handles the query according to its root operator. It gets as input the schema of the relational database $\mathbf{R}$, the syntax tree of a DRC query Q, the current c-instance I, and the current homomorphism from Q to I. For the case where the query has no quantifiers (Line 2-7), the algorithm converts the syntax tree into a list of conjunction of atoms/atomic conditions, and then an instance is created for each conjunction under the homomorphism h by ADD-TO-INS. The algorithm proceeds according to the root of the syntax tree ($\land, \lor, \exists, \forall$) by calling procedures that get the same input as Algorithm 2, call Algorithm 1 recursively, and output a list of c-instances that are sent back to Algorithm 1.

Algorithm 2 Tree-Chase

TREE-CHASE($\mathbf{R}, Q, I, h, limit$)

Input: R: the database schema; Q: the query as its syntax tree;
I: current c-instance; h: current homomorphism from Q to I;
$limit$.
Output: a list of c-instances
1 res = [ ]
2 **if** there are no quantifiers in Q
3   L = TREE-TO-CONJ(Q)
4   **for** $\psi \in L$
5   J = ADD-TO-INS($\mathbf{R}, I, h(\psi)$)
6   **if** IsConsistent(J)
7   res.append(J)
8 **elseif** Q.root.operator $\in \{\wedge\}$
9   res = HANDLE-CONJUNCTION($\mathbf{R}, Q, I, h, limit$)
10 **elseif** Q.root.operator $\in \{\vee\}$
11   res = HANDLE-DISJUNCTION($\mathbf{R}, Q, I, h, limit$)
12 **elseif** Q.root.operator $\in \{\exists\}$
13   res = HANDLE-EXISTENTIAL($\mathbf{R}, Q, I, h, limit$)
14 **elseif** Q.root.operator $\in \{\forall\}$
15   res = HANDLE-UNIVERSAL($\mathbf{R}, Q, I, h, limit$)
16 **return** res

Algorithm 3 Handle-Conjunction

HANDLE-CONJUNCTION($\mathbf{R}, Q, I, h, limit$)

1 res = [ ]
2 lres = TREE-CHASE-BFS($\mathbf{R}, Q.root.lchild, I, h, limit$)
3 **for** $J \in lres$
4   **if** IsConsistent(J) = $false$
5   Continue
6   rres = TREE-CHASE-BFS($\mathbf{R}, Q.root.rchild, J, h, limit$)
7   **for** $K \in rres$
8   **if** IsConsistent(K)
9   res.append(K)
10 **return** res

Handling conjunction (∧). The HANDLE-CONJUNCTION proce-
dure recursively calls Algorithm 1 on both children of the root, and
every pair of solutions to each child is merged into one instance
by taking a union of every solution to the left subtree with every
solution to the right subtree, and adding the consistent instances
to the list of c-instances.

Handling disjunction (∨). The HANDLE-DISJUNCTION procedure
(Algorithm 4) reduces the disjunctive tree into three conjunctive
trees by replacing $Q = Q_1 \vee Q_2$ with $Q_1 \wedge Q_2$, $Q_1 \wedge \neg Q_2$, and
$\neg Q_1 \wedge Q_2$, since one of the three formulae is True iff $Q_1 \vee Q_2$ is
True. This conversion introduces negation to some subtrees, such a
negated subtree is translated into a syntax tree with negation only
on the leaves. Then, Algorithm 1 is called with each of the modified
trees, and each set of c-instances obtained from the three cases is
added to the result set.

Handling existential (∃) and universal (∀) quantifiers. If the
root node has $\exists x$, Algorithm 2 calls HANDLE-EXISTENTIAL (Algo-
rithm 5) that iterates over all labeled nulls or constants in the
domain of x, also adds a fresh labeled null, updates the homomor-
phism (as the quantified variable becomes free in the subtree), and
recursively calls Algorithm 1. HANDLE-UNIVERSAL (Algorithm 6)
handles the case when the root has $\forall x$. The difference with $\exists$ is that,
like the $\wedge$ case, the solutions to all labeled nulls and constants that
x is mapped to are merged into one instance. The algorithm first
checks whether there is no root and returns the inputted c-instance
in that case (Lines 2–3) adds each mapping from x to a labeled null

or constant to the homomorphism, runs the recursive procedure
to find all c-instances with this mapping and merges it with other
c-instances generated with other homomorphism that map x to
other labeled nulls or constants (Lines 5–14). It further generates
c-instances by mapping x to a fresh labeled null (Lines 19–24).

Ensuring minimality in post-processing. After Algorithm 1 re-
turns a set of c-instances, we remove the c-instances that are not
minimal in the following manner. For each c-instance in the set, we
compute a hash string for its coverage (we keep track of the cover-
age of each c-instance as it is created). Then, for each c-instance
in the set, we get all other c-instances in the set with the same
string representing its coverage and remove all but the minimal
one according to their size. Note that the hash function is applied to
the *coverage of the c-instance rather than the c-instances themselves*,
i.e., the function hashes the covered atoms of the query. Thus, it
allows us to efficiently detect c-instances with the same coverage
and remove those that are not minimal in terms of their number of
tuples and atomic conditions (ref. Section 3.3).

Soundness, termination, and complexity. Given a syntax tree
Q of a DRC query, Algorithm 1 when given ($\mathbf{R}, Q, \emptyset, \emptyset$) will output
a list of c-instances that are consistent, minimal, and satisfy the
query denoted by Q (validated in Line 13), i.e., our procedure is
sound and generates a valid minimal c-solution.

Although the problem of verifying if a satisfying c-instance
exists is undecidable (Proposition 3.1), Algorithm 1 is guaranteed
to terminate given the *limit* parameter. There are finitely many
distinct c-instances (that are not isomorphic in terms of renaming
of variables) up to size *limit* given a query. If the size of a c-instance
increases over *limit*, the algorithm will ignore this c-instance and
not push it into *queue* (Lines 10-11). The algorithm will also not
get into an infinite loop because of the *visited* set. Every generated
c-instance is placed into this set and c-instance already found is
the set are ignored (including renaming of variables) and are not
pushed into *queue* (Lines 18–19). Since the size of the schema is
constant, given a limit on size of the c-instances, the domains of all
labeled nulls in the c-instances are also finite since the domain is
derived from existing labeled nulls in the c-instance.

The running time of the algorithm is exponential in the number
of operators and size of the c-instances (bounded by *limit* × no. of
relations × max no. of attributes) since the algorithm performs an
exhaustive search on c-instances subject to the size limit, resulting
in a high complexity. This motivates us to design a more efficient
algorithm by generating a possibly smaller minimal c-solution that
we describe in the next subsection.

4.3 Optimization by Conjunctive Tree Chase

The optimized approach converts the original syntax tree into a set
of syntax trees where each tree does not contain disjunctions (∨),
then performs the chase procedure described in Algorithm 1 on
each one of the trees. This speeds up the solution dramatically since
there is no need to expand every disjunctive operator in the tree
into a set of trees that do not contain disjunction, thus simplifying
the process of generating c-instances.

Conversion of a tree with ∨ into conjunctive trees. For sets,
$Q_1 \vee Q_2$ is equivalent to three sets $\{Q_1 \wedge Q_2, \neg Q_1 \wedge Q_2, Q_1 \wedge \neg Q_2\}$.
However, this is not always true for FOL formulae, since $f_1 =$

$\forall x(P(x) \lor Q(x))$ is not equivalent to $f_2 = (\forall x P(x) \land Q(x)) \lor (\forall x \neg P(x) \land Q(x)) \lor (\forall x P(x) \land \neg Q(x))$ and only $f_2 \Rightarrow f_1$ holds; we demonstrate this below, first with a toy example and then using our running example.

EXAMPLE 10 (TOY EXAMPLE). *Consider $f = \forall x \, (even(x) \lor odd(x))$ where the predicate $even(x) \, (odd(x))$ is true if x is even (odd). Suppose the domain of x is $\{3, 4\}$. f is clearly satisfied with this domain. Now consider the conversion of f into three conjunctions: $f_1 = \forall x \, (even(x) \land odd(x))$, $f_2 = \forall x \, (\neg even(x) \land odd(x))$, and $f_3 = \forall x \, (even(x) \land \neg odd(x))$. f_1 is not satisfied since neither 3 nor 4 are both even and odd, f_2 is not satisfied since 4 is even, and f_2 is not satisfied since 3 is odd, thereby losing completeness in the conversion.*

EXAMPLE 11. *Consider the syntax tree of $Q_B - Q_A$ in Figure 5, and a sub-formula of its right subtree:*

$$\forall p_3 \big(\neg Serves(x_1, b_1, p_3) \lor \exists x_3, p_4 \big(Serves(x_3, b_1, p_4) \land p_3 < p_4 \big) \big)$$

We can convert this into the following three formulae:

$$\{ \forall p_3 \big(\neg Serves(x_1, b_1, p_3) \land \exists x_3, p_4 \big(Serves(x_3, b_1, p_4) \land p_3 < p_4 \big) \big),$$

$$\forall p_3 \big(\neg Serves(x_1, b_1, p_3) \land \forall x_3, p_4 \big(\neg Serves(x_3, b_1, p_4) \lor p_3 \geq p_4 \big) \big),$$

$$\forall p_3 \big(Serves(x_1, b_1, p_3) \land \exists x_3, p_4 \big(Serves(x_3, b_1, p_4) \land p_3 < p_4 \big) \big) \}.$$

These three formulae are equivalent to the original one if there is only one variable that p_3 can be mapped to in the price domain (in this case the original formula would be unsatisfiable). However, when there are more than two variables or constants in the domain, this conversion is not equivalence-preserving and will miss satisfying c-instances. For example, the c-instance $\mathcal{I}_0$ in Figure 4 satisfies the original formula but does not satisfy any of the three conjunctive formulae - assigning p_1 from $\mathcal{I}_0$ to the universally quantified p_3 will not satisfy any of the formulae.

