Computing Approximate Certain Answers over Incomplete Databases

Sergio Greco, Cristian Molinaro, and Irina Trubitsyna
{greco,cmolinaro,trubitsyna}@dimes.unical.it

DIMES, Università della Calabria, 87036 Rende (CS), Italy

Abstract. Certain answers are a widely accepted semantics of query answering over incomplete databases. Since their computation is a coNP-hard problem, recent research has focused on developing evaluation algorithms with *correctness guarantees*, that is, techniques computing a sound but possibly incomplete set of certain answers.
In this paper, we show how novel evaluation algorithms with correctness guarantees can be developed leveraging conditional tables and the conditional evaluation of queries, while retaining polynomial time data complexity.

1 Introduction

Incomplete information arises naturally in many database applications, such as data integration [9, 21], data exchange [3, 10, 17, 7], inconsistency management [4, 5, 16, 20], data cleaning [11], ontological reasoning [6, 8], and many others.

A principled way of answering queries over incomplete databases is to compute *certain answers*, which are query answers that can be obtained from all the complete databases an incomplete database represents. This central notion is illustrated below.

Example 1. Consider the database D consisting of the two unary relations $R = \{\langle a\rangle\}$ and $S = \{\langle\bot\rangle\}$, where $\bot$ is a null value. Under the missing value interpretation of nulls (i.e., a value for $\bot$ exists but is unknown), D represents all the databases obtained by replacing $\bot$ with an actual value.

A certain answer to a query is a tuple that belongs to the query answers for every database represented by D.

For instance, consider the query $\sigma_{\$1=a}(R)$, selecting the tuples in R whose first value is equal to a. The certain answers to the query are $\{\langle a\rangle\}$, because no matter how $\bot$ is replaced, the query answers always contain $\langle a\rangle$. □

For databases containing (labeled) nulls, certain answers to positive queries can be easily computed in polynomial time by applying a "naive" evaluation, that is, treating nulls as standard constants. However, for more general queries with negation the problem becomes coNP-hard.

To make query answering feasible in practice, one might resort to SQL's evaluation, but unfortunately, the way SQL behaves in the presence of nulls may result in wrong answers.

Specifically, as evidenced in [25], there are two ways in which certain answers and SQL's evaluation may differ: *(i)* SQL can miss some of the tuples that belong to certain answers, thus producing *false negatives*, or *(ii)* SQL can return some tuples that do not belong to certain answers, that is, *false positives*. While the first case can be seen as an under-approximation of certain answers (a sound but possibly incomplete set of certain answers is returned), the second scenario must be avoided, as the result might contain plain incorrect answers, that is, tuples that are not certain. The experimental analysis in [18] showed that false positive are a real problem for queries involving negation—they were always present and sometimes they constitute almost 100% of the answers.

Example 2. Consider again the database D of Example 1. There are no certain answers to the query $R - S$, as when $\perp$ is replaced with a, the query answers are the empty set.

Assuming that R and S's attribute is called A, the same query can be expressed in SQL as follows:

```
SELECT R.A FROM R WHERE NOT EXISTS (
  SELECT * FROM S WHERE R.A = S.A )
```

However, the answer to the query above is $\langle a \rangle$, which is not a certain answer. The problem with the SQL semantics is that every comparison involving at least one null evaluates to the truth value *unknown*, then 3-valued logic is used to evaluate the classical logical connectives *and*, *or*, and *not*, and eventually only those tuples whose condition evaluates to *true* are kept.

Going back to the query above, the nested subquery compares $\perp$ with a, and since such a comparison evaluates to *unknown* (because a null is involved) then the result of the nested subquery is empty. As a consequence, the final result of the query is $\langle a \rangle$. □

Thus, on the one hand, SQL's evaluation is efficient but flawed, on the other hand, certain answers are a principled semantics but with high complexity. One way of dealing with this issue is to develop polynomial time evaluation algorithms computing approximate certain answers.

In this regard, there has been recent work on evaluation algorithms with *correctness guarantees*, that is, techniques providing a sound but possibly incomplete set of certain answers [18, 24, 25]. Such approaches consists in rewriting a query Q into two queries $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ (resp., Q^{+} and $Q^{?}$) computing approximations of certainly true and certainly false answers (resp., certainly true and possible answers). However, there are still very simple queries for which the approximation can be unsatisfactory.

