SQL's Three-Valued Logic and Certain Answers

Leonid Libkin

School of Informatics, University of Edinburgh

Abstract

SQL uses three-valued logic for evaluating queries on databases with nulls. The standard theoretical approach to evaluating queries on incomplete databases is to compute certain answers. While these two cannot coincide, due to a significant complexity mismatch, we can still ask whether the two schemes are related in any way. For instance, does SQL always produce answers we can be certain about?

This is not so: SQL's and certain answers semantics could be totally unrelated. We show, however, that a slight modification of the three-valued semantics for relational calculus queries can provide the required certainty guarantees. The key point of the new scheme is to fully utilize the three-valued semantics, and classify answers not into certain or non-certain, as was done before, but rather into certainly true, certainly false, or unknown. This yields relatively small changes to the evaluation procedure, which we consider at the level of both declarative (relational calculus) and procedural (relational algebra) queries. We also introduce a new notion of certain answers with nulls, which properly accounts for queries returning tuples containing null values.

1998 ACM Subject Classification H.2.4 Query Processing

Keywords and phrases Null values, incomplete information, query evaluation, three-valued logic, certain answers

Digital Object Identifier 10.4230/LIPIcs.ICDT.2015.1

1 Introduction

SQL's query evaluation engine uses three-valued logic when it comes to handling incomplete information: comparisons involving null values have the truth value *unknown* [7]. This results in a number of well known paradoxes. Consider, for instance, two relations R and S with a single numerical attribute A, and assume that S contains a single row with a null value in it. Then

```
select S.A from S where S.A <= 0 or S.A > 0
```
(1)

returns nothing despite the condition in the `where` clause being a tautology. This is because both `null <= 0` and `null > 0` evaluate to *unknown* and so does their disjunction. Worse yet, for the same reason, the query computing $R - S$:

```
select R.A from R where R.A not in (select S.A from S)
```
(2)

returns nothing if S contains a single null, no matter what R is, telling us that might well have $|R| > |S|$ and $R - S = \emptyset$ at the same time.

However unintuitive these answers are (which led to very severe criticism of the design of null-related features of SQL [6, 7]) they at least seem not to give us any false positives. To understand what it means, we appeal to the standard theoretical notion of query answering in the presence of incompleteness, *certain answers* [1, 12]. Each incomplete database D has an associated *semantics* $[\![D]\!]$. We can think of $[\![D]\!]$ as the set of possible complete databases that D can represent, i.e., all databases obtained by substituting values for nulls. Then

18th International Conference on Database Theory (ICDT'15).
Editors: Marcelo Arenas and Martin Ugarte; pp. 1–16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

certain answers contain tuples that will be in the answer to Q over all possible complete databases represented by D:

$$\mathsf{certain}(Q, D) \;=\; \bigcap\{Q(D') \mid D' \in [\![D]\!]\} \tag{3}$$

How does SQL evaluation of queries relate to certain answers? There is a simple argument that they cannot coincide for relational calculus queries: SQL's evaluation is tractable (*very* tractable, in fact, of AC^0 data complexity), but data complexity of certain answers is intractable: at least coNP-complete for commonly considered semantics [2]. Examples (1) and (2) seem to suggest that we at least get a subset of certain answers, but this is not the case: false positives are possible. Consider the query:

```
select R.A from R
where R.A not in (select R1.A from R R1                    (4)
                  where R1.A not in (select * from S))
```

expressing $R - (R - S)$ and a database $R = \{1\}$ and $S = \{\bot\}$. SQL's evaluation results in $\{1\}$. At the same time the certain answer is empty: if $\bot$ is interpreted as any value other than 1, the query produces $\emptyset$.

Can we remedy this? Clearly we cannot modify SQL's evaluation rules to generate certain answers due to the complexity mismatch. So the best we can hope for is a *reasonable approximation without false positives*. The idea itself is not new: in fact for the first time it was expressed in [22], even before complexity bounds for certain answers were known. Despite this, we do not yet have such approximation schemes for SQL query evaluation. Providing them is our goal here. Specifically, we want to achieve the following:

- find query answers fast, without a significant modification of the existing evaluation techniques, and at the same time
- guarantee that no false positives occur, i.e., every returned tuple is a certain answer.

We achieve this by providing a small modification to the three-valued logic approach of SQL that restores correctness guarantees: query evaluation no longer produces false positives, and all returned results are guaranteed to be certain answers.

To understand the idea of the modification, notice that SQL's query evaluation actually *mixes* three- and two-valued logic. Three-valued logic is used to evaluate conditions, but then query results return only those tuples for which conditions evaluate to *true*, effectively collapsing *unknown* and *false*. This works fine for positive queries, but once negation, especially negation in subqueries (e.g., `not in` or `not exists`) enters the picture, we have a problem, as it flips truth values. Now *true* flips to *false*, but both *unknown* and *false* (which were collapsed to one value when a subquery was evaluated) flip to *true*! This is how unintended tuples end up in the answer.

So to get correctness guarantees, we just need to be faithful to the three-valued approach. This means that there will be three possible outcomes for each candidate answer tuple: it can be either

- certainly in the answer (truth value *true*); or
- certainly *not* in the answer (truth value *false*); or
- possibly in the answer, or possibly not (truth value *unknown*).

The second modification that we need is using *marked*, or *naïve* nulls [1, 12] in tables. Such nulls can appear multiple times in tables, and they are often required by

applications such as data integration and exchange [3, 13]. In fact they have already been implemented in connection with such applications [11, 19]. Generally, SQL's nulls can be modeled with naïve nulls, simply by forbidding repetition. The reason we need marked nulls is twofold. Firstly, we want to produce more general results. Secondly, we need to overcome an additional (and quite unreasonable) deficiency of SQL's handling of nulls: even comparing whether a null value equals *itself* produces truth value *unknown*. Indeed, consider a table `T(A,B)` with a single tuple `(1,null)` and a query `select T1.A from T T1, T T2 where T1.A=T2.A and T1.B=T2.B`, i.e., $\pi_A(T \cap T)$. Instead of the expected 1, it gives the empty result, as comparing a value with itself does not evaluate to true.

We remark that using the multi-valued approach has proved very useful in two closely related areas: model-checking [4, 10], and knowledge representation [14, 18]. In fact the procedure of [14] that uses three-valued reasoning with knowledge bases is similar in spirit with the modification of SQL query evaluation that we propose (although the technical details of our procedure are quite different from [14]), and its modifications to achieve tractable reasoning [18] relied on database query evaluation techniques. In the database field the three-valued approach has, by and large, belonged to the practice rather than the theory.