Bearing this in mind, we describe an algorithm that performs this conversion. Given a syntax tree Q, the algorithm converts it into a set of conjunctive syntax trees based on the above principal.

We start with a syntax tree Q that may contain the $\lor$ operator in different nodes. The algorithm recurses on the tree where the base case is a Q that contains a single atom, then the algorithm simply creates a conjunctive tree from this atom, or its negation if it is negated. If the root of Q is an $\land$ node, the algorithm continues to recurse over the two children and joins each pair of obtained subtrees. If the root of Q is an $\lor$ node, the algorithm considers three cases, as mentioned above: (1) converting the root into $\land$ and recursing over both of its children, (2) converting the root into $\land$, negating the right child, and recursing over both of its children, and, finally, (3) converting the root into $\land$, negating the left child, and recursing over both of its children, All solutions to the three cases are added to the list of c-instances *res*. If the root of Q is a $\forall$ or $\exists$ quantifier, the algorithm recurses over the child of the root and adds the resulting trees to *res*.

Chase for conjunctive trees. The main chase procedure utilizes Algorithm 1 and applies it on the conjunctive tree obtained from the conversion algorithm. It gets a schema, a syntax tree, and a limit as input. It first converts the input tree into a set of conjunctive trees using the conversion algorithm. It then calls Algorithm 1 with each one of the conjunctive trees and adds the resulting c-instances to the list of results which is then outputted.

Soundness and complexity. The soundness of the algorithm, i.e., that it returns a set of minimal satisfying c-instances, follows from the fact that the final set is returned by Algorithm 1. Although there is an exponential dependency in the number of operators, the algorithm gives a better running time by avoiding recursive calls to split a query tree into three query trees recursively for the disjunction operator, at the cost of possibly not generating some satisfying c-instances that are generated by Algorithm 1.

Other optimizations. The time complexity is also largely affected by the number of labeled nulls in each domain, especially when handling universal quantifiers. Hence, we try to optimize the algorithm by disallowing the universal quantifier to add new labeled nulls, though this might lose completeness. Moreover, notice that our TREE-CHASE-BFS is always initially called with $\mathcal{I}_0 =$ an empty c-instance, we could manipulate it to achieve different coverage by calling TREE-CHASE-BFS on a c-instance that is properly initialized with the tuples or atomic conditions we target to cover. We further evaluate these optimizations in Section 5.

Setting the limit parameter in Algorithm 1. The *limit* parameter can be set in several different manners in practice. One approach is setting a default *limit* according to query complexity. *limit* determines the maximal size of the c-instance. In our experimental results (Section 5) we have seen that it can be set to some multiple of the number of query atoms to safely allow for a c-instance to covers all the atoms of the query (we have used a multiple of 2 in our evaluation). Another alternative, aimed at an interactive experience, is to set a timeout parameter instead of the *limit* (as done in Section 5), thus allowing Algorithm 1 to explore higher limits as needed up to the allotted time.

5 EXPERIMENTS

We investigate the performance of our approach and compare it to different variations of our approach in the following aspects: (1) runtime for varying query complexity (2) varying the limit parameter in Algorithm 1, (3) properties of the output c-instances, and (4) case studies showing the actual obtained c-instances for a sample of the experimental queries.

Setup. We implemented our methods in Python 3.7. We ran all experiments locally on a 64-bit Ubuntu 18.04 LTS server with 3.20GHz Intel Core i7-8700 CPU and 32GB 2666MHz DDR4 RAM. We compare the following variants of our algorithms and optimizations.

- **DISJ-NAIVE**: this method implements the exhaustive chase procedure described in Section 4.2.
- **CONJ-NAIVE**: this method implements the optimized conjunctive tree chase procedure described in Section 4.3 that converts the original syntax tree into a set of syntax trees without disjunction.
- **DISJ-EO/CONJ-EO**: this method adapts DISJ-NAIVE/CONJ-NAIVE by only allowing the algorithm to add labeled nulls to the c-instance when handling an existential quantifier node.
- **DISJ-ADD/CONJ-ADD**: this method first runs DISJ-EO(or CONJ-EO) on the empty c-instance, then gets the minimal c-solution. If there are still leaf atoms not covered by any of the c-instance in the c-solution, then it iterates over every

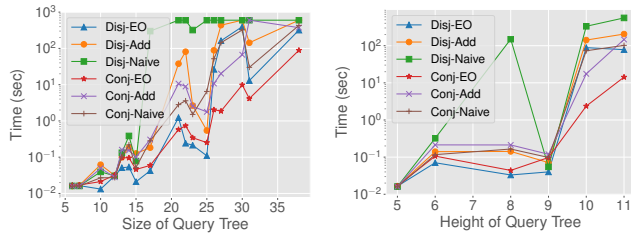**Figure 8: Running time vs. various measures of query complexity.** $limit = 10, timeout = 600sec.$

Dataset	# Queries	Mean # Atoms	Mean # Quantifiers	Mean # Or	Mean Height
Beers	35	6.40	13.94	2.17	9.54
TPC-H	28	11.96	23.07	4.18	12.07

Table 1: Dataset statistics.

remaining uncovered leaf atom, creating corresponding labeled nulls and adding the atom to the initial c-instance, and runs Disj-EO (Conj-EO resp.) on the initialized c-instance.

We evaluate both the efficiency/scalability of our algorithms in terms of runtime and the quality of results with respect to different measures, and compare them against systems from related work.

Datasets. We used two datasets in our experiments, **Beers** and *TPC-H*. For Beers dataset, queries in these experiments come from submissions by students for an assignment in an undergrad database course. We picked 5 questions (skipped those with only simple selection and join) and sampled a few students' queries, then manually rewrote them into domain relational calculus. There were 5 (correct) standard queries and 10 students' wrong queries; we also considered the difference between the standard queries and the wrong queries (and also the opposite direction), resulting in additional 20 queries. Some queries are very complex as they use the difference operator multiple times, resulting in nested universal quantification in the DRC query. Similarly, for TPC-H, we picked 4 queries (Q4, Q16, Q19, and Q21) and dropped their aggregate functions, then made two wrong queries each, resulting in 28 test queries in total. The statistics of the datasets are in Table 1.

5.1 Performance Evaluation

Scalability. To evaluate the scalability of our approach, we study how query complexity affects the running time. We consider four measures of query complexity: (1) number of nodes in the query tree, (2) the height of the query tree, (3) number of universal quantifiers plus number of disjunction that is below a universal quantifier, and (4) the number of both universal and existential quantifiers. Although most of these parameters are specific to our algorithms that operate on DRC queries, the number of universal quantifiers also has a corresponding complexity notion in the SQL form since each universal quantifier in DRC would lead to at least one negated sub-query in SQL.

Example 12. *Recall the query in Figure 3 and its syntax tree in Figure 5. The number of nodes in the tree is 27 (measure (1)), the height of the tree is 8 (measure (2)), the query contains 2 universal quantifier, 3 disjunctions below it, and 6 existential quantifiers, so measure (3) is 5 and (4) is 8. For reference, the queries Q_A and Q_B from our running example are shown in SQL in Figure 9.*

We set the limit threshold to be 10 for the Beers dataset and 15 for TPC-H, and stops the algorithm if it does not finish in 10
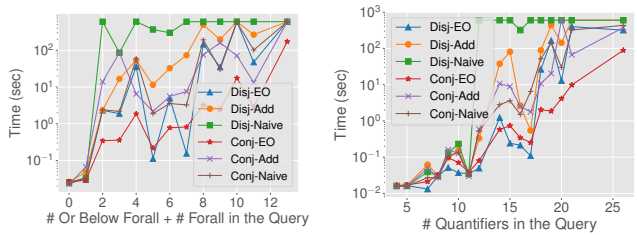
```
SELECT l.beer, s.bar
FROM Likes l, Serves s
WHERE l.drinker LIKE 'Eve_%' AND
l.beer = s.beer
AND NOT EXISTS(
    SELECT * FROM Serves
    WHERE beer = s.beer AND price >
        s.price);
```
(a) Correct query Q_A

```
SELECT S1.beer, S1.bar
FROM Likes L, Serves S1, Serves S2
WHERE L.drinker LIKE 'Eve%' AND
L.beer = S1.beer AND L.beer = S2.
    beer
    AND S1.price > S2.price;
```
(b) Incorrect query Q_B

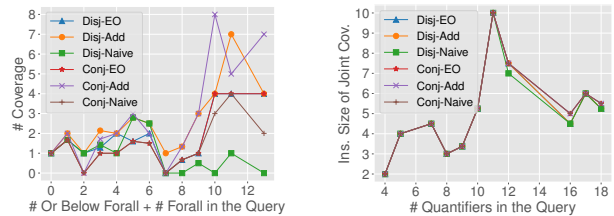**Figure 9: Queries from our running example in SQL.**

Figure 10: Result quality by query complexity. $limit = 10, timeout = 600sec.$

minutes (20 minutes for TPC-H). The results are shown in Figure 8 and Figure 11, respectively. We report the average running time for different queries with the same value of the complexity measure.