Example 3. Consider the database D of Example 1 and the query $R - \sigma_{\$1=b}(S)$. In this case, the certain answers are $\{\langle a \rangle\}$. However, the approximation provided by the approach in [18] is the empty set. □

In this paper, we show how conditional tables and the conditional evaluation of queries [19] can be leveraged to develop new approximation algorithms with correctness guarantees. The approach allows us to keep track of useful information that can be profitably used to determine if a tuple is a certain answer. The very basic idea is illustrated in the following example.

Example 4. Consider again the database D of Example 1 and the query $R - \sigma_{\$1=b}(S)$. The conditional evaluation of the query is carried out by applying the "conditional" counterpart of each relational algebra operator. Rather than returning a set of tuples, the conditional evaluation of a relational algebra operator returns pairs of the form $\langle t, \varphi \rangle$, where t is a tuple and φ is an expression stating under which conditions t can be derived.

Considering the query above, first the conditional evaluation of $\sigma_{\$1=b}(S)$ is performed, which gives $\langle \bot, \varphi' \rangle$, where φ' is the condition $(\bot = b)$. Then, the conditional evaluation of the difference operator is carried out, yielding $\langle a, \varphi'' \rangle$, where φ'' is the condition $(\bot \neq a) \vee (\bot \neq b)$. Indeed, this is the result of the conditional evaluation of the whole query. □

As we will show, conditions are valuable information that can exploited to determine which tuples are certain answers. For instance, from an analysis of φ'' in the example above, one can realize that the condition is always true—thus, $\langle a \rangle$ is a certain answer. There might be different ways of evaluating tuples' conditions. In this paper, we propose some strategies.

2 Preliminaries

In this section, we recall basic notions on incomplete databases, conditional tables, and approximation algorithms for query answering over incomplete databases.

2.1 Incomplete databases

Basics. We assume the existence of the following disjoint countably infinite sets: a set Const of *constants* and a set Null of *(labeled) nulls*. Nulls are denoted by the symbol $\bot$ subscripted.

A *tuple* t of arity k is an element of $(\mathsf{Const} \cup \mathsf{Null})^k$, where k is a non-negative integer. The i-th element of t is denoted as $t[i]$, where $1 \leq i \leq k$. Given a possibly empty ordered sequence Z of integers $i_1, \ldots, i_h$ in the range $[1..k]$, we use $t[Z]$ to denote the tuple $\langle t[i_1], \ldots, t[i_h] \rangle$.

A *relation* of arity k is a finite set of tuples of arity k. A *relational schema* is a set of *relation names*, each associated with a non-negative arity. A *database* D associates a relation R^D of arity k with each relation name R of arity k. With a slight abuse of notation, when the database is clear from the context, we simply write R instead of R^D for the relation itself. The arity of R is denoted as $ar(R)$.

The sets of all constants and nulls occurring in a database D are denoted by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$, respectively. The *active domain* of D is $\mathsf{adom}(D) = \mathsf{Const}(D) \cup \mathsf{Null}(D)$. If $\mathsf{Null}(D) = \emptyset$, we say that D is *complete*. Likewise, a relation is *complete* if it does not contain nulls. Databases as defined above have been called also *naive tables*, *V-tables*, and *e-tables* [19, 1, 15].

A *valuation* ν is a mapping from $\mathsf{Const} \cup \mathsf{Null}$ to Const s.t. $\nu(c) = c$ for every $c \in \mathsf{Const}$. Valuations can be applied to tuples, relations, and databases in the obvious way. For instance, the result of applying ν to a database D, denoted $\nu(D)$, is the complete database obtained from D by replacing every null $\bot_i$ with $\nu(\bot_i)$.

The semantics of a database D is given by the set of complete databases $\{\nu(D) \mid \nu \text{ is a valuation}\}$, which are also called *possible worlds*. Indeed, this is the semantics under the missing value interpretation of nulls (i.e., every null stands for a value that exists but is unknown), and is referred to as the closed-world semantics of incompleteness.[1]

Query answering. We consider queries expressed with the relational algebra, that is, by means of the following operators: selection σ, projection π, cartesian product $\times$, union $\cup$, intersection $\cap$, and difference $-$. In the rest of the paper, a query is understood to be a relational algebra expression built up from the above operators, unless otherwise indicated. The result of evaluating a query Q on a database D, treating nulls as standard constants (i.e., every (labeled) null or constant is equal to itself and different from every other element of $\mathsf{Const} \cup \mathsf{Null}$), is denoted as $Q(D)$. A query Q returning k-tuples is said to be of arity k, and $ar(Q)$ denotes its arity.