Organization In Section 2 we present basic definitions. Section 3 describes the evaluation procedure for relational calculus and SQL's three-valued approach in the presence of nulls. Section 4 presents the modified evaluation procedure and states its correctness. In Section 5 we prove a generalization of that result, relying on a new notion of certain answers with nulls. This generalization properly accounts for all three possible outcomes of query evaluation (certainly true, certainly false, unknown). In Section 6 we look at certainty guarantees for relational algebra queries. Concluding remarks are in Section 7. Due to space limitations, only proof sketches are presented here; complete proofs are available in the full version.

2 Preliminaries

Incomplete databases We begin with some standard definitions [1, 12]. Incomplete databases are populated by *constants* and *nulls*. The sets of constants and nulls are countably infinite sets denoted by Const and Null respectively. Nulls are denoted by $\bot$, sometimes with sub- or superscripts.

A relational schema (vocabulary) is a set of relation names with associated arities. An incomplete relational instance D assigns to each k-ary relation symbol S from the vocabulary a k-ary relation S^D over $\mathsf{Const} \cup \mathsf{Null}$, i.e., a finite subset of $(\mathsf{Const} \cup \mathsf{Null})^k$. When the instance is clear from the context we shall write S, rather than S^D, for the relation itself as well.

The sets of constants and nulls that occur in D are denoted by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$. If $\mathsf{Null}(D)$ is empty, we refer to D as *complete*. That is, complete databases are those without nulls. The *active domain* of D is $\mathrm{adom}(D) = \mathsf{Const}(D) \cup \mathsf{Null}(D)$.

Homomorphisms, valuations, and semantics Given two relational structures D and D', a homomorphism $h : D \to D'$ is a map from the active domain of D to the active domain of D' such that:

1. for every relation symbol S, if a tuple $\bar{u}$ is in relation S in D, then the tuple $h(\bar{u})$ is in the relation S in D'; and
2. $h(c) = c$ for every $c \in \mathsf{Const}(D)$.

By $h(D)$ we denote the image of D, i.e., the set of all tuples $S(h(\bar{u}))$ where $S(\bar{u})$ is in D. If $h : D \to D'$ is a homomorphism, then $h(D)$ is a subinstance of D'.

A homomorphism $h : D \to D'$ is called a *valuation* if $h(x)$ is a constant for every $x \in \mathrm{adom}(D)$; in other words, it provides a valuation of nulls as constant values. If h is a valuation, then $h(D)$ is complete. We now define the semantics of incomplete databases by means of valuations:

$$[\![D]\!] \;=\; \{h(D) \mid h \text{ is a valuation}\}.$$

This is often referred to as the closed-world assumption, or CWA semantics of incompleteness [12, 21]. Another common semantics uses the open-world assumption, or OWA, and allows adding complete tuples to $h(D)$. In the study of incompleteness, the closed-world semantics is a bit more common [1, 2, 12] since it is better behaved. We shall offer some comments on the OWA semantics in Section 5.2.

Query languages As our basic query languages we consider relational calculus and its fragments. Relational calculus has exactly the power of *first-order logic*, or FO. Its formulae are built from relational atoms $R(\bar{x})$, equality atoms $x = y$, by closing them under conjunction $\wedge$, disjunction $\vee$, negation $\neg$, existential $\exists$ and universal $\forall$ quantifiers. If $\bar{x}$ is the list of free variables of a formula φ, we write $\varphi(\bar{x})$ to indicate this. We write $|\bar{x}|$ for the length of $\bar{x}$.

Conjunctive queries (CQs, also known as select-project-join queries) are defined as queries expressed in the $\exists, \wedge$-fragment of FO. The class UCQ of *unions of conjunctive queries* is the class of formulae of the form $\varphi_1 \vee \ldots \vee \varphi_m$, where each φ_i is a conjunctive query. In terms of its expressive power, this is the existential-positive fragment of FO, i.e., the $\exists, \vee, \wedge$-fragment.

We shall use relational algebra, the procedural language equivalent to FO, that has operations of selection σ, projection π, cartesian product $\times$, union $\cup$, and difference $-$. We use the unnamed perspective of relational algebra which does not require the renaming operator [1] (more on this in Section 6, where we shall add explicit intersection to relational algebra). The fragment without the difference operator is referred to as positive relational algebra; it has the same expressiveness as existential positive formulae (and thus unions of conjunctive queries).

3 Evaluation procedures for FO queries

We shall look at different query evaluation procedures. Each such procedure Eval will take a query (an FO formula) $\varphi(\bar{x})$, a database D, and an assignment ν of values to the free variables $\bar{x}$. The output $\mathsf{Eval}(\varphi, D, \nu)$ is a *truth value*. For the standard Boolean logic, the domain of truth values is $\{0, 1\}$, with 0 meaning *false* and 1 meaning *true*. For three-valued logic, the domain is $\{0, \frac{1}{2}, 1\}$, with $\frac{1}{2}$ interpreted as *unknown*.

An assignment ν maps each free variable to an element of $\mathrm{adom}(D)$. Note that such an element could be a constant or a null; assignments thus are *not* valuations. We write $\nu[a/x]$ for the assignment that changes ν by mapping x to a. Also, given a tuple $\bar{x} = (x_1, \ldots, x_n)$ of free variables, and a tuple $\bar{a} = (a_1, \ldots, a_n)$, we write simply $\mathsf{Eval}(\varphi, D, \bar{a})$ if the assignment ν is such that $\nu(x_i) = a_i$ for all $i \leq n$.

Given an evaluation procedure Eval, the outcome of query evaluation for $\varphi(\bar{x})$ with $|\bar{x}| = k$ is

$$\mathsf{Eval}(\varphi, D) \;=\; \{\bar{a} \in \mathrm{adom}(D)^k \mid \mathsf{Eval}(\varphi, D, \bar{a}) = 1\}$$

For all of the evaluation procedures that we use (except two in Subsection 5.2), the

evaluation of the Boolean connectives and quantifiers is completely standard:

$$\begin{array}{rcl} \mathsf{Eval}(\varphi \vee \psi, D, \nu) & = & \max(\mathsf{Eval}(\varphi, D, \nu), \mathsf{Eval}(\psi, D, \nu)) \\ \mathsf{Eval}(\varphi \wedge \psi, D, \nu) & = & \min(\mathsf{Eval}(\varphi, D, \nu), \mathsf{Eval}(\psi, D, \nu)) \\ \mathsf{Eval}(\neg\varphi, D, \nu) & = & 1 - \mathsf{Eval}(\varphi, D, \nu) \\ \mathsf{Eval}(\exists x\varphi, D, \nu) & = & \max\{\mathsf{Eval}(\varphi, D, \nu[a/x]) \mid a \in \mathrm{adom}(D)\} \\ \mathsf{Eval}(\forall x\varphi, D, \nu) & = & \min\{\mathsf{Eval}(\varphi, D, \nu[a/x]) \mid a \in \mathrm{adom}(D)\} \end{array} \tag{5}$$