As shown in in Figure 8, the running time increases with query complexity. Disj-Naive has the worst time complexity (more than exponential), which did not finish for most of the complex queries with more than 10 quantifiers, followed by Disj-EO and Disj-Add, whereas Conj-Naive, Conj-EO, Conj-Add perform better. The running time of Conj-Naive, Conj-EO, Conj-Add increases exponentially as expected since their complexity largely depends on the number of conjunctive syntax trees generated from the original query. Compared to the total number of nodes and the height, the number of universal quantifiers and the number of disjunction nodes are more crucial to the growth in the running time. Similar trends are illustrated in Figure 11, whereas Conj-EO still performs better than Disj-EO, while the running time of Conj-Add is very close to Disj-Add. We conjecture that this is because the overall complexity of queries in the TPC-H dataset is much higher than the Beers dataset, as shown in Table 1, leading to more generated conjunctive syntax trees. Our results indicate that our solution scales well for complex schemas and queries (except for very complex and long queries: for only 4 extremely complex cases out of the 28 cases in the TPC-H dataset, our algorithm failed to return any results) .

Result quality. Our optimized approaches (Conj-EO, Disj-EO, Conj-Add and Disj-Add) run much faster than Disj-Naive by compromise on the completeness of the minimal c-solution to different extents. To evaluate the result quality of these approaches in terms of both completeness and minimality, we show in Figure 10 the number of distinct coverage from the returned c-solutions and the
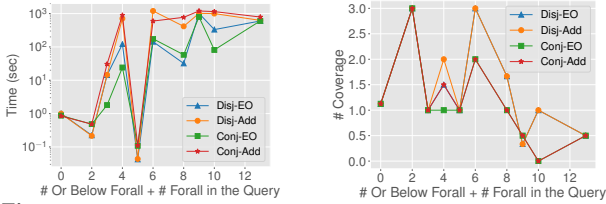
Figure 11: Running time and result quality by query complexity on TPC-H dataset $limit = 15, timeout = 1200sec.$
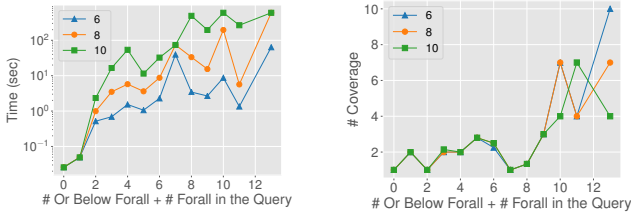

Figure 12: Parameter sensitivity varying limit. DISJ-ADD.
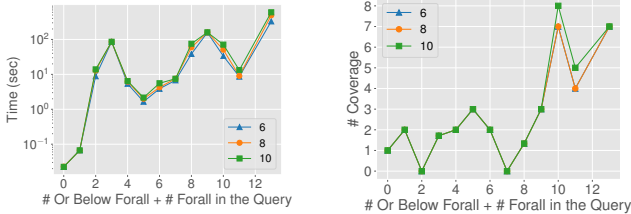

Figure 13: Parameter sensitivity varying limit. CONJ-ADD.

average size of the c-solutions. Notice that the number of returned minimal c-solution of each variant can be different (either they are unable to find some results, e.g. DISJ-EO returns a subset of DISJ-ADD; or some variants finish before the timeout but the others do not), to guarantee a fair comparison, for each query we only consider c-solutions with a coverage set returned by all of the variants. For example, for a query Q if CONJ-NAIVE returns two c-solutions with coverage C_1 and C_2, and DISJ-ADD returns three c-solutions with coverage C_1, C_2, C_3, we will only report the average c-solution size of the two with coverage C_1 and C_2 for Q.

Figure 10 shows that DISJ-ADD returns more distinct coverage sets in most cases, while CONJ-ADD, CONJ-NAIVE,CONJ-EO, and DISJ-EO might fail to return any satisfying instances. There are a few exceptions for some very complex queries where DISJ-ADD did not finish before timeout and thus CONJ-ADD returns more distinct coverage sets. Although DISJ-NAIVE did not finish running in most cases, the c-solutions it returns can be smaller than other variants when there are more than 10 quantifiers in the query. There are few cases where the DISJ-ADD and DISJ-EO return smaller c-solutions compared to the variants using conjunctive trees.

Parameter sensitivity. To ensure that the algorithm terminates, we used a *limit* parameter to restrict the size of the c-instances. Figure 12 and Figure 13 show how this limit affects the running time and completeness for DISJ-ADD and CONJ-ADD. Although the running time grows exponentially with the query complexity, DISJ-ADD runs one order of magnitude faster when limit=6 than limit=10, losing completeness only when the query tree is very complex. For CONJ-ADD, the difference in running time when varying the limit is negligible in most cases, and the number of distinct returned coverage sets only changes for the most complex queries.

Interactivity. To improve interactivity, our algorithms can output the instances one at a time as soon as they are generated, so users can start exploring immediately and have a more interactive experience. The time to produce the first instance for our algorithms on the Beers dataset is only 4.78 seconds on average (DisjAdd) or 0.77 seconds (ConjAdd), and the average delay between two consecutive output instances with different coverage is 18.34 seconds (DisjAdd) or 5.22 seconds (ConjAdd). While on the TPC-H dataset, the time to produce the first instance is longer but still tolerable: 101.25 seconds on average (DisjAdd) or 88.02 seconds (ConjAdd), and the average delay between two consecutive output instances with different coverage is 19.12 seconds (DisjAdd) or 54.16 seconds (ConjAdd). Note that doing so may risk returning non-minimal instances, as minimality is verified in postprocessing (Section 4.2). Another option is to start with the optimized version (Section 4.3) and if further insights are needed, run the exhaustive search (Section 4.2). We also note that slightly longer wait times might be acceptable in some scenarios, e.g., providing offline feedback to student solutions, or to help students/instructors when manual debugging would take significantly more effort for complex queries or subtly wrong solutions.

5.2 Case Study

By providing the "basis" to a query Q, our work yields a set of abstract instances that can help users understand and debug their query in practice. To evaluate the usefulness of the set of abstract instances (the minimal c-solution returned by the algorithms, providing a proxy for the universal solution), we report one case study on the same real-world dataset as the performance evaluation from an undergrad database course. We pick two most complex standard solution queries from an assignment each with one wrong query from student submissions. Table 2 shows the solution queries, wrong queries, and the universal solution for the difference query of the standard and wrong queries.

The universal solution captures different errors in the wrong query. To compare, we use the ground instances that serve as "counterexamples" for the wrong queries by a previous system [41] based on a randomly generated testing database instance.

For Q_1 (the same as our running example), the first and the second c-instances pinpoint that if the drinker's first name is not 'Eve' but has 'Eve' as its prefix. While the first c-instance does not contain the first name condition, it shows that if all three bars serve the same beer at different prices, the query would go wrong. Note that if we add to the last instance the condition $\neg(d_1 \text{LIKE 'Eve\_\%'})$, it is still a satisfying c-instance, but it is not minimal because its coverage is the same as the second c-instance. In comparison, while the ground instance by [41] (as in Figure 1) is in the represented world of the first c-instance, it does not highlight that the reason behind the wrong query result is that the prices are ordered in a particular way, but the actual values are unimportant.

For Q_2, the c-instances in the universal solution indicate that the query would go wrong if there is a drinker frequents to a bar that does not serve any beer, no matter the drinker likes a beer or not (the 1st and 3rd instances). This may pinpoint the error that the Likes table does not interact with the Serves table. Furthermore, the 2nd, 5th, and 6th c-instances imply that if there is a beer served at a bar, to make the query return a wrong result, the drinker should not

Query description	Queries (DRC)	c-instances
Q_1: for each beer liked by any drinker whose first name is Eve, find the bars that serve this beer at the highest price	Correct query: Q_A in Figure 2a. Wrong query: Q_B in Figure 2b.	I_0 in Figure 4
		I_1 in Figure 6
		Drinker(d_1, *), Drinker(d_2, *), Beer(b_1, *), Bar(x_1, *), Bar(x_2, *), Likes(d_1, b_1), Serves(x_1, b_1, p_1), Serves(x_2, b_1, p_2), d_1 LIKE'Eve%' $\wedge$ $\neg$(d_1 LIKE'Eve_%') $\wedge$ $\neg$(Likes(d_2, b_1)) $\wedge$ $p_1 < p_2$
Q_2: Find names of all drinkers who frequent only bars that serve some beer they like	Correct Query: $Q_{2A} = \{(d_1) \mid \exists a_1 (\text{Drinker}(d_1, a_1) \wedge$ $\forall x_1 \forall t_1 (\neg\text{Frequents}(d_1, x_1, t_1) \vee \exists b_1, p_1$ $(\text{Serves}(x_1, b_1, p_1) \wedge \text{Likes}(d_1, b_1))))\}$ Wrong Query: $Q_{2B} = \{(d_1) \mid \exists a_1 (\text{Drinker}(d_1, a_1) \wedge$ $\forall b_1 (\forall t_1, t_1, p_1 (\neg\text{Frequents}(d_1, x_1, t_1) \vee$ $\neg\text{Serves}(x_1, b_1, p_1)) \vee \text{Likes}(d_1, b_1)))\}$ Showing universal solution for $Q_{2B} - Q_{2A}$	Drinker(d_1, *), Beer(b_1, *), Bar(x_1, *), Likes(d_1, b_1), Frequents(d_1, x_1, t_1)
		Drinker(d_1, *), Beer(b_1, *), Bar(x_1, *), Bar(x_2, *), Likes(d_1, b_1), Serves(x_1, b_1, p_1), Frequents(d_1, x_2, t_1), $\neg$(Frequents(d_1, x_1, t_1))
		Drinker(d_1, *), Beer(b_1, *), Bar(x_1, *), Frequents(d_1, x_1, *),
		Drinker(d_1, *), Beer(b_1, *), Bar(x_1, *), Bar(x_2, *), Likes(d_1, b_1), Serves(x_1, b_1, p_1), Frequents(d_1, x_1, t_1), Frequents(d_1, x_2, t_1)
		Drinker(d_1, *), Beer(b_1, *), Bar(x_1, *), Bar(x_2, *), Serves(x_1, b_1, p_1), Frequents(d_1, x_2, t_1), $\neg$(Likes(d_1, b_1)) $\wedge$ $\neg$(Frequents(d_1, x_1, t_1))
		Drinker(d_1, *), Drinker(d_2, *), Beer(b_1, *), Bar(x_1, *), Bar(x_2, *), Likes(d_2, b_1), Serves(x_1, b_1, p_1), Frequents(d_2, x_2, t_1), $\neg$(Frequents(d_2, x_1, t_1)) $\wedge$ $\neg$(Likes(d_1, b_1))
		Drinker(d_1, *), Beer(b_1, *), Beer(b_2, *), Bar(x_1, *), Bar(x_2, *), Likes(d_1, b_1), Serves(x_1, b_1, p_1), Frequents(d_1, x_1, t_1), Frequents(d_1, x_2, t_1), $\neg$(Likes(d_1, b_2))