A widely accepted semantics of query answering relies on the notion of a certain answer. The *certain answers* to a query Q on a database D are the tuples in $\bigcap\{Q(\nu(D)) \mid \nu \text{ is a valuation}\}$. Computing certain answers is coNP-hard (data complexity), even in the case of the more restricted Codd tables, that is, databases as defined above where the same null cannot occur multiple times [2].

For query answering, we will use a more general notion first proposed in [26] and called *certain answers with nulls* in [25].

Definition 1. *Given a query Q and a database D, the* certain answers with nulls*, denoted $\mathsf{cert}(Q, D)$, is the set of all tuples t such that $\nu(t) \in Q(\nu(D))$ for every valuation ν.*

As opposed to the more commonly used notion of certain answers, certain answers with nulls allow tuples with nulls in query answers. Consider for instance the relation $R = \{\langle a, \bot\rangle, \langle b, c\rangle\}$ and the identity query. The certain answers consist only of $\langle b, c\rangle$, but it is more intuitive to return the entire relation, as we are certain that its tuples are query answers. In this case, the certain answer with nulls are both $\langle a, \bot\rangle$ and $\langle b, c\rangle$. We refer the interested reader to [22, 23] for a discussion of other drawbacks of the standard notion of certain answers.

Below we report the definition of evaluation algorithm with *correctness guarantees* [18], which we use as the soundness property an approximation algorithm must satisfy.

Definition 2. *A query evaluation algorithm has* correctness guarantees *for a query Q if for every database D it returns a subset of $\mathsf{cert}(Q, D)$.*

When a query evaluation algorithm has correctness guarantees for every query, we say that it has correctness guarantees.

[1] Under the open-world semantics of incompleteness, which we do not consider in this paper, the semantics of D is $\{\nu(D) \cup D' \mid \nu \text{ is a valuation and } D' \text{ is a complete database}\}$.

2.2 Conditional tables

Syntax and semantics. Conditional tables have been proposed in [19].[2] Essentially, they are relations (as defined in the previous subsection, thus possibly containing nulls) extended by one additional special column that specifies a "condition" for each tuple.

Formally, let $\mathcal{E}$ be the set of all expressions, called *conditions*, that can be built using the standard logical connectives $\wedge$, $\vee$, and $\neg$ with expressions of the form $(\alpha = \beta)$, $(\alpha \neq \beta)$, true, and false, where $\alpha, \beta \in \mathsf{Const} \cup \mathsf{Null}$. We say that a valuation ν *satisfies* a condition φ, denoted $\nu \models \varphi$, if its assignment of constants to nulls makes φ true.

A *conditional tuple* (*c-tuple* for short) of arity k ($k \geq 0$) is a pair $\langle t, \varphi \rangle$, where t is a tuple of arity k and $\varphi \in \mathcal{E}$. Notice that φ may involve nulls and constants not necessarily appearing in t—e.g., t is the tuple $\langle a, \bot_1 \rangle$ and φ is the condition $(\bot_2 = c) \wedge (\bot_1 \neq \bot_3)$.

A *conditional table* (*c-table* for short) of arity k is a finite set of c-tuples of arity k. A *conditional database* C associates a c-table R^C of arity k with each relation name R of arity k. With a slight abuse of notation, when the conditional database is clear from the context, we simply write R instead of R^C for the c-table itself.

Let T be a conditional table. The result of applying a valuation ν to T is

$$\nu(T) = \{\nu(t) \mid \langle t, \varphi \rangle \in T \text{ and } \nu \models \varphi\}.$$

Thus, $\nu(T)$ is the (complete) relation obtained from T by keeping only the c-tuples in T whose condition is satisfied by ν, and applying ν to such c-tuples.

The set of complete relations represented by T is $rep(T) = \{\nu(T) \mid \nu \text{ is a valuation}\}$.

Likewise, a conditional database $C = \{T_1, \ldots, T_m\}$ represents the following set of complete databases: $rep(C) = \{\{\nu(T_1), \ldots, \nu(T_m)\} \mid \nu \text{ is a valuation}\}$.