Thus, from now we only explain the valuation of *atomic* formulae $R(\bar{x})$ and equalities $x = y$. The classical FO evaluation gives us the procedure $\mathsf{Eval}_{\mathsf{FO}}$ with the range $\{0, 1\}$ defined by (5) and:

$$\mathsf{Eval}_{\mathsf{FO}}(R(\bar{x}), D, \nu) \;=\; \begin{cases} 1 & \text{if } \nu(\bar{x}) \in R^D \\ 0 & \text{if } \nu(\bar{x}) \notin R^D \end{cases}$$

$$\mathsf{Eval}_{\mathsf{FO}}(x = y, D, \nu) \;=\; \begin{cases} 1 & \text{if } \nu(x) = \nu(y) \\ 0 & \text{if } \nu(x) \neq \nu(y) \end{cases}$$

SQL's evaluation has $\{0, \frac{1}{2}, 1\}$ as the range of values. Again it uses rules (5), and the rule for $\mathsf{Eval}_{\mathsf{SQL}}(R(\bar{x}), D, \nu)$ is exactly the same as for $\mathsf{Eval}_{\mathsf{FO}}$, but for equality atoms the rule differs:

$$\mathsf{Eval}_{\mathsf{SQL}}(x = y, D, \nu) \;=\; \begin{cases} 1 & \text{if } \nu(x) = \nu(y) \text{ and } \nu(x), \nu(y) \in \mathsf{Const} \\ 0 & \text{if } \nu(x) \neq \nu(y) \text{ and } \nu(x), \nu(y) \in \mathsf{Const} \\ \frac{1}{2} & \text{if } \nu(x) \in \mathsf{Null} \text{ or } \nu(y) \in \mathsf{Null} \end{cases}$$

Indeed, SQL's approach is to declare every comparison as *unknown* if a null is involved. Note that over complete databases, $\mathsf{Eval}_{\mathsf{FO}}$ and $\mathsf{Eval}_{\mathsf{SQL}}$ coincide. Also, over incomplete databases, $\mathsf{Eval}_{\mathsf{FO}}$ is usually referred to as naïve evaluation [1, 12].

How do these relate to certain answers? We now examine FO and SQL evaluation. But first note that the definition (3) ensures that only tuples of constants are present in certain answers. There is no such restriction on the standard evaluation procedures. So to do a fair comparison we only compare sets of constant tuples returned by evaluation procedures (this will be relaxed later in the paper).

▶ **Definition 1.** *Given a class $\mathcal{Q}$ of queries, an evaluation procedure* Eval *has* certainty guarantees *for $\mathcal{Q}$ if for every query $\varphi(\bar{x}) \in \mathcal{Q}$, every database D, and every tuple $\bar{a}$ of constants with $|\bar{a}| = |\bar{x}|$, we have*

$$\bar{a} \in \mathsf{Eval}(\varphi, D) \;\Rightarrow\; \bar{a} \in \mathsf{certain}(\varphi, D).$$

In other words,

$$\mathsf{Eval}(\varphi, D) \cap \mathsf{Const}^{|\bar{x}|} \;\subseteq\; \mathsf{certain}(\varphi, D).$$

Certain answers and $\mathsf{Eval}_{\mathsf{FO}}$ The first observation is immediate:

$$\mathsf{certain}(\varphi, D) \;\subseteq\; \mathsf{Eval}_{\mathsf{FO}}(\varphi, D).$$

The converse in general is not true, we can have $\mathsf{Eval}_{\mathsf{FO}}(\varphi(\bar{x}), D) \cap \mathsf{Const}^{|\bar{x}|} \not\subseteq \mathsf{certain}(\varphi, D)$. Consider for instance $\varphi(x) = R(x) \wedge \neg S(x)$ expressing the difference of R and S. Let D contain $R^D = \{1\}$ and $S^D = \{\bot\}$; then $\mathsf{Eval}(\varphi, D) = \{1\}$ while $\mathsf{certain}(\varphi, D) = \emptyset$.

However, sometimes certainty guarantees can be established. It has long been known [12] that we get them by excluding universal quantification and negation from first-order logic: $\mathsf{Eval}_{\mathsf{FO}}$ has certainty guarantees for the class UCQ. This was recently extended in [8] which showed that the same is true for queries from a rather significant expansion of the class UCQ, by adding universal quantification and a limited form of implication. More precisely, we look at the class $\mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$ defined as follows:

- atomic formulae $R(\bar{x})$ and $x = y$ are in $\mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$;
- if $\varphi, \psi \in \mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$ then so are $\varphi \vee \psi$ and $\varphi \wedge \psi$;
- if $\varphi \in \mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$ then so are $\exists x \varphi$ and $\forall x \varphi$;
- if $\varphi(\bar{x}, \bar{y})$ is in $\mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$, then so is $\forall \bar{x}\ (R(\bar{x}) \to \varphi(\bar{x}, \bar{y}))$, where R is a relation symbol in the schema, and $\bar{x}$ does not have a repetition of variables.

Then $\mathsf{Eval}_{\mathsf{FO}}$ has certainty guarantees for $\mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$ queries [8]. From the point of view of relational algebra, the class $\mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$ corresponds to operations $\sigma, \pi, \cup, \times$ and the division operation $Q \div Q'$, where Q' is written in the $\pi, \cup, \times$-fragment of relational algebra, see [16].

Certain answers and $\mathsf{Eval}_{\mathsf{SQL}}$ How does SQL change things? Actually, it changes them for the *worse*: now there is no connection between $\mathsf{Eval}_{\mathsf{SQL}}(\varphi, D)$ and $\mathsf{certain}(\varphi, D)$ whatsoever. Indeed, we saw that for the query $\varphi(x) = R(x) \wedge \neg(R(x) \wedge \neg S(x))$ and database D with $R^D = \{1\}$ and $S^D = \{\bot\}$, the certain answer is empty while $\mathsf{Eval}_{\mathsf{SQL}}(\varphi, D) = \{1\}$, and for $\psi(x) = R(x) \wedge (S(x) \vee \neg S(x))$, the certain answer is $\{1\}$, while $\mathsf{Eval}_{\mathsf{SQL}}(\psi, D) = \emptyset$.