Table 2: Queries used in the case study; universal solutions generated using Disj-Add, *limit* = 10. Note that the symbols in queries (representing the query variables) and the symbols in c-instances (representing labeled nulls) are not the same.

frequent this bar, which could be interpreted as the correct solution uses the frequents table together with negation in a different way. Actually, the wrong query joins Frequents with Serves, while the correct solution joins Likes with Serves. Therefore, the universal solution provides different perspectives in understanding the query and formulates a hint on how to modify the wrong query. The ground instance by [41] only consists of four tuples: Drinker("Bryan", "39934 Main St."), Beer("Amstel", "A. Brewer"), Bar("The Edge", "802 Morris St."), and Frequents("Bryan", "The Edge", 3), which is in the represented world of the third c-instance from our universal solution in Table 2. Such a simple counterexample might be less helpful for users to understand why the query goes wrong, as one would benefit from the explicit conditions with negation.

5.3 User Study

We conducted a user study for the Beers dataset to evaluate: (R1) how effective our approach is for explaining and understanding bugs in queries, and (R2) whether completeness as a quality metric is helpful. For (R1), we specifically compare our approach (c-instances) with **concrete instances [41]** having constant values. Note that our approach takes only the queries and the schema as input, whereas [41] also takes a database instance as input and outputs a sub-instance as a concrete counterexample[1].

Participants. We recruited 64 participants, including 22 graduate students from CS departments and 42 undergraduate students from an undergraduate database course. Participation was voluntary and anonymous, though the undergraduates were offered small souvenirs as a reward for their participation (we did not get enough responses from the undergraduates in our initial pilot surveys). The undergraduate students were already familiar with the schema of the Beers dataset from an earlier homework; while for the graduate students, we explained the schema details and also asked about their familiarity with SQL. We note that the undergraduate students have also been exposed to the tool using concrete instances developed by [41] (but only for relational algebra queries); because of this

familiarity, concrete instances might hold a slight advantage over c-instances for these students. Half of the graduate students (11 out of 22) graduate students declared high familiarity with SQL queries and the rest reported moderate or low familiarity; our observations for both groups were similar in this study, therefore we report the overall statistics for graduate students.

Tasks. We asked all participants to spot errors in two SQL queries (each has two major errors, see Table 3) querying the Beers database, with the help of either our c-instances or concrete instances from [41]. We provided each participant with one query followed by c-instances and other query followed by concrete instances as counterexamples, randomly dividing them into two groups: one group saw (Q_1, c-instances) + (Q_2, concrete instances) and the other saw (Q_1, concrete instances) + (Q_2, c-instances). The order of showing these two questions for both groups was chosen at random to avoid any familiarity bias against either c-instances or concrete instances. Instead of showing completely abstract c-instances, we added an example concrete value to each variable in the c-instance, showing one way that it can be grounded. This is a trivial extension done to help alleviate novices' potential discomfort with seeing symbols and conditions alone. This approach somewhat blurs the line between c-instances and concrete instances, but faithfully represents how in practice c-instances would be deployed in an educational setting. Then, for each query we have the *treatment group* (with c-instances as explanation) and the *control group* (with concrete-value-only instances as explanation). To study (R2), following the task involving the first c-instance above, we presented a second c-instance for the same query but with a different coverage (which would illustrate a different error), and asked the participant what errors they found upon seeing both c-instances, and whether they felt the second c-instance provided additional help. (Note that [41] and other related work, there is no option for generating additional concrete instances that illustrate different errors in the same query.) At the end of the study, back to (R1), we also asked the participants about their preferences between c-instances and concrete instances for spotting errors in queries.

Results and analysis. Objectively, we evaluate the user performance by the number of errors they spotted. Figures 14 show the percentage of users who failed to spot any error, spotted one error, and spotted both errors in each group. For example, consider the last three bars in the left sub-figure in Figure 14, which show the

overall statistics (combining both queries) for all undergraduate participants. Showing the concrete instance alone is already quite helpful: only 31% of the users failed to find an error ("total-conc"). Going from "total-conc" to "total-CI1," we see a clear performance improvement among users who were shown c-instances: percentage of the users failing to find an error goes down to 19%. Going further from "total-CI1" to "total-CI2," we see that as soon as users are show a second c-instance, practically all of them were able to spot at least one error, and the majority (64.3%) of them indeed spot both errors in the query; in contrast, no users were able to spot both errors with only an concrete instance ("total-conc"). Similar conclusions can be drawn from per-query statistics (shown by the first two batches of three bars in the left part of Figure 14) as well as from statistics for graduate students (shown in the right part of Figure 14). Overall, these results convincingly show that for (R1), c-instances hold a clear advantage over concrete instances in objectively improving participants' performance in spotting errors in queries; and for (R2), showing multiple c-instances with different coverage dramatically improves participants' ability in spotting remaining errors in queries.

A number of other observations from these results are worth noting but largely confirms intuition. First, Q_2 was easier to debug than Q_1. Second, graduate students overall perform better than undergraduates. The coupling of these factors explains why we did not see any difference from "Q2-conc" to "Q2-CI1" in the right part of Figure 14; apparently most graduate students got enough help from the concrete instance in order to spot at least one error in the simpler Q_2. Nonetheless, they still needed the help with an additional c-instance to uncover the second error.

Figures 15 and 16 summarize the subjective responses from participants regarding their preference for c-instances vs. concrete instances, and their opinion on the usefulness of additional c-instances. A clear majority of the participants found the additional c-instance useful per Figure 16. However, from Figure 15, it is apparent that many participants prefer viewing concrete instances, despite the fact that they perform objectively better with the help of c-instances. This preference is stronger among undergraduates—only a third preferred c-instances, compared with more than a half for concrete instances. A relatively lower fraction of graduate students—but still a half of them—preferred concrete instances. It would be interesting to conduct additional study to pinpoint their reluctance to embrace c-instances despite their objective advantages, but there are several possible explanations. First, the abstraction provided by variables and conditions the c-instances may be seen as more intimidating, especially for undergraduates. This conjecture is corroborated by some of the free-form feedback comments we received. Second, as mentioned earlier in this section, the undergraduates already had some familiarity working with concrete instances before this user study. Overall, the fact that still about a third of the participants preferred c-instances shows that there is a sizable and compelling demand for this approach. We also believe we can mitigate some of the reluctance in this user base with improved interfaces and familiarity.

6 CONCLUSIONS AND FUTURE WORK

We have defined and studied the problem of compact query characterization using the coverage of abstract c-instances. We have

Query description	Wrong Queries
Q_1: 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 in Figure 9 (our running example)
Q_2: Among the drinkers who frequent "The Edge", find the names of those who do not like "Erdinger".	**SELECT DISTINCT** S.beer **FROM** Serves S, Likes L **WHERE** S.bar = 'Edge' **AND** S.beer = L.beer **AND** L.drinker <> 'Richard';

Table 3: Queries used in the user study.

Figure 14: User performance on spotting errors; *-conc: concrete instance only, *-CI1: the first C-Instance, *CI2: the second C-Instance (left: undergrad, right: graduate).
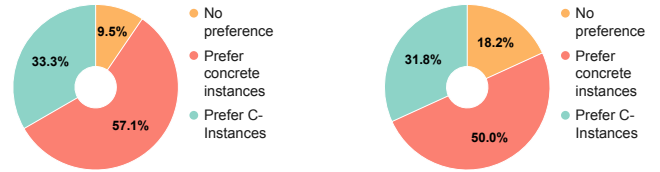

Figure 15: Preference on explanation types (left: undergraduate, right: graduate)
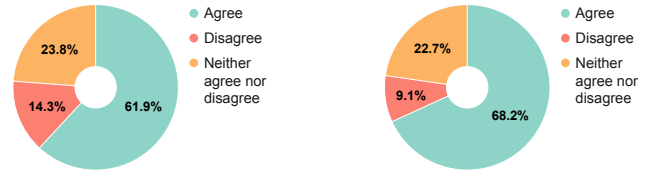

Figure 16: User feedback on "The second C-Instance provided additional help" (left: undergraduate, right: graduate)

devised algorithms and optimizations for computing such characterizations building on the concept of chase and utilizing the structure of the syntax tree. We experimentally showed that our approach is effective at finding c-instances that characterize the query and examined the effect of query complexity and parameter changes on the scalability of our approach. In future work, we plan to study the development of further optimizations for finding such solution for more query classes different properties of c-instances. In this paper we showed that the problem of finding a universal solution is poly-time for CQ⁻ queries, while the decision version is undecidable for general DRC queries: understanding the computability and complexity for universal solutions for other query classes in between is another interesting research direction. Finally, while our model can support queries with the same final aggregate and different bodies by removing the aggregate, extending our model to support arbitrary aggregate queries is another intriguing direction of future work.