Conditional evaluation. Below we recall the *conditional evaluation* of a query over a conditional database (see [19, 15]). Basically, it consists in evaluating the relational algebra operators so that they can take c-tables as input and return a c-table as output. The conditional evaluation of a query over a conditional database is then simply obtained by applying the conditional evaluation of each operator.

Let T_1 and T_2 be c-tables of arity n and m, respectively. In the definitions below, for the union and difference operators it is assumed that $n = m$. For projection, Z is a possibly empty ordered sequence of integers in the range $[1..n]$. For selection, θ is a Boolean combination of expressions of the form $(\$i = \$j)$, $(\$i = c)$, $(\$i \neq \$j)$, $(\$i \neq c)$, where $1 \leq i, j \leq n$, and $c \in \mathsf{Const}$. The conditional evaluation of a relational algebra operator op is denoted as $\dot{op}$. In the following, given two tuples t_1 and t_2 of arity n, we use $(t_1 = t_2)$ as a shorthand for the condition $\bigwedge_{i \in [1..n]} (t_1[i] = t_2[i])$.

- *Projection:*
$$\dot{\pi}_Z(T_1) = \{\langle t[Z], \varphi \rangle \mid \langle t, \varphi \rangle \in T_1\}.$$
- *Selection:*
$$\dot{\sigma}_\theta(T_1) = \{\langle t, \varphi' \rangle \mid \langle t, \varphi \rangle \in T_1 \text{ and } \varphi' = \varphi \wedge \theta(t)\},$$
where $\theta(t)$ is the condition obtained from θ by replacing every $\$i$ with $t[i]$.

[2] A generalization of conditional tables augmenting them with "global conditions" has been proposed in [14], see also [15].

– *Union:*

$$T_1 \dot{\cup} T_2 = \{\langle t, \varphi\rangle \mid \langle t, \varphi\rangle \in T_1 \text{ or } \langle t, \varphi\rangle \in T_2\}.$$

– *Difference:*

$$T_1 \dot{-} T_2 = \{\langle t_1, \varphi'\rangle \mid \langle t_1, \varphi_1\rangle \in T_1 \text{ and } \varphi' = \varphi_1 \wedge \varphi_{t_1,T_2}\},$$

where $\varphi_{t_1,T_2} = \bigwedge_{\langle t_2,\varphi_2\rangle \in T_2} \neg(\varphi_2 \wedge (t_1 = t_2))$.

– *Cartesian product:*

$$T_1 \dot{\times} T_2 = \{\langle t_1 \circ t_2, \varphi_1 \wedge \varphi_2\rangle \mid \langle t_1, \varphi_1\rangle \in T_1,\ \langle t_2, \varphi_2\rangle \in T_2\},$$

where $t_1 \circ t_2$ is the tuple obtained as the concatenation of t_1 and t_2.

The result of the conditional evaluation of a query Q over a conditional database C is denoted as $\dot{Q}(C)$. Notice that $\dot{Q}(C)$ is a c-table. For a fixed query Q and a conditional database C, $\dot{Q}(C)$ can be evaluated in polynomial time in the size of C (see [15]). The size of a conditional table T is $||T|| = |T| + \sum_{\langle t,\varphi\rangle \in T} ||\varphi||$, where $|T|$ is the number of c-tuples in T and $||\varphi||$ is the length in symbols of condition φ. The size of a conditional database C is $||C|| = \sum_{T_i \in C} ||T_i||$.

Recall that c-tables are a *strong representation system* for the relational algebra, that is, for every relational algebra query Q and conditional database C, it is possible to compute a c-table T s.t. $rep(T) = \{Q(D) \mid D \in rep(C)\}$. Indeed, such a c-table can be computed through the conditional evaluation, that is, $T = \dot{Q}(C)$.