In a restricted case we provide correctness guarantees:

▶ **Proposition 2.** $\mathsf{Eval}_{\mathsf{SQL}}$ *has certainty guarantees for unions of conjunctive queries.*

Proof sketch. This follows from the fact for unions of conjunctive queries, $\mathsf{Eval}_{\mathsf{SQL}}(\varphi, D, \nu) = 1$ implies $\mathsf{Eval}_{\mathsf{FO}}(\varphi, D, \nu) = 1$ (shown by induction), and known results for FO evaluation for unions of conjunctive queries [12]. □

4 Evaluation procedures with certainty guarantees

We now introduce an evaluation procedure that comes with certainty guarantees for *all* relational calculus queries. For that, we have to explain what is wrong with FO and SQL evaluation procedures shown above, particularly for evaluation of atomic formulae.

Atomic relational formulae $R(\bar{x})$ For both SQL and FO, one simply checks, for a given assignment ν, whether $\nu(\bar{x})$ belongs to R. However, returning 0 if $\nu(\bar{x}) \notin R$ is too strong if we view 0 as saying that the tuple certainly *cannot* belong to R.

Indeed, consider $R = \{(\bot_1, 1), (2, \bot_2)\}$ and let ν be the identity (recall that the range of ν is the whole active domain). Consider a tuple $\bar{x} = (\bot_1, \bot_2)$. It is not in R, but can it be in R under some valuation h? Of course it can: if $h(\bot_1) = 2$ and $h(\bot_2) = 1$, then $h(\bar{x}) = (2, 1)$ and $h(R) = \{(2, 1)\}$, i.e., $h(\bar{x}) \in h(R)$. On the other hand, if $h'(\bot_1) = 1$ and $h'(\bot_2) = 2$, then $h'(\bar{x}) = (1, 2)$ and $h'(R) = \{(1, 1), (2, 2)\}$, so $h'(\bar{x}) \notin h'(R)$. Thus, the correct value for evaluating the membership of $\bar{x}$ in R seems to be $\frac{1}{2}$, not 0. Value 0 should be reserved for cases when no valuation h makes $h(\bar{x}) \in h(R)$ possible.

The $\mathsf{Eval}_{\mathsf{FO}}$ and $\mathsf{Eval}_{\mathsf{SQL}}$ procedures return 0 too eagerly, and this becomes a problem when negation is applied to a formula, as 0 becomes a 1, and suddenly we have a false positive answer that in fact is not certain at all. If the value is kept at $\frac{1}{2}$, applying negation still results in $1 - \frac{1}{2} = \frac{1}{2}$, and thus no false 'certain answers' appear.

Equality formulae $x = y$ FO evaluation results in 0 if $\nu(x)$ and $\nu(y)$ are different nulls, but they could still be mapped to the same constant, so the right value should be $\frac{1}{2}$, not 0. On the other hand, SQL evaluation produces $\frac{1}{2}$ if one of $\nu(x)$ or $\nu(y)$ is a null. But if we know $\nu(x) = \nu(y)$, then for every valuation h we will have $h(\nu(x)) = h(\nu(y))$, so the evaluation procedure must return 1 and not $\frac{1}{2}$ in this case, or else it will miss some certain answers.

Now with this in mind, we introduce a proper *3-valued evaluation procedure* $\mathsf{Eval}_{3\mathsf{v}}$. For this, we need one additional concept. Given two tuples $\bar{t}_1$ and $\bar{t}_2$ of the same length over $\mathsf{Const} \cup \mathsf{Null}$, we say that they *unify* if there is a homomorphism h such that $h(\bar{t}_1) = h(\bar{t}_2)$. We then write $\bar{t}_1 \Uparrow \bar{t}_2$.

It is easy to see that we can define $\bar{t}_1 \Uparrow \bar{t}_2$ by asking for a valuation h so that $h(\bar{t}_1) = h(\bar{t}_2)$. By classical results on unification, it is known that $\bar{t}_1 \Uparrow \bar{t}_2$ can be tested in linear time [20].

Now the evaluation procedure is as follows. It uses rules (5) and the following rules for atomic formulae:

$$\mathsf{Eval}_{3\mathsf{v}}(R(\bar{x}), D, \nu) = \begin{cases} 1 & \text{if } \nu(\bar{x}) \in R^D \\ 0 & \text{if there is no } \bar{t} \in R^D \text{ such that } \nu(\bar{x}) \Uparrow \bar{t} \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

$$\mathsf{Eval}_{3\mathsf{v}}(x = y, D, \nu) = \begin{cases} 1 & \text{if } \nu(x) = \nu(y) \\ 0 & \text{if } \nu(x), \nu(y) \in \mathsf{Const} \text{ and } \nu(x) \neq \nu(y) \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

Coming back to the example in the beginning of the section, if we have a database D with $R^D = \{(\bot_1, 1), (2, \bot_2)\}$ and $\nu : (x, y) \mapsto (\bot_1, \bot_2)$, then $\mathsf{Eval}_{3\mathsf{v}}(R(x, y), D, \nu) = \frac{1}{2}$. Indeed, even though $(\bot_1, \bot_2)$ is not in R^D, there are valuations h so that $h(\bot_1, \bot_2) \in h(R^D)$. On the other hand, no valuation h makes $(1, 2) \in h(R^D)$ possible, so for $\nu' : (x, y) \mapsto (1, 2)$ we have $\mathsf{Eval}_{3\mathsf{v}}(R(x, y), D, \nu') = 0$.

These modifications turn out to be sufficient to ensure certainty guarantees for all relational calculus queries.

▶ **Theorem 3.** $\mathsf{Eval}_{3\mathsf{v}}$ *has certainty guarantees for all* FO *queries.*

As an example, consider again query (4), or $\varphi(x) = R(x) \wedge \neg(R(x) \wedge \neg S(x))$ over D with $R^D = \{1\}$ and $S^D = \{\bot\}$. It produced a false positive since $\mathsf{Eval}_{\mathsf{SQL}}(\varphi, D) = \{1\}$ but the certain answer is empty. But now we have $\mathsf{Eval}_{3\mathsf{v}}(\varphi, D) = \emptyset$. Indeed, we had $\mathsf{Eval}_{\mathsf{SQL}}(R(x) \wedge \neg S(x), D, 1) = 0$, and thus $\mathsf{Eval}_{\mathsf{SQL}}(\varphi, D, 1) = 1$, but now $\mathsf{Eval}_{3\mathsf{v}}(R(x) \wedge \neg S(x), D, 1) = \frac{1}{2}$ and hence $\mathsf{Eval}_{3\mathsf{v}}(\varphi, D, 1) = \frac{1}{2}$.