ACKNOWLEDGMENTS

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[3] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. 1979. The Theory of Joins in Relational Databases. *ACM Trans. Database Syst.* 4, 3 (1979), 297–314.

[4] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[5] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*.

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

[7] Paolo Atzeni, Luigi Bellomarini, Paolo Papotti, and Riccardo Torlone. 2019. Meta-Mappings for Schema Mapping Reuse. *Proc. VLDB Endow.* 12, 5 (2019), 557–569.

[8] Vince Bárány, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2017. Declarative Probabilistic Programming with Datalog. *ACM Trans. Database Syst.* 42, 4 (2017), 22:1–22:35.

[9] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *SIGMOD*. 341–352.

[10] 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.

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

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

[13] J. Cheney, L. Chiticariu, and W. C. Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* (2009), 379–474.

[14] Laura Chiticariu and Wang Chiew Tan. 2006. Debugging Schema Mappings with Routes. In *Proc. VLDB Endow.* 79–90.

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

[16] Edgar F Codd et al. 1972. *Relational completeness of data base sublanguages*.

[17] 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.

[18] Daniel Deutch, Nave Frost, and Amir Gilad. 2020. Explaining Natural Language query results. *VLDB J.* 29, 1 (2020), 485–508.

[19] Daniel Deutch, Nave Frost, Amir Gilad, and Tomer Haimovich. 2020. Explaining Missing Query Results in Natural Language. In *EDBT*. 427–430.

[20] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. 2008. The chase revisited. In *PODS*. 149–158.

[21] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2003. Data Exchange: Semantics and Query Answering. In *ICDT*. 207–224.

[22] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. 2003. Data exchange: getting to the core. In *PODS*. 90–101.

[23] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. 2005. Data exchange: getting to the core. *ACM Trans. Database Syst.* 30, 1 (2005), 174–210.

[24] R. Fink, L. Han, and D. Olteanu. 2012. Aggregation in Probabilistic Databases via Knowledge Compilation. *PVLDB* 5, 5 (2012), 490–501.

[25] F. Geerts and A. Poggi. 2010. On database query languages for K-relations. *J. Applied Logic* 8, 2 (2010), 173–185.

[26] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller. 2013. Provenance for Data Mining. In *TaPP*.

[27] Georg Gottlob and Alan Nash. 2006. Data exchange: computing cores in polynomial time. In *PODS*. 40–49.

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

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

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

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

[32] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (1984), 761–791.

[33] Michel Lacroix and Alain Pirotte. 1977. Domain-Oriented Relational Languages. In *VLDB*. 370–378.

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

[35] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endow.* (2014), 73–84.

[36] Ester Livshits, Leopoldo E. Bertossi, Benny Kimelfeld, and Moshe Sebag. 2020. The Shapley Value of Tuples in Query Answering. In *ICDT*, Vol. 155. 20:1–20:19.

[37] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2010. Generating Databases for Query Workloads. *Proc. VLDB Endow.* 3, 1 (2010), 848–859.

[38] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing Implications of Data Dependencies. *ACM Trans. Database Syst.* 4, 4 (1979), 455–469.

[39] Yashwant K Malaiya, Michael Naixin Li, James M Bieman, and Rick Karcich. 2002. Software reliability growth with test coverage. *IEEE Transactions on Reliability* 51, 4 (2002), 420–426.

[40] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and Explanations in Databases. *PVLDB* 7, 13 (2014), 1715–1716.

[41] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *SIGMOD*. 503–520.

[42] Joan C. Miller and Clifford J. Maloney. 1963. Systematic mistake analysis of digital computer programs. *Commun. ACM* 6, 2 (1963), 58–63.

[43] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2.

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

[45] Judea Pearl et al. 2009. Causal inference in statistics: An overview. *Statistics surveys* 3 (2009), 96–146.

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

[47] Anupam Sanghi, Raghav Sood, Jayant R. Haritsa, and Srikanta Tirthapura. 2018. Scalable and Dynamic Regeneration of Big Data Volumes. In *EDBT*. 301–312.

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

[49] LS SHAPLEY. 1953. A value for n-person games. *Contributions to the Theory of Games* 28 (1953), 307–317.

[50] Boris Trakhtenbrot. 1950. The Impossibility of an Algorithm for the Decidability Problem on Finite Classes. *Proceedings of the USSR Academy of Sciences* 70, 4 (1950), 569–572.

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

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

[53] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.

A APPENDIX

This appendix is part of the full version of the paper "Understanding Queries by Conditional Instances".

A.1 Omitted Pseudocodes and Proof

We given the pseudo code of the algorithms that was omitted from Section 4.2. Namely, Algorithms 4 and 6 that handle the disjunction connective and universal operator in Algorithm 6 (Lines 11 and 15, respectively). We further give the pseudo code and proof of correctness for the procedure that checks whether a c-instance satisfies a query. This procedure, called TREE-SAT, is used in Line 13 of Algorithm 1.

Algorithm 4 Handle-Disjunction

HANDLE-OR($\mathbf{R}, Q, I, f$)

1 ConjTrees = EXPAND-DISJTREE($Q.root$)
2 **for** $T \in ConjTrees$
3   res.extend(TREE-CHASE-BFS($\mathbf{R}, T, I, f$))
4 **return** res

Algorithm 5 Handle-Existential-Operator

HANDLE-EXISTENTIAL($\mathbf{R}, Q, I, f$)

1 **for** $x \in I.domain(Q.root.variable)$
2    $g = f \cup \{Q.root.variable \rightarrow x\}$
3   res.extend(RD-TREE-CHASE-NAIVE($\mathbf{R}, Q.root.child, I, g, limit$))
4 Create a fresh labeled null y in the domain of $Q.root.variable$
5 $g = f \cup \{Q.root.variable \rightarrow y\}$
6 $I.domain(Q.root.variable) = I.domain(Q.root.variable) \cup \{y\}$
7 res.extend(RD-TREE-CHASE-NAIVE($\mathbf{R}, Q.root.child, I, g, limit$))
8 **return** res

Checking if a c-instance satisfies a query. The TREE-SAT procedure is used in Algorithm 1 (Line 13) to verify that a c-instance satisfies the syntax tree of a query. We next describe the pseudo code of the procedure and prove its correctness. The algorithm gets as input the query syntax tree Q, the current c-instance I, and the current homomorphism h. It starts by adding $\exists$ quantifier for every free variable in Q (Lines 1–3) and checks whether Q is a single atom. If it is, the algorithm checks whether this atom is negated or an atomic condition and whether it is contained in the condition of I, $\phi(I)$, or if the atom is not negated and is mapped to a tuple in I. In both cases the algorithm returns True and returns False otherwise (Lines 4–8). If the root of Q is a $\wedge$ node the algorithm checks recursively whether both of its subtrees are satisfied by I and h (Lines 9–10). If the root of Q is a $\vee$ node the algorithm checks recursively whether at least one of its subtrees are satisfied by I and h (Lines 11–12). If the root of Q is a $\exists$ node the algorithm checks recursively whether there is a mapping for the quantified variable to a labeled null x that satisfies the subtree of the quantifier with I and h extended with this mapping (Lines 13–18). If the root of Q is a $\forall$ node the algorithm checks recursively whether all mappings to labeled nulls x for the quantified variable satisfy the subtree of the quantifier with I and h, extended with these mappings (Lines 19–24).

Algorithm 6 Handle-Universal

HANDLE-UNIVERSAL($\mathbf{R}, Q, I, h, limit$)

1   res = [], $Ilist = [I]$
2   **if** $I.domain(Q.root.variable) = \emptyset$
3   res.append(I)
4   **else**
5   **for** $x \in I.domain(Q.root.variable)$
6       $g = h \cup \{Q.root.variable \rightarrow x\}$
7       $cur = []$
8   **for** $J_1 \in Ilist$
9         $Ilist' = $ TREE-CHASE-BFS($\mathbf{R}, Q.root.child, J_1, g, limit$)
10   **for** $J_2 \in Ilist'$
11   **if** IsConsistent(J_2)
12   cur.append(J_2)
13        $Ilist = cur$
14    $res = res \cup Ilist$
15 Create a fresh labeled null y in the domain of $Q.root.variable$
16 $g = h \cup \{Q.root.variable \rightarrow y\}$
17 $I.domain(Q.root.variable) = I.domain(Q.root.variable) \cup \{y\}$
18 $cur = []$
19 **for** $J_1 \in Ilist$
20    $Ilist' = $ TREE-CHASE-BFS($\mathbf{R}, Q.root.child, J_1, g, limit$)
21   **for** $J_2 \in Ilist'$
22   **if** IsConsistent(J_2)
23   cur.append(J_2)
24 $res = res \cup cur$
25 **return** res

PROPOSITION A.1. *Given I, and Q, Algorithm 7 returns True for Q and I iff I satisfies Q.*

PROOF. We prove that Q is satisfied by I with the homomorphism h iff Algorithm 7 returns True for some Q, I, and h using induction over the size of Q. If Q is of size 1, then the algorithm returns True in Line 6 after checking that the only atom in Q has a mapping to the tuple in I if it is not negated, and if it is negated or an atomic condition, it checks whether the global condition of I contains the tuple or atomic condition mapped to the atom in Q. For the induction hypothesis, assume that for every Q of size $< n$, Algorithm 7 returns True for Q and I iff I satisfies Q. Now, suppose Q is of size n.