In the rest of the paper we assume that every selection condition is a conjunction of expressions of the form $(\$i = \$j)$, $(\$i = c)$, $(\$i \neq \$j)$, and $(\$i \neq c)$. There is no loss of generality, as an arbitrary selection $\sigma_\theta(R)$ can be rewritten as follows to comply with our assumption. First, θ is rewritten in disjunctive normal form (DNF), that is, into a formula $\theta_1 \vee \cdots \vee \theta_m$, where each θ_i is a conjunction of expressions of the form $(\$i = \$j)$, $(\$i = c)$, $(\$i \neq \$j)$, and $(\$i \neq c)$. Then, $\sigma_\theta(R)$ is replaced by $\sigma_{\theta_1}(R) \cup \cdots \cup \sigma_{\theta_m}(R)$. Even though the conversion to DNF can lead to an exponential blow-up in size, this does not affect the data complexity (as the query is fixed).

3 Approximation Algorithms

In this section, we show how to exploit conditional tables to compute a sound (but possibly incomplete) set of certain answers with nulls. The basic idea is to rely on the conditional evaluation of the relational algebra operators, in order to keep track of how each tuple is derived during query evaluation, and then apply a strategy to evaluate conditions, so that each tuple is eventually associated with a truth value. Tuples that are associated with the condition `true` are certain answers with nulls.

We start by introducing an explicit conditional evaluation for intersection. In the previous section, we recalled the conditional evaluation of different relational algebra

operators. Clearly, even if intersection was not reported, it can be expressed in terms of the other operators. Let T_1 and T_2 be c-tables of arity n. Then,

$$T_1 \dot{\cap} T_2 = \{\langle t_1, \varphi' \rangle \mid \langle t_1, \varphi_1 \rangle \in T_1, \langle t_2, \varphi_2 \rangle \in T_2, \varphi' = \varphi_1 \wedge \varphi_2 \wedge (t_1 = t_2)\}.$$

We slightly generalize c-tables to allow also unknown as a condition. Thus, from now on, $\mathcal{E}$ is the set of all expressions that can be built using the standard logical connectives with expressions, called *atomic* conditions, of the form $(\alpha = \beta)$, $(\alpha \neq \beta)$, true, false, and unknown, where $\alpha, \beta \in \mathsf{Const} \cup \mathsf{Null}$. We will discuss different strategies to "evaluate" conditions, that is, to reduce them to either true or false or unknown—as shown in the following, tuples having condition true are certain answers with nulls.

We assume the following strict ordering false $<$ unknown $<$ true.

The *three-valued evaluation* of a condition $\varphi \in \mathcal{E}$, denoted $\mathsf{eval}(\varphi)$, is defined inductively as follows:

- $\mathsf{eval}((\alpha = \beta)) = \begin{cases} \mathsf{true} & \text{if } \alpha = \beta, \\ \mathsf{false} & \text{if } \alpha \neq \beta \text{ and } \alpha, \beta \in \mathsf{Const}, \\ \mathsf{unknown} & \text{otherwise.} \end{cases}$
- $\mathsf{eval}((\alpha \neq \beta)) = \begin{cases} \mathsf{true} & \text{if } \alpha \neq \beta \text{ and } \alpha, \beta \in \mathsf{Const}, \\ \mathsf{false} & \text{if } \alpha = \beta, \\ \mathsf{unknown} & \text{otherwise.} \end{cases}$
- $\mathsf{eval}((\varphi_1 \wedge \varphi_2)) = \min\{\mathsf{eval}(\varphi_1), \mathsf{eval}(\varphi_2)\}$.
- $\mathsf{eval}((\varphi_1 \vee \varphi_2)) = \max\{\mathsf{eval}(\varphi_1), \mathsf{eval}(\varphi_2)\}$.
- $\mathsf{eval}((\neg\varphi)) = \begin{cases} \mathsf{true} & \text{if } \mathsf{eval}(\varphi) = \mathsf{false}, \\ \mathsf{false} & \text{if } \mathsf{eval}(\varphi) = \mathsf{true}, \\ \mathsf{unknown} & \text{otherwise.} \end{cases}$
- $\mathsf{eval}(v) = v$ for $v \in \{\mathsf{true}, \mathsf{unknown}, \mathsf{false}\}$.

The basic idea of our first evaluation algorithm, which we call *naive evaluation*, is to perform the three-valued evaluation of conditions after each relational algebra operator is applied (that is, after its conditional evaluation). One interesting fact about the naive evaluation is that it is equivalent to the evaluation algorithm of [18].