As another remark, note that the result of $\mathsf{Eval}_{3\mathsf{v}}$ need not be contained in the result of $\mathsf{Eval}_{\mathsf{SQL}}$, i.e., $\mathsf{Eval}_{3\mathsf{v}}$ can produce results that SQL evaluation misses. For instance, given a database D with $R^D = \{(\bot, \bot)\}$ and a query $\psi = \exists x, y\, R(x, y) \wedge x = y$, one can easily check that $\mathsf{Eval}_{3\mathsf{v}}(\psi, D, \nu) = 1$ (for the only possible valuation over a singleton active domain), while $\mathsf{Eval}_{\mathsf{SQL}}(\psi, D, \nu) = \frac{1}{2}$.

Theorem 3 will be a consequence of a more general result (Theorem 6), that does not restrict us to constant tuples. But for this we first need to define certain answers with nulls.

5 Certain answers with nulls

While the definition of certain answers (3) has been with us for 30+ years [17], recently it has been questioned [15, 16]. One of the problems with this definition is that it only returns tuples containing constants. Consider a database D with a relation $R^D = \{(1,2),(3,\bot)\}$ and a query $\psi(x,y) = R(x,y)$. Then $\mathsf{certain}(\psi, D) = \{(1,2)\}$ but intuitively we should return the entire relation R^D since we are certain its tuples are in the answer. The reason we are certain about it is that for every valuation h, the tuple $(3, h(\bot))$ is in $h(D)$. We turn this reasoning into a definition.

▶ **Definition 4.** Given an incomplete database D and a k-ary query Q defined over complete databases, *certain answers with nulls* $\mathsf{certain}_\bot(Q, D)$ is defined as the set of all tuples $\bar{u} \in \mathrm{adom}(D)^k$ such that $h(\bar{u}) \in Q(h(D))$ for all valuations h.

For instance, if a query is given by an FO formula with k free variables, then

$$\mathsf{certain}_\bot(\varphi, D) \;=\; \{\bar{u} \in \mathrm{adom}(D)^k \;\mid\; \mathsf{Eval}_{\mathsf{FO}}(\varphi, h(D), h(\bar{u})) = 1 \text{ for every valuation } h\}.$$

Returning to the above example, we have $\mathsf{certain}_\bot(\psi, D) = \{(1,2),(3,\bot)\}$, so the tuple $(3,\bot)$ is no longer omitted.

We now summarize properties of certain answers with nulls. The usual certain answers can be obtained from certain answers with nulls by dropping tuples containing nulls, and certain answers with nulls are always contained in the result of the simple FO evaluation of formulae. Sometimes, but not always, they may coincide with the result of such an evaluation.

Formally, we have the following.

▶ **Proposition 5.** *The following hold:*

- $\mathsf{certain}(\varphi(\bar{x}), D) = \mathsf{certain}_\bot(\varphi, D) \cap \mathsf{Const}^{|\bar{x}|}$.
- $\mathsf{certain}_\bot(\varphi, D) \subseteq \mathsf{Eval}_{\mathsf{FO}}(\varphi, D)$ *for every* FO *query* φ.
- *If* $\varphi \in \mathcal{Q}^{\mathsf{cert}}_{\mathsf{FO}}$, *then* $\mathsf{certain}_\bot(\varphi, D) = \mathsf{Eval}_{\mathsf{FO}}(\varphi, D)$.
- *There exist* FO *queries* φ *so that* $\mathsf{certain}_\bot(\varphi, D) \neq \mathsf{Eval}_{\mathsf{FO}}(\varphi, D)$.

We can now state a more general description of the evaluation procedure $\mathsf{Eval}_{\mathsf{3v}}$: the output value 1 guarantees that a tuple belongs to certain answers with nulls for query φ, the output value 0 guarantees that it belongs to certain answers with nulls for the negation $\neg\varphi$, and output value $\frac{1}{2}$ comes with no guarantees.

▶ **Theorem 6.** *For every* FO *query* $\varphi(\bar{x})$ *and every database* D,

$$\mathsf{Eval}_{\mathsf{3v}}(\varphi, D) \subseteq \mathsf{certain}_\bot(\varphi, D).$$

Moreover, if $\bar{a} \in adom(D)^{|\bar{x}|}$ *and* $\mathsf{Eval}_{\mathsf{3v}}(\varphi, D, \bar{a}) = 0$, *then* $\bar{a} \in \mathsf{certain}_\bot(\neg\varphi, D)$.

Theorem 3 is now an immediate corollary: if $\bar{a}$ is a tuple of constants and $\mathsf{Eval}_{\mathsf{3v}}(\varphi, D, \bar{a}) = 1$, then by Theorem 6, $\bar{a} \in \mathsf{certain}_\bot(\varphi, D)$, and by Proposition 5, $\bar{a} \in \mathsf{certain}(\varphi, D)$.

Proof sketch. We first show an auxiliary result that $\bar{u} \in \mathsf{certain}_\bot(\varphi, D)$ if and only if $\mathsf{Eval}_{\mathsf{FO}}(\varphi, h(D), h(\bar{u})) = 1$ for every homomorphism h (rather than every valuation h). Then the theorem is a consequence of the following:

$$\mathsf{Eval}_{\mathsf{3v}}(\varphi, D, \nu) = 1 \;\Rightarrow\; \forall \text{ homomorphism } h:\; \mathsf{Eval}_{\mathsf{FO}}(\varphi, h(D), h(\nu(\bar{x}))) = 1 \qquad (*)$$

$$\mathsf{Eval}_{3\mathsf{v}}(\varphi, D, \nu) = 0 \;\Rightarrow\; \forall \text{ homomorphism } h:\; \mathsf{Eval}_{\mathsf{FO}}(\varphi, h(D), h(\nu(\bar{x}))) = 0 \qquad (**)$$

This is shown by induction on φ; we provide the proof for the case of atomic formulae (for which $\mathsf{Eval}_{3\mathsf{v}}$ differs from $\mathsf{Eval}_{\mathsf{FO}}$) here.

If $\varphi(\bar{x})$ is a relational atom $R(\bar{x})$, then:

(*) If $\mathsf{Eval}_{3\mathsf{v}}(\varphi, D, \nu) = 1$ then $\nu(\bar{x}) \in R^D$; in particular, $h(\nu(\bar{x})) \in h(R^D)$ for every homomorphism h, showing $h(\nu(\bar{x})) \in \mathsf{Eval}_{\mathsf{FO}}(\varphi, h(D))$.