If $Q = Q_1 \wedge Q_2$, Algorithm 7 returns True in Line 10 iff both Q_1 and Q_2 are satisfied by I and h. According to the induction hypothesis, both Q_1 and Q_2 are satisfied by I and h iff then the algorithm returns True on both.

If $Q = Q_1 \vee Q_2$, Algorithm 7 returns True in Line 10 iff one of Q_1 or Q_2 are satisfied by I and h. According to the induction hypothesis, Q_1 or Q_2 are satisfied by I and h iff then the algorithm returns True for one of them.

If $Q = \exists x. Q'$, Q is satisfied by I and h iff there exists $y \in I.domain(Q.root.variable)$, Q' is satisfied by I and $h \cup \{x = y\}$. In Line 14, the algorithm iterates over the domain of the quantified variable, adds it to the mapping list and checks whether replacing the quantified variable with its current mapping satisfies the subtree rooted at the child of the $\exists$ quantifier. Indeed, since the algorithm tries all mappings of x from the domain, it will try $x = y$. Thus, Algorithm 7 will return True iff Q is satisfied by I and h.

If $Q = \forall x. Q'$, Q is satisfied by I and h iff for all $y \in I.domain(Q.root.variable)$, Q' is satisfied by I and $h \cup \{x = y\}$. In Line 20, the algorithm iterates over the domain of the quantified variable, adds it to the mapping list and checks whether replacing the quantified variable with its current mapping of x satisfies Q'.

Algorithm 7 Tree-SAT

Tree-SAT(Q, I, f)

 Input: Q: a syntax tree of a DRC query;
 I: current c-instance;
 f: current mapping from $\mathcal{V}_Q \cup C_Q \to \mathcal{L}_I \cup C_I$.
 Output: True iff I satisfies Q
1 **if** Q has free variables
2 **for** $x \in FreeVar(Q)$
3   $Q = \exists x Q(x)$
4 **if** Q.root is an atom
5 **if** (Q.negated and $\neg f(Q.root.atom) \in \phi(I)$) or
 ($Q = x \circ y$ and $f(x) \circ f(y) \in \phi(I)$) or
 (Q.negated = False and $f(Q.root.atom) \in I$)
6   **return** True
7 **else**
8   **return** False
9 **if** Q.root.operator $\in \{\wedge\}$
10 **return**
 Tree-SAT($Q.root.lchild, I, f$)$\wedge$ Tree-SAT($Q.root.rchild, I, f$)
11 **elseif** Q.root.operator $\in \{\vee\}$
12 **return**
 Tree-SAT($Q.root.lchild, I, f$)$\vee$ Tree-SAT($Q.root.rchild, I, f$)
13 **elseif** Q.root.operator $\in \{\exists\}$
14 **for** $x \in I.domain(Q.root.variable)$
15   $g = f \cup \{Q.root.variable \to x\}$
16   **if** Tree-SAT($Q.root.child, I, g$)
17   **return** True
18 **return** False
19 **elseif** Q.root.operator $\in \{\forall\}$
20 **for** $x \in I.domain(Q.root.variable)$
21   $g = f \cup \{Q.root.variable \to x\}$
22   **if** *not* Tree-SAT($Q.root.child, I, g$)
23   **return** False
24 **return** True

If it does not, it returns False. If all mappings of x satisfy Q', the algorithm returns True. According to the induction hypothesis, for each such mapping of x, Indeed, since the algorithm tries all mappings of x from the domain, Algorithm 7 will return True iff Q' with this mapping is satisfied by I and $h \cup \{x = y\}$. Thus, Algorithm 7 will return True iff Q is satisfied by I and h.

$\square$

Checking if a c-instance is consistent. To guarantee correctness and reduce the search space, Algorithm 1 verifies that a c-instance is consistent every time adding a c-instance to the queue (Line 13, Line 18). As defined in Definition 5, a c-instance $\mathcal{I}$ is consistent if $PWD(\mathcal{I}) \neq \emptyset$, which is reduced to deciding the satisfiablity of the global condition of $\mathcal{I}$. In our implementation we used the Z3 SMT solver to support complex constraints involving integers, real numbers, and strings.

A.2 Additional Examples

We next provide examples for the algorithms described in the paper.

Example 13 (Example of Algorithm 1). *We demonstrate the operation of Algorithm 1 using the query $Q_B - Q_A$ shown in Figure 3 with its syntax tree in Figure 5. Suppose limit = 20 (basically, the*

limit does not affect the execution), and the algorithm begins with empty h_0 and $\mathcal{I}_0$.

At the beginning (Lines 2-5 in Algorithm 1), for each free variable in $Q_B - Q_A$ (x_1, b_1), a labeled null is created and added to the domain of $\mathcal{I}_0$ and then to h_0. Now we have $\mathcal{I}_0.\text{Dom}(\textit{Bar.name}) = \{x_1\}$, $\mathcal{I}_0.\text{Dom}(\textit{Beer.name}) = \{b_1\}$, $h = \{x_1 \to x_1, b_1 \to b_1\}$.

Then we add $\mathcal{I}_0$ to the queue and start the BFS procedure. The current $\mathcal{I}_0$ has not been visited (Lines 10-12), and it does not satisfy the query tree and is consistent (Line 13), so we call Algorithm 2.

In Algorithm 2, it first goes into the $\exists$ case, and since the domain of drinker names is empty, we can only create new labeled null d_1 and add $d_1 \to d_1$ to g, and do the same for p_1 when we run algorithm 1 recursively. The formula under $\exists d_1, p_1(\cdots)$ has no quantifiers and only conjunction. Therefore, we directly obtain a c-instance from its left branch (Lines 2-7 Algorithm 2) with the tuples $Likes(d_1, b_1)$ and $Serves(x_1, b_1, p_1)$, and the condition d_1 LIKE $'Eve\%'$.

For the next existential quantifier node $\exists x_2$, however, there are two options: we can either map x_2 to the existing labeled null x_1 created earlier, or we can create a new x_2 and add it to the domain of bar names in the instance. It is the same for $\exists p_2$. Hence, we can reach the node $\wedge$ below $\exists x_2, p_2$ with four different mappings for x_2 and p_2: $\{x_2 \to x_1, p_2 \to p_1\}$, $\{x_2 \to x_1, p_2 \to p_2\}$, $\{x_2 \to x_2, p_2 \to p_1\}$, $\{x_2 \to x_2, p_2 \to p_2\}$. Again, we obtain c-instances by adding the atoms on the left branch of the $\wedge$ node. The resulting instances will be inconsistent if we choose to map p_2 to p_1, because there is an atomic condition $p_1 > p_2$, which will be false if we map p_1 and p_2 to the same labeled null.

Continuing, we use the case $\{x_2 \to x_2, p_2 \to p_2\}$ to illustrate. Then, we come to the $\forall d_2$ node. Note that, currently, we have one labeled null d_1 in the domain of drinker names, thus Algorithm 6 will try first mapping d_2 to d_1 and traverse the next node (Lines 5-7), where $\forall p_3$ would first map p_3 to p_1, p_2 one by one and goes to the $\vee$ node, where the disjunction algorithm will enumerate all three cases that convert disjunction to conjunction. Assume that for $\{p_3 \to p_1\}$, we negate the right branch of $\vee$ and keep the left branch the same, then one possible conjunction under the mapping from the left branch can be $Likes(d_1, b_1) \wedge d_2$ LIKE $'Eve\_\%'$ $\wedge Serves(x_1, b_1, p_1)$ (certainly, we cannot have $\neg Likes(d_1, b_1)$ or $\neg Serves(x_1, b_1, p_1)$). We transform the right branch into $\forall x_3 \forall p_4(\neg Serves(x_3, b_1, p_4) \vee p_3 \geq p_4)$ to get negation only on the leaves, and we have:

(1) $Serves(x_1, b_1, p_1) \wedge p_1 \geq p_1$ for $\{x_3 \to x_1, p_4 \to p_1\}$
(2) $\neg Serves(x_2, b_1, p_1) \wedge p_1 \geq p_1$ for $\{x_3 \to x_2, p_4 \to p_1\}$
(3) $\neg Serves(x_1, b_1, p_2) \wedge p_1 \geq p_2$ for $\{x_3 \to x_1, p_4 \to p_2\}$
(4) $Serves(x_2, b_1, p_2) \wedge p_1 \geq p_2$ for $\{x_3 \to x_2, p_4 \to p_2\}$

For $\{p_3 \to p_2\}$, we keep both branches the same. One possible conjunction under the mapping from the left branch can be $Likes(d_1, b_1)$ $\wedge \neg(d_2$ LIKE $'Eve\_\%')$ $\wedge \neg Serves(x_1, b_1, p_2)$; for the right branch, we can map x_3 to x_1 and p_4 to p_1 and thus we get $Serves(x_1, b_1, p_1) \wedge p_2 < p_1$. By merging the resulting instances in each $\wedge$ node, we can obtain the c-instance $\mathcal{I}_2$ in Figure 7.