Given a c-tuple $\mathbf{t} = \langle t, \varphi \rangle$, with a slight abuse of notation, we use $\mathsf{eval}(\mathbf{t})$ to denote $\langle t, \mathsf{eval}(\varphi) \rangle$. Likewise, given a conditional table T, $\mathsf{eval}(T)$ denotes $\{\mathsf{eval}(\mathbf{t}) \mid \mathbf{t} \in T\}$.

To define the naive evaluation, we first provide the following definitions:

$$\begin{aligned}
&\mathsf{Eval}^{\mathsf{n}}(R) = \mathsf{eval}(R) \\
&\mathsf{Eval}^{\mathsf{n}}(Q_1 \cup Q_2) = \mathsf{eval}(\mathsf{Eval}^{\mathsf{n}}(Q_1) \ \dot{\cup} \ \mathsf{Eval}^{\mathsf{n}}(Q_2)) \\
&\mathsf{Eval}^{\mathsf{n}}(Q_1 \cap Q_2) = \mathsf{eval}(\mathsf{Eval}^{\mathsf{n}}(Q_1) \ \dot{\cap} \ \mathsf{Eval}^{\mathsf{n}}(Q_2)) \\
&\mathsf{Eval}^{\mathsf{n}}(Q_1 - Q_2) = \mathsf{eval}\big(\mathsf{Eval}^{\mathsf{n}}(Q_1) \ \dot{-} \ \mathsf{Eval}^{\mathsf{n}}(Q_2)\big) \\
&\mathsf{Eval}^{\mathsf{n}}(Q_1 \times Q_2) = \mathsf{eval}\big(\mathsf{Eval}^{\mathsf{n}}(Q_1) \ \dot{\times} \ \mathsf{Eval}^{\mathsf{n}}(Q_2)\big) \\
&\mathsf{Eval}^{\mathsf{n}}(\sigma_\theta(Q)) = \mathsf{eval}(\dot{\sigma}_\theta(\mathsf{Eval}^{\mathsf{n}}(Q))) \\
&\mathsf{Eval}^{\mathsf{n}}(\pi_Z(Q)) = \mathsf{eval}(\dot{\pi}_Z(\mathsf{Eval}^{\mathsf{n}}(Q)))
\end{aligned}$$

Given a relation R, we define the c-table $\overline{R} = \{\langle t, \mathsf{true} \rangle \mid t \in R\}$. Analogously, given a database D, we define $\overline{D}$ as the conditional database obtained from D by replacing every relation R of it with $\overline{R}$. Here the basic idea is to convert a database D into

a simple conditional database $\overline{D}$ where all conditions are true, so that $\overline{D}$ can be used as the starting point for the conditional evaluation of queries.

Given a query Q and a database D, we use $\mathsf{Eval}^{\mathsf{n}}(Q, D)$ to denote the result of evaluating $\mathsf{Eval}^{\mathsf{n}}(Q)$ over $\overline{D}$. Finally, we define:

$$\mathsf{Eval}^{\mathsf{n}}_{\mathsf{t}}(Q, D) = \{t \mid \langle t, \mathsf{true}\rangle \in \mathsf{Eval}^{\mathsf{n}}(Q, D)\},$$
$$\mathsf{Eval}^{\mathsf{n}}_{\mathsf{p}}(Q, D) = \{t \mid \langle t, \varphi\rangle \in \mathsf{Eval}^{\mathsf{n}}(Q, D) \text{ and } \varphi \neq \mathsf{false}\}.$$

Example 5. Consider the database D consisting of the two unary relations $R = \{\langle a\rangle\}$ and $S = \{\langle \bot_1\rangle\}$. Consider also the query $Q_5 = R - S$. The conditional database $\overline{D}$ consist of the two c-tables $\overline{R} = \{\langle a, \mathsf{true}\rangle\}$ and $\overline{S} = \{\langle \bot_1, \mathsf{true}\rangle\}$. Then,

$$\begin{aligned}\mathsf{Eval}^{\mathsf{n}}(Q_5, D) &= \mathsf{eval}\left(\mathsf{eval}\left(\overline{R}\right) \dot{-} \mathsf{eval}\left(\overline{S}\right)\right)\\ &= \mathsf{eval}\left(\overline{R} \dot{-} \overline{S}\right)\\ &= \mathsf{eval}\left(\{\langle a,\ \mathsf{true} \wedge \neg(\mathsf{true} \wedge (a = \bot_1))\rangle\}\right)\\ &= \{\langle a, \mathsf{unknown}\rangle\}.\end{aligned}$$