(**) If $\mathsf{Eval}_{3\mathsf{v}}(\varphi, D, \nu) = 0$ then for each tuple $\bar{t} \in R^D$ we have that $\nu(\bar{x}) \Uparrow \bar{t}$ does not hold. Thus for each homomorphism h, and each tuple $\bar{t} \in R^D$, we have $h(\nu(\bar{x})) \neq h(\bar{t})$. This means that $h(\bar{x}) \notin h(R^D)$, and thus for each homomorphism h we have $\mathsf{Eval}_{\mathsf{FO}}(\varphi, h(D), h(\nu(\bar{x}))) = 0$.

If $\varphi(x, y)$ is an equational atom $x = y$, then:

(*) If $\mathsf{Eval}_{3\mathsf{v}}(x = y, D, \nu) = 1$ then $\nu(x) = \nu(y)$, and thus for for every homomorphism h, we have $h(\nu(x)) = h(\nu(y))$; in particular, $\mathsf{Eval}_{\mathsf{FO}}(x = y, h(D), \nu) = 1$.

(**) If $\mathsf{Eval}_{3\mathsf{v}}(x = y, D, \nu) = 0$, then both $\nu(x)$ and $\nu(y)$ are constants and $\nu(x) \neq \nu(y)$. Since they are constants, every homomorphism leaves them intact, and thus $\mathsf{Eval}_{\mathsf{FO}}(x = y, h(D), \nu) = 0$. □

Another corollary says that we can use $\mathsf{Eval}_{3\mathsf{v}}$ to find *overapproximations* of certain answers:

▶ **Corollary 7.** *For every* FO *query* $\varphi(\bar{x})$ *we have*

$$\mathsf{certain}_{\perp}(\varphi, D) \;\subseteq\; adom(D)^{|\bar{x}|} - \mathsf{Eval}_{3\mathsf{v}}(\neg\varphi, D).$$

As for the complexity of the procedure, one can easily show the following.

▶ **Proposition 8.** *For each relational vocabulary* σ *and* $\alpha \in \{0, \frac{1}{2}, 1\}$*, from every* FO *query* $\varphi(\bar{x})$ *one can compute* FO *queries* $\varphi^{\alpha}(\bar{x})$ *in the vocabulary that extends* σ *with a unary predicate* $\mathsf{const}(\cdot)$ *interpreted as the set of constants, such that, for every database* D*,*

$$\{\bar{a} \in adom(D)^{|\bar{x}|} \mid \mathsf{Eval}_{3\mathsf{v}}(\varphi, D, \bar{a}) = \alpha\} \;=\; \mathsf{Eval}_{\mathsf{FO}}(\varphi^{\alpha}, D).$$

Consequently, data complexity of computing $\mathsf{Eval}_{3\mathsf{v}}(\varphi, D)$ *is in* AC^0*.*

This gives us a complexity argument showing that there are cases when $\mathsf{Eval}_{3\mathsf{v}}$ fails to produce *all* certain answers. A concrete example of strict containment of $\mathsf{Eval}_{3\mathsf{v}}$ in $\mathsf{certain}_{\perp}$ will be shown below in Section 5.1.

5.1 CQs and UCQs with inequalities

A common extension of conjunctive queries and their unions is by adding inequalities [1]. This is a very mild form of negation; essentially, we only allow negation to be applied to equality atoms. Instead of writing them as $\neg(x = y)$, it is common to use $x \neq y$ in formulae, and refer to them as inequality atoms. Then the $\exists, \wedge$-closure of relational, equality and inequality atoms is referred to as CQs with inequalities, and the $\exists, \wedge, \vee$-closure as *UCQs with inequalities*. This class of queries is denoted by $\mathsf{UCQ}^{\neq}$.

We now present a particularly easy evaluation procedure that correctly accounts for $\mathsf{Eval}_{3\mathsf{v}}$ producing value 1 for UCQs with inequalities, and thus gives us correctness guarantees for

those queries. This procedure uses two-valued, rather than three-valued, logic and only one rule that separates it from $\mathsf{Eval}_{\mathsf{FO}}$. To understand it, note for an inequality atom $x \neq y$, FO evaluation returns true if x and y are assigned different values – even if they are different nulls. But actually the evaluation of conditions such as $\bot_1 \neq \bot_2$ must be false, since $\bot_1$ and $\bot_2$ can be mapped, by a valuation, to the same element. For $\mathsf{UCQ}^{\neq}$, there is no risk with assigning *false* rather than *unknown*, since negation will never be applied further on. This lets us define the evaluation procedure for $\mathsf{UCQ}^{\neq}$ by adding the following explicit rule for $\neq$ formulae to the $\mathsf{Eval}_{\mathsf{FO}}$ rules:

$$\mathsf{Eval}_{\mathsf{UCQ}^{\neq}}(x \neq y, D, \nu) \;=\; \begin{cases} 1 & \text{if } \nu(x), \nu(y) \in \mathsf{Const} \text{ and } \nu(x) \neq \nu(y) \\ 0 & \text{otherwise} \end{cases}$$

This evaluation is particularly easy to implement in SQL with the usual `is not null` conditions in the `where` clause. And it has the desired correctness guarantees.

▶ **Theorem 9.** *For every* $\mathsf{UCQ}^{\neq}$ *query* φ*, we have*

$$\mathsf{Eval}_{\mathsf{UCQ}^{\neq}}(\varphi, D) \;=\; \mathsf{Eval}_{\mathsf{3v}}(\varphi, D) \;\subseteq\; \mathsf{certain}_{\bot}(\varphi, D).$$

In particular, $\mathsf{Eval}_{\mathsf{UCQ}^{\neq}}$ *has certainty guarantees for* $\mathsf{UCQ}^{\neq}$ *queries.*

Proof sketch. We prove, by induction on the formulae, that $\mathsf{Eval}_{\mathsf{UCQ}^{\neq}}(\varphi, D, \nu) = 1$ iff $\mathsf{Eval}_{\mathsf{3v}}(\varphi, D, \nu) = 1$ for $\mathsf{UCQ}^{\neq}$ queries. □

One cannot capture $\mathsf{certain}_{\bot}(\varphi, D)$ precisely with the $\mathsf{UCQ}^{\neq}$ evaluation procedure. Indeed, consider the query $\psi = \exists x \exists y\ R(x, y) \wedge x \neq y$ and a database D with $R^D = \{(\bot, 1), (\bot, 2)\}$. One easily checks $\mathsf{certain}_{\bot}(\psi, D) = \mathsf{certain}(\psi, D) = \text{true}$ but at the same time $\mathsf{Eval}_{\mathsf{UCQ}^{\neq}}(\psi, D) = 0$. By Theorem 9, this also means that $\mathsf{Eval}_{\mathsf{3v}}(\psi, D)$ fails to capture $\mathsf{certain}_{\bot}(\varphi, D)$; this is the example promised at the end of the last section.