Example 14 (Example of the optimized approach in Section 4.3). *A partial result of the algorithm is shown in Figure 17, showing two out of the three obtained trees for the subtree of the different query $Q_B - Q_A$ shown in Figure 5. The top tree negates the right subtree below the $\vee$ connective, while the bottom negates the left one,*
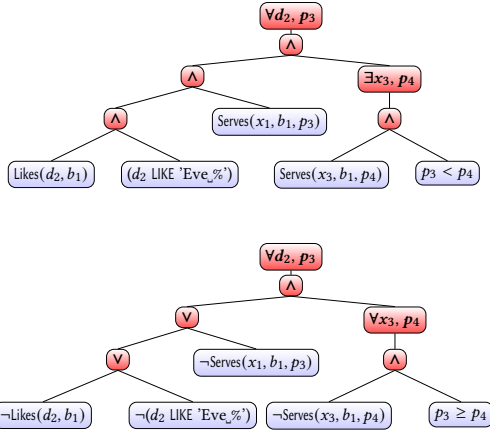
Figure 17: Two of the three conjunctive right subtrees of the syntax tree of $Q_B - Q_A$ depicted in Figure 5. The trees represent $\neg A \wedge B$ and $A \wedge \neg B$, where A (B) is the left (right) subtree of the top $\wedge$ connective.

representing the two cases where $\neg A \wedge B$ and $A \wedge \neg B$. For instance, in the top tree, the formula in the right subtree that was originally $\neg Serves(x_1, b_1, p_3) \vee (\neg Likes(d_2, b_1) \wedge \neg(d_2 \text{ LIKE } 'Eve\_\%'))$ has been converted into $Serves(x_1, b_1, p_3) \wedge (Likes(d_2, b_1) \wedge d_2 \text{ LIKE } 'Eve\_\%')$.

A.3 Query Details

Tables 4 and 5 show the queries that we used in our experiments and their complexity. Note that in order to run our algorithms on TPC-H queries, we modified the original queries and rewrote them in DRC.

Query description	DRC Query	Size	Height	# ∀+∃	# Or	# Or Below ∀ + # ∀
Correct Q_{1A}	$\{(x_1, b_1) \mid \exists d_2, p_3(((Serves(x_1, b_1, p_3) \land d_2\ LIKE\ "Eve\ \%") \land Likes(d_2, b_1)) \land \forall p_4, x_3(\neg serves(x_3, b_1, p_4) \lor p_4 \leq p_3)))\}$	15	9	10	1	3
Wrong Q_{1B}	$\{(x_1, b_1) \mid \exists d_1, p_1(((Serves(x_1, b_1, p_1) \land Likes(d_1, b_1)) \land d_1\ LIKE\ "Eve\%") \land \exists x_2, p_2((p_2 < p_1 \land Serves(x_2, b_1, p_2)) \land x_1\ !=\ x_2))\}$	17	10	11	0	0
Q_{1A}-Q_{1B}	Q_{1A}-Q_{1B}	31	11	20	6	9
Q_{1B}-Q_{1A}	Q_{1B}-Q_{1A}	31	11	20	3	3
Correct Q_{2A}	$\{(b_1) \mid \exists tr_1(Beer(b_1, tr_1) \land \forall td_1 \neg Likes(td_1, b_1))\}$	6	5	4	0	1
Wrong Q_{2B}	$\{(b_1)(\exists x_1, p_1 Serves(x_1, b_1, p_1) \land \neg \exists d_1 Likes(d_1, b_1))\}$	7	5	5	0	1
Q_{2A}-Q_{2B}	Q_{2A}-Q_{2B}	13	6	9	1	3
Q_{2B}-Q_{2A}	Q_{2B}-Q_{2A}	13	6	9	1	3
Correct Q_{3A}	$\{(b_1, x_1) \mid \exists tp_1(Serves(x_1, b_1, tp_1) \land \forall tp_2, tx_2(\neg serves(tx_2, b_1, tp_2) \lor tp_2 \leq tp_1))\}$	10	8	7	1	3
Wrong Q_{3B}	$\{(b_1, x_1) \mid \exists x_2, p_1, p_2(((Serves(x_1, b_1, p_1) \land Serves(x_2, b_1, p_2)) \land p_2 \leq p_1) \land x_1\ ==\ x_2)\}$	12	9	8	0	0
Q_{3A}-Q_{3B}	Q_{3A}-Q_{3B}	21	10	14	4	7
Q_{3B}-Q_{3A}	Q_{3B}-Q_{3A}	21	10	14	1	2
Wrong Q_{3C}	$\{(b_1, x_1) \mid \exists r_1, p_1(Beer(b_1, r_1) \land (Serves(x_1, b_1, p_1) \land \neg \exists x_2, p_2(Serves(x_2, b_1, p_2) \land p_1 < p_2)))\}$	13	10	9	1	3
Q_{3A}-Q_{3C}	Q_{3A}-Q_{3C}	22	11	15	3	6
Q_{3C}-Q_{3A}	Q_{3C}-Q_{3A}	22	11	15	2	5
Correct Q_{4A}	$\{(d_1) \mid \exists ta_1(Drinker(d_1, ta_1) \land \neg \exists tx_1, tt_1(Frequents(d_1, tx_1, tt_1) \land \neg \exists tb_1, tp_1(Likes(d_1, tb_1) \land Serves(tx_1, tb_1, tp_1))))\}$	13	10	9	1	3
Wrong Q_{4B}	$\{(d_1) \mid \exists x_1, b_1(\exists p_1, t_1(Frequents(d_1, x_1, t_1) \land Serves(x_1, b_1, p_1)) \land Likes(d_1, b_1))\}$	10	8	7	0	0
Q_{4A}-Q_{4B}	Q_{4A}-Q_{4B}	23	11	16	3	9
Q_{4B}-Q_{4A}	Q_{4B}-Q_{4A}	23	11	16	2	5
Wrong Q_{4C}	$\{(d_1) \mid \exists x_1(\exists t_1 Frequents(d_1, x_1, t_1) \land \neg(\exists t_2 Frequents(d_1, x_1, t_2) \land \neg \exists b_1, p_1(Likes(d_1, b_1) \land Serves(x_1, b_1, p_1))))\}$	13	8	9	1	1
Q_{4A}-Q_{4C}	Q_{4A}-Q_{4C}	26	11	18	3	9
Q_{4C}-Q_{4A}	Q_{4C}-Q_{4A}	26	11	18	3	6
Wrong Q_{4D}	$\{(d_1)(\exists a_1 Drinker(d_1, a_1) \land \neg \exists b_1(\exists x_1, t_1, p_1(Frequents(d_1, x_1, t_1) \land Serves(x_1, b_1, p_1)) \land \neg Likes(d_1, b_1)))\}$	13	9	9	2	6
Q_{4A}-Q_{4D}	Q_{4A}-Q_{4D}	26	11	18	2	4
Q_{4D}-Q_{4A}	Q_{4D}-Q_{4A}	26	11	18	4	11
Correct Q_{5A}	$\{(d_1) \mid \exists ta_1(Drinker(d_1, ta_1) \land \neg \exists tx_1(\exists tb_1, tp_1(Likes(d_1, tb_1) \land Serves(tx_1, tb_1, tp_1)) \land \neg \exists tt_1 Frequents(d_1, tx_1, tt_1)))\}$	13	9	9	2	5
Wrong Q_{5B}	$\{(d_1) \mid \exists x_1, t_1(Frequents(d_1, x_1, t_1) \land \neg \exists x_2(\exists b_1, p_1(Likes(d_1, b_1) \land Serves(x_2, b_1, p_1)) \land \exists t_2 \neg Frequents(d_1, x_2, t_2)))\}$	14	10	10	2	6
Q_{5A}-Q_{5B}	Q_{5A}-Q_{5B}	27	11	19	3	8
Q_{5B}-Q_{5A}	Q_{5B}-Q_{5A}	27	11	19	3	9
Wrong Q_{5C}	$\{(d_1)(\exists b_1, x_1, t_1, p_1((Frequents(d_1, x_1, t_1) \land Serves(x_1, b_1, p_1)) \land Likes(d_1, b_1)) \land \neg \exists x_2, b_2(\exists p_2(Likes(d_1, b_2) \land Serves(x_2, b_2, p_2)) \land \neg \exists p_3, t_2((Frequents(d_1, x_2, t_2) \land Serves(x_2, b_2, p_3)) \land Likes(d_1, b_2))))\}$	25	10	17	2	5
Q_{5A}-Q_{5C}	Q_{5A}-Q_{5C}	38	11	26	7	13
Q_{5C}-Q_{5A}	Q_{5C}-Q_{5A}	38	11	26	3	8
Wrong Q_{5D}	$\{(d_1)(\exists b_1, x_1, p_1(Likes(d_1, b_1) \land Serves(x_1, b_1, p_1)) \land \neg \exists x_2(\exists b_2, p_2(Likes(d_1, b_2) \land Serves(x_2, b_2, p_2)) \land \neg \exists t_1 Frequents(d_1, x_2, t_1)))\}$	17	8	12	2	5
Q_{5A}-Q_{5D}	Q_{5A}-Q_{5D}	30	10	21	4	10
Q_{5D}-Q_{5A}	Q_{5D}-Q_{5A}	30	10	21	3	8

Table 4: Beers queries used in the experiments and their complexity measures.