Thus, $\mathsf{Eval}^{\mathsf{n}}_{\mathsf{t}}(Q_5, D) = \emptyset$ and $\mathsf{Eval}^{\mathsf{n}}_{\mathsf{p}}(Q_5, D) = \{\langle a\rangle\}$. □

The following theorem states that the naive evaluation is equivalent to the evaluation algorithm of [18]. We point out that, in the following claim, D is a database (thus, possibly containing nulls), but without conditions. However, the naive evaluation first converts D into a conditional database $\overline{D}$ with all conditions being true, and then performs the evaluation $\mathsf{Eval}^{\mathsf{n}}()$ over $\overline{D}$, so that eventually tuples associated with true are certain answers with nulls.

Theorem 1. *The naive evaluation is equivalent to the evaluation algorithm of [18].*

Corollary 1. $\mathsf{Eval}^{\mathsf{n}}_{\mathsf{t}}(Q)$ *has correctness guarantees.*

Theorem 2. $\mathsf{Eval}^{\mathsf{n}}(Q, D)$ *can be computed in polynomial time in the size of* D*, for every query* Q *and database* D*.*

Obviously, $\mathsf{Eval}^{\mathsf{n}}_{\mathsf{t}}(Q, D)$ and $\mathsf{Eval}^{\mathsf{n}}_{\mathsf{p}}(Q, D)$ can be computed in polynomial time too (data complexity).

The naive evaluation presented above is somehow limited, since it does not use much the power of conditional tables, as shown in the following example.

Example 6. Consider the database D of Example 5 and the query $Q_6 = R - \sigma_{\$1=b}(S)$.

Clearly, $\mathsf{cert}(Q_6, D) = \{\langle a\rangle\}$, as the selection returns either $\{\langle b\rangle\}$ or $\emptyset$ in every possible world, and thus the result of the difference is $\{\langle a\rangle\}$ in every possible world. However, the naive evaluation is not able to realize this aspect and behaves as follows:

$$\begin{aligned}&\mathsf{Eval}^{\mathsf{n}}(Q_6, D) =\\ &\quad= \mathsf{eval}\left(\mathsf{eval}\left(\overline{R}\right) \dot{-} \mathsf{eval}\left(\dot{\sigma}_{\$1=b}\left(\mathsf{eval}\left(\overline{S}\right)\right)\right)\right)\\ &\quad= \mathsf{eval}\left(\{\langle a, \mathsf{true}\rangle\} \dot{-} \mathsf{eval}\left(\dot{\sigma}_{\$1=b}\left(\{\langle \bot_1, \mathsf{true}\rangle\}\right)\right)\right)\\ &\quad= \mathsf{eval}\left(\{\langle a, \mathsf{true}\rangle\} \dot{-} \mathsf{eval}\left(\{\langle \bot_1, \mathsf{true} \wedge (\bot_1 = b)\rangle\}\right)\right)\\ &\quad= \mathsf{eval}\left(\{\langle a, \mathsf{true}\rangle\} \dot{-} \{\langle \bot_1, \mathsf{unknown}\rangle\}\right)\\ &\quad= \mathsf{eval}\left(\{\langle a, \mathsf{true} \wedge \neg(\mathsf{unknown} \wedge (a = \bot_1))\rangle\}\right) = \{\langle a, \mathsf{unknown}\rangle\}.\end{aligned}$$

Thus, $\mathsf{Eval}^{\mathsf{n}}_{\mathsf{t}}(Q_6, D) = \emptyset$. In this case, the crucial point is that $\mathsf{eval}\left(\dot{\sigma}_{\$1=b}\left(\mathsf{eval}\left(\overline{S}\right)\right)\right) = \{\langle \bot_1, \mathsf{unknown}\rangle\}$, and the fact that $\bot_1$ can only be equal to b gets lost. □

The previous example suggests that equalities might be exploited to refine the evaluation of conditions, in order to provide more accurate information, as illustrated in the following example.