In fact there could be no polynomial-time evaluation procedure for finding certain answers for $\mathsf{UCQ}^{\neq}$ queries since they have coNP-complete data complexity, even without free variables. Indeed, suppose we have a graph $G = \langle V, E\rangle$ where the set of vertices is $\{a_1, \ldots, a_n\}$. Create a binary relation D_G with $\mathrm{adom}(D_G) = \{\bot_1, \ldots, \bot_n\}$ and pairs $(\bot_i, \bot_j)$ for every edge $(a_i, a_j) \in E$. Let $\varphi \in \mathsf{UCQ}^{\neq}$ be given by $\exists x\ D_G(x, x) \vee \exists x, y, z, u\ (x \neq y \wedge x \neq z \wedge x \neq u \wedge y \neq z \wedge y \neq u \wedge z \neq u)$. Then $\mathsf{certain}(\varphi, D_G)$ is true iff G is not 3-colorable.

5.2 Open world semantics

Another commonly used semantics of incompleteness is based on the *open-world assumption*, or owa [1, 12, 21]. Under this assumption, after applying a valuation h to a database, finitely many complete tuples can be added to it. That is,

$$[\![D]\!]_{\mathrm{owa}} \;=\; \{h(D) \cup D' \mid h \text{ is a valuation and } D' \text{ is complete}\}.$$

Certain answers under owa are defined as $\mathsf{certain}_{\mathrm{owa}}(Q, D) = \bigcap\{Q(D') \mid D' \in [\![D]\!]_{\mathrm{owa}}\}$.

The evaluation procedure $\mathsf{Eval}_{\mathsf{3v}}$ no longer has certainty guarantees under owa. To see this, consider D with relations $R^D = \{(1, 2)\}$ and $S^D = \{(\bot_1, 1), (2, \bot_2)\}$. Let $\varphi(x, y) = R(x, y) \wedge \neg S(x, y)$. Since the tuple $(1, 2)$ does not unify with either tuple in S^D, we have $(1, 2) \in \mathsf{Eval}_{\mathsf{3v}}(\varphi, D)$. However, under owa, it is not a certain answer: for instance, the database D' with $R^{D'} = \{(1, 2)\}$ and $S^{D'} = \{(1, 1), (2, 2), (1, 2)\}$ is in $[\![D]\!]_{\mathrm{owa}}$, and $\mathsf{Eval}_{\mathsf{FO}}(\varphi, D')$ is empty.

Thus, our question is whether the approach of Eval_{3v}, guaranteeing correctness for all FO queries under CWA, can be extended to OWA. Of course there is always a trivial positive answer: the evaluation procedure that always returns 0 vacuously has correctness guarantees. Since $[\![D]\!]_{\text{CWA}} \subseteq [\![D]\!]_{\text{OWA}}$, certain answers under OWA will be included in certain answers under CWA, so the question really is how much we eliminate from the latter so that the result is still meaningful, and provides certainty guarantees under OWA. Note also that finding certain answers under OWA is undecidable [2] (even for data complexity [9]) which ties our hands even more in terms of finding suitable approximations.

To understand the changes that need to be made under OWA, consider again relational atoms. For them, there is no way to assert with certainty that a tuple does not belong to a relation, since each relation can be expanded under OWA. Hence, the case when evaluation produces 0 must go.

Next, look at existential formulae. Again we cannot state with certainty that the result of evaluation of those is 0, as perhaps in some extension of the database there is a witness for the existential formula, so the lowest value for evaluating such a formula is $\frac{1}{2}$, not 0. Likewise, for universal formulae, one cannot state with certainty that the result of evaluation is 1, as it requires checking the universal conditions in all extensions of the database, which is an undecidable problem. Hence, the highest value in this case is $\frac{1}{2}$ and not 1.

This explains the three changes that we make for the evaluation procedure. The procedure $\mathsf{Eval}^{\text{OWA}}_{3v}$ has the range $\{0, \frac{1}{2}, 1\}$ and differs from Eval_{3v} in three rules:

$$\mathsf{Eval}^{\text{OWA}}_{3v}(R(\bar{x}), D, \nu) = \begin{cases} 1 & \text{if } \nu(\bar{x}) \in R^D \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

$$\mathsf{Eval}^{\text{OWA}}_{3v}(\exists x\varphi, D, \nu) = \max\left\{\tfrac{1}{2},\ \max\{\mathsf{Eval}^{\text{OWA}}_{3v}(\varphi, D, \nu[a/x]) \mid a \in \text{adom}(D)\}\right\}$$

$$\mathsf{Eval}^{\text{OWA}}_{3v}(\forall x\varphi, D, \nu) = \min\left\{\tfrac{1}{2},\ \min\{\mathsf{Eval}^{\text{OWA}}_{3v}(\varphi, D, \nu[a/x]) \mid a \in \text{adom}(D)\}\right\}$$

Note that this procedure is the only one that modifies rules (5). These modifications are sufficient for correctness under OWA.

▶ **Proposition 10.** *The evaluation algorithm* $\mathsf{Eval}^{\text{OWA}}_{3v}$ *has correctness guarantees under* OWA.

Returning to the example from the beginning of the subsection, note that the value of $\mathsf{Eval}^{\text{OWA}}_{3v}(S(x,y), D, (1,2))$ is $\frac{1}{2}$ (for Eval_{3v} it would have been 0), and thus the result $\mathsf{Eval}^{\text{OWA}}_{3v}(R(x,y) \wedge \neg S(x,y), D, (1,2))$ is $\frac{1}{2}$ as well; in particular, $(1,2) \notin \mathsf{Eval}^{\text{OWA}}_{3v}(\varphi, D)$ while we had $(1,2) \in \mathsf{Eval}_{3v}(\varphi, D)$.

A remark on equivalence of queries under Eval_{3v}. Under the usual FO semantics, called $\mathsf{Eval}_{\mathsf{FO}}$ here, we are used to a number of equivalences that are *not* necessarily true when Eval_{3v} is used instead. Consider, for instance, a formula $\varphi(x) = \exists y\big(R(x,y) \wedge (y = 1 \vee y \neq 1)\big)$. Of course we expect it to be equivalent to $\varphi'(x) = \exists y R(x,y)$. However, under the three-valued semantics these are not equivalent: if $R^D = \{(1, \bot)\}$ and $\nu : x \mapsto 1$, then $\mathsf{Eval}_{\mathsf{FO}}(\varphi, D, \nu) = \mathsf{Eval}_{\mathsf{FO}}(\varphi', D, \nu) = 1 = \mathsf{Eval}_{3v}(\varphi', D, \nu)$, but at the same time $\mathsf{Eval}_{3v}(\varphi, D, \nu) = \frac{1}{2}$. This point will be important for us in the next section, where we present an evaluation procedure of relational algebra for databases with nulls.

6 Evaluation procedure for relational algebra

Queries that get executed in a DBMS are procedural queries, in particular, in the relational case, they are written in relational algebra, or some of its extensions. We now present

an algorithm that provides an evaluation with correctness guarantees for relational algebra expressions. Even though from the point of view of expressiveness, relational algebra is equivalent to FO, the equivalence itself, established under the standard two-valued semantics, is not yet a guarantee that it will provide us with a desired evaluation procedure in the three-valued world.

To expand on this, note that by Proposition 8, for every FO query $\varphi(\bar{x})$ we have a relational algebra expression e_φ which has access to the extra predicate $\mathsf{const}(\cdot)$ so that e_φ faithfully implements $\mathsf{Eval}_{3\mathsf{v}}(\varphi, \cdot)$. So it seems that starting with a relational algebra query Q, we could find an equivalent FO query φ_Q and then consider e_{φ_Q} to evaluate Q.

Reasoning of this sort, however, mixes the equivalence of FO and relational algebra (that is true with respect to the usual two-valued FO evaluation) with the three-valued evaluation. Still, from the equivalence of $\mathsf{Eval}_{\mathsf{FO}}(\varphi_Q, \cdot)$ and Q one can easily derive $e_{\varphi_Q}(D) = \mathsf{Eval}_{3\mathsf{v}}(\varphi_Q, D) \subseteq \mathsf{certain}_\perp(Q, D)$, so we do in fact get correctness guarantees with this approach. Nonetheless, it not satisfactory for two reasons. First, the detour via translation into FO and back to algebra may produce unnecessarily complicated expressions. Second, this approach assumes a particular translation between relational algebra and FO (which of course is not unique), and the quality of the resulting query depends on that translation. For instance, we view expressions R and $\sigma_{A=1}(R) \cup \sigma_{A\neq 1}(R)$ as equivalent, but using the latter in e_{φ_Q} can miss some answers with certainty guarantees due to the presence of nulls.

The bottom line is that it is better to have a *direct* evaluation procedure for relational algebra that gives us correctness guarantees without going through both algebra-to-FO and FO-to-algebra translations.

In the two-valued world sound translations for relational algebra have been considered in the past [22]. Our goal is a bit different though as we have to provide specific correctness guarantees, and relate them to SQL's way of evaluating queries; in fact we shall produce approximations for sets of tuples on which $\mathsf{Eval}_{3\mathsf{v}}$ returns 1 and 0.

We now explain the procedure for correct evaluation of relational algebra queries. First, recall the operations of relational algebra. These are selection σ, projection π, cartesian product $\times$, union $\cup$, intersection $\cap$, and difference $-$. To avoid the clutter, and in particular to avoid renaming, we use the *unnamed* perspective for presenting relational algebra [1], that is, for each expression returning an m-attribute relation, we simply assume that the names of those attributes are $\sharp 1, \ldots, \sharp m$. As conditions θ in selections, we use positive Boolean combinations of equalities and inequalities between attribute values and constants. For instance, $(\sharp 1 \neq \sharp 2) \vee (\sharp 3 = 1)$ is a condition that can be used in selection. Note that such conditions are closed under negation, simply by propagating it all the way to (in)equalities, so we shall also refer sometimes to conditions $\neg\theta$, meaning the result of such a propagation. We refer to this standard relational algebra as RA.

We also consider an extension called $\mathsf{RA}_{\mathsf{null}}$. In this extension, conditions θ are positive Boolean combinations of

- equalities and inequalities between attributes, and
- conditions $\mathsf{const}(\sharp n)$ and $\mathsf{null}(\sharp n)$ stating that the value of attribute $\sharp n$ is a constant or a null, respectively.

Our goal is to provide a translation $\mathsf{RA} \to \mathsf{RA}_{\mathsf{null}}$ that associates with each query Q of RA a query Q^+ of $\mathsf{RA}_{\mathsf{null}}$ such that $Q^+(D) \subseteq \mathsf{certain}_\perp(Q, D)$.

As noticed already, due to coNP-data complexity of $\mathsf{certain}_\perp(Q, D)$, we cannot hope for equality, so this correctness guarantee is the best we can count on.

We shall actually produce more. Let $\bar{Q}$ be the query that computes the complement of Q, i.e., for an n-ary Q, the result of $\bar{Q}(D)$ is $\mathrm{adom}(D)^n - Q(D)$. Then we actually provide

a translation

$$Q \;\mapsto\; (Q^+, Q^-)$$

of RA queries into a pair of $\mathsf{RA}_{\mathsf{null}}$ queries such that

$$Q^+(D) \;\subseteq\; \mathsf{certain}_\perp(Q, D) \quad \text{and} \quad Q^-(D) \subseteq \mathsf{certain}_\perp(\bar{Q}, D).$$

When this happens, we say that the translation $Q \;\mapsto\; (Q^+, Q^-)$ *provides correctness guarantees*.

Since $\mathsf{certain}_\perp(Q, D) \cap \mathsf{certain}_\perp(\bar{Q}, D) = \emptyset$, this also means that $Q^+(D) \cap Q^-(D) = \emptyset$. One can think of Q^+ and Q^- as analogs of finding tuples for which $\mathsf{Eval}_{3\mathsf{v}}$ produces 0 or 1. Everything that does not fall into the results of these two, is essentially 'unknowns'.

We now provide the translations. We need three auxiliary elements: a translation $\theta \mapsto \theta^*$ from RA conditions to $\mathsf{RA}_{\mathsf{null}}$ conditions, one RA query, and one $\mathsf{RA}_{\mathsf{null}}$ query. These are given as follows:

The translation $\theta \mapsto \theta^*$ is defined inductively. We assume that in conditions $\sharp n = \sharp m$ or $\sharp n \neq \sharp m$, attributes $\sharp n$ and $\sharp m$ are different (otherwise they are easily eliminated).

- If θ is $(\sharp n = \sharp m)$ or $(\sharp m = c)$, where c is a constant, then $\theta^* = \theta$.
- $(\sharp n \neq \sharp m)^* = (\sharp n \neq \sharp m