Query description	DRC Query	Size	Height	# ∀+∃	# Or	# Or Below ∀ + # ∀
Correct Q_{4A}	$\{(o_1, o_2)(\exists o_3, o_6(orders(o_1, o_3, *, *, o_6, o_2, *, *, *) \wedge (19930701 \leq o_6 \wedge o_6 < 19931001)) \wedge \exists l_2, l_3, l_{12}, l_{13}(lineitem(o_1, l_2, l_3, *, *, *, *, *, *, *, *, l_{12}, l_{13}, *, *, *) \wedge l_{12} < l_{13}))\}$	17	9	12	0	0
Wrong Q_{4B}	$\{(o_1, o_2)(\exists o_3, o_6(orders(o_1, o_3, *, *, o_6, o_2, *, *, *) \wedge (19930701 \leq o_6 \wedge o_6 < 19931001)) \wedge \exists l_2, l_3, l_{12}, l_{13}(lineitem(o_1, l_2, l_3, *, *, *, *, *, *, *, *, l_{12}, l_{13}, *, *, *) \wedge l_{13} < l_{12}))\}$	17	9	12	0	0
Q_{4A}-Q_{4B}	Q_{4A}-Q_{4B}	33	10	23	4	8
Q_{4B}-Q_{4A}	Q_{4B}-Q_{4A}	33	10	23	4	8
Wrong Q_{4C}	$\{(o_1, o_2)(\exists o_3, o_6(orders(o_1, o_3, *, *, o_6, o_2, *, *, *) \wedge (19930701 \leq o_6 \wedge o_6 < 19931001)) \wedge \neg\exists l_2, l_3, l_{12}, l_{13}(lineitem(o_1, l_2, l_3, *, *, *, *, *, *, *, *, l_{12}, l_{13}, *, *, *) \wedge l_{12} < l_{13}))\}$	17	9	12	1	5
Q_{4A}-Q_{4C}	Q_{4A}-Q_{4C}	33	10	23	3	3
Q_{4C}-Q_{4A}	Q_{4C}-Q_{4A}	33	10	23	5	13
Correct Q_{16A}	$\{(p_4, p_5, p_6, ps_2) \mid \exists p_1(\exists p_2((part(p_1, p_2, *, p_4, p_5, p_6, *, *, *) \wedge (49 == p_6 \vee 14 == p_6)) \wedge ("Brand\#45" \; != \; p_4 \wedge p_5 \; LIKE \; "MEDIUM \; POLISHED\%")) \wedge (partsupp(p_1, ps_2, *, *, *) \wedge \neg\exists s_7(supplier(ps_2, *, *, *, *, *, s_7) \wedge s_7 \; LIKE \; "\%complain")))\}$	22	11	14	2	2
Wrong Q_{16B}	$\{(p_4, p_5, p_6, ps_2) \mid \exists p_1(\exists p_2((part(p_1, p_2, *, p_4, p_5, p_6, *, *, *) \wedge (49 == p_6 \vee 14 == p_6)) \wedge ("Brand\#45" \; != \; p_4 \wedge p_5 \; LIKE \; "MEDIUM \; POLISHED\%")) \wedge (partsupp(p_1, ps_2, *, *, *) \wedge \neg\exists s_7(supplier(ps_2, *, *, *, *, *, s_7) \wedge s_7 \; LIKE \; "\%complain")))\}$	22	11	14	2	2
Q_{16A}-Q_{16B}	Q_{16A}-Q_{16B}	41	12	25	7	6
Q_{16B}-Q_{16A}	Q_{16B}-Q_{16A}	41	12	25	7	6
Wrong Q_{16C}	$\{(p_4, p_5, p_6, ps_2) \mid \exists p_1(\exists p_2((part(p_1, p_2, *, p_4, p_5, p_6, *, *, *) \wedge (49 == p_6 \vee 14 == p_6)) \wedge ("Brand\#45" \; != \; p_4 \wedge p_5 \; LIKE \; "MEDIUM \; POLISHED\%")) \wedge (partsupp(p_1, ps_2, *, *, *) \wedge \exists s_7(supplier(ps_2, *, *, *, *, *, s_7) \wedge \neg s_7 \; LIKE \; "\%complain")))\}$	22	11	14	1	0
Q_{16A}-Q_{16C}	Q_{16A}-Q_{16C}	41	12	25	8	8
Q_{16C}-Q_{16A}	Q_{16C}-Q_{16A}	41	12	25	6	4
Correct Q_{19A}	$\{(l_6, l_7) \mid \exists l_1, l_2, l_4, l_5, l_{15}, p_4, p_6, p_7$ $((lineitem(l_1, l_2, *, l_4, l_5, l_6, l_7, *, *, *, *, *, *, "DELIVER \; IN \; PERSON", "AIR", *) \wedge$ $part(l_2, *, *, p_4, *, p_6, p_7, *, *)) \wedge ((("Brand\#12" == p_4 \wedge p_7 \; LIKE \; "SM\%") \wedge (l_5 <= 11 \wedge p_6 <= 5))$ $\vee (("Brand\#23" == p_4 \wedge p_7 \; LIKE \; "MED\%") \wedge ((10 \leq l_5 \wedge l_5 <= 20) \wedge p_6 <= 10))))\}$	31	16	20	1	0
Wrong Q_{19B}	$\{(l_6, l_7) \mid \exists l_1, l_2, l_4, l_5, l_{15}, p_4, p_6, p_7$ $((lineitem(l_1, l_2, *, l_4, l_5, l_6, l_7, *, *, *, *, *, *, "DELIVER \; IN \; PERSON", "AIR", *) \wedge$ $part(l_2, *, *, p_4, *, p_6, p_7, *, *)) \wedge ((("Brand\#12" == p_4 \wedge p_7 \; LIKE \; "SM\%") \wedge (l_5 <= 10 \wedge p_6 <= 5))$ $\vee (("Brand\#234" == p_4 \wedge p_7 \; LIKE \; "MED\%") \wedge (l_5 <= 20 \wedge p_6 <= 10))))\}$	29	15	19	1	0
Q_{19A}-Q_{19B}	Q_{19A}-Q_{19B}	59	17	38	9	9
Q_{19B}-Q_{19A}	Q_{19B}-Q_{19A}	59	17	38	10	9
Wrong Q_{19C}	$\{(l_6, l_7) \mid \exists l_1, l_2, l_4, l_5, l_{15}, p_4, p_6, p_7$ $((lineitem(l_1, l_2, *, l_4, l_5, l_6, l_7, *, *, *, *, *, *, "DELIVER \; IN \; PERSON", "AIR", *) \wedge$ $part(l_2, *, *, p_4, *, p_6, p_7, *, *)) \wedge (("Brand\#12" == p_4 \wedge p_7 \; LIKE \; "SM\%") \wedge$ $(l_5 <= 11 \wedge p_6 <= 5)))\}$	21	14	15	0	0
Q_{19A}-Q_{19C}	Q_{19A}-Q_{19C}	51	17	34	6	9
Q_{19C}-Q_{19A}	Q_{19C}-Q_{19A}	51	17	34	9	9
Correct Q_{21A}	$\{(s_1, s_2, o_1)((\exists l_{12}, l_{13}(lineitem(o_1, *, s_1, *, *, *, *, *, *, *, *, l_{12}, l_{13}, *, *, *) \wedge l_{12} < l_{13}) \wedge$ $\exists ll_3, ll_{12}, ll_{13}(lineitem(o_1, *, ll_3, *, *, *, *, *, *, *, *, ll_{12}, ll_{13}, *, *, *) \wedge ll_3 \; != \; s_1)) \wedge$ $((orders(o_1, *, "F", *, *, *, *, *, *) \wedge \exists s_4(supplier(s_1, s_2, *, s_4, *, *, *) \wedge$ $nation(s_4, "SAUDI \; ARABIA", *, *))) \wedge \neg\exists lll_3, lll_{12}, lll_{13}$ $(lineitem(o_1, *, lll_3, *, *, *, *, *, *, *, *, lll_{12}, lll_{13}, *, *, *) \wedge (lll_{12} < lll_{13} \wedge lll_3 \; != \; s_1))))\}$	31	11	21	2	4
Wrong Q_{21B}	$\{(s_1, s_2, o_1)((\exists l_{12}, l_{13}(lineitem(o_1, *, s_1, *, *, *, *, *, *, *, *, l_{12}, l_{13}, *, *, *) \wedge l_{12} < l_{13}) \wedge$ $(orders(o_1, *, "F", *, *, *, *, *, *) \wedge \exists s_4(supplier(s_1, s_2, *, s_4, *, *, *) \wedge$ $nation(s_4, "SAUDI \; ARABIA", *, *)))) \wedge \exists lll_3, lll_{12}, lll_{13}$ $(lineitem(o_1, *, lll_3, *, *, *, *, *, *, *, *, lll_{12}, lll_{13}, *, *, *) \wedge (lll_{13} \leq lll_{12} \wedge lll_3 \; != \; s_1)))\}$	24	10	16	0	0
Q_{21A}-Q_{21B}	Q_{21A}-Q_{21B}	53	12	35	9	13
Q_{21B}-Q_{21A}	Q_{21B}-Q_{21A}	53	12	35	7	9
Wrong Q_{21C}	$\{(s_1, s_2, o_1)(\exists l_{12}, l_{13}(lineitem(o_1, *, s_1, *, *, *, *, *, *, *, *, l_{12}, l_{13}, *, *, *) \wedge l_{12} < l_{13}) \wedge (\exists o_3 orders(o_1, *, o_3, *, *, *, *, *, *) \wedge \exists s_4(supplier(s_1, s_2, *, s_4, *, *, *) \wedge nation(s_4, "SAUDI \; ARABIA", *, *))))\}$	16	8	11	0	0
Q_{21A}-Q_{21C}	Q_{21A}-Q_{21C}	45	12	30	6	10
Q_{21C}-Q_{21A}	Q_{21C}-Q_{21A}	45	12	30	7	9

Table 5: TPC-H queries used in the experiments and their complexity measures. * denotes "don't care".