Example 7. Consider again the database D of Example 5 and the query Q_6 of Example 6. By "propagating" equalities into conditions and tuples after each relational algebra operator is conditionally evaluated, we get:

$$\begin{aligned}
&\{\langle a, \mathsf{true}\rangle\} \dot{-} \dot{\sigma}_{\$1=b}(\{\langle \bot_1, \mathsf{true}\rangle\}) \\
&= \{\langle a, \mathsf{true}\rangle\} \dot{-} \{\langle \bot_1, \mathsf{true} \wedge (\bot_1 = b)\rangle\} \\
&= \{\langle a, \mathsf{true}\rangle\} \dot{-} \{\langle b, \mathsf{unknown}\rangle\} \\
&= \{\langle a, \mathsf{true} \wedge \neg(\mathsf{unknown} \wedge (a = b))\rangle\} \\
&= \{\langle a, \mathsf{true}\rangle\}.
\end{aligned}$$

Thus, we conclude that $\langle a\rangle$ is a certain answer with nulls. Recall that $\mathsf{cert}(Q_6, D) = \{\langle a\rangle\}$ and $\mathsf{Eval}^{\mathsf{n}}_{\mathsf{t}}(Q_6, D) = \emptyset$. □

In the previous example, it is interesting to notice that the selection returns the c-tuple $\langle b, \mathsf{unknown}\rangle$, meaning that the only tuple that can be returned (in some cases) is $\langle b\rangle$, and thus providing more accurate information than the naive evaluation.

4 Conclusion

Certain answers are a principled manner to answer queries on incomplete databases. Since their computation is a coNP-hard problem, recent research has focused on polynomial time algorithms providing under-approximations. Leveraging conditional tables, we have shown how new approximation algorithms can be developed.

We are currently working on more refined evaluation strategies which better exploit the power of conditional tables and the conditional evaluation, with the aim of computing strictly more certain answers with nulls, while retaining polynomial time data complexity. For instance, conditions might be rewritten into a more suitable form (e.g., conjunctive normal form or disjunctive normal form [12, 13]) so as to allow better analyses.

References

1. Serge Abiteboul and Gösta Grahne. Update semantics for incomplete databases. In *Proc. Very Large Data Bases (VLDB) Conference*, pages 1–12, 1985.
2. Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78(1):158–187, 1991.
3. Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.
4. Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 68–79, 1999.
5. Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

6. Meghyn Bienvenu and Magdalena Ortiz. Ontology-mediated query answering with data-tractable description logics. In *Reasoning Web*, pages 218–307, 2015.
7. Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Exploiting equality generating dependencies in checking chase termination. *PVLDB*, 9(5):396–407, 2016.
8. Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, 14:57–83, 2012.
9. Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. On reconciling data exchange, data integration, and peer data management. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 133–142, 2007.
10. Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.
11. Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The LLUNATIC data-cleaning framework. *Proceedings of the VLDB Endowment*, 6(9):625–636, 2013.
12. Georg Gottlob and Enrico Malizia. Achieving new upper bounds for the hypergraph duality problem through logic. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 43:1–43:10, 2014.
13. Georg Gottlob and Enrico Malizia. Achieving new upper bounds for the hypergraph duality problem through logic. *SIAM Journal on Computing*, 2017.
14. Gösta Grahne. Dependency satisfaction in databases with incomplete information. In *Proc. Very Large Data Bases (VLDB) Conference*, pages 37–45, 1984.
15. Gösta Grahne. *The Problem of Incomplete Information in Relational Databases*, volume 554 of *Lecture Notes in Computer Science*. Springer, 1991.
16. Sergio Greco, Cristian Molinaro, and Francesca Spezzano. *Incomplete Data and Data Dependencies in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.
17. Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Checking chase termination: Cyclicity analysis and rewriting techniques. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):621–635, 2015.
18. Paolo Guagliardo and Leonid Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 211–223, 2016.
19. Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.
20. Paraschos Koutris and Jef Wijsen. The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 17–29, 2015.
21. Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.
22. Leonid Libkin. Certain answers as objects and knowledge. In *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 328–337, 2014.
23. Leonid Libkin. Incomplete data: what went wrong, and how to fix it. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 1–13, 2014.
24. Leonid Libkin. Sql's three-valued logic and certain answers. In *Proc. International Conference on Database Theory (ICDT)*, pages 94–109, 2015.
25. Leonid Libkin. SQL's three-valued logic and certain answers. *ACM Transactions Database Systems*, 41(1):1, 2016.
26. Witold Lipski. On relational algebra with marked nulls. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 201–203, 1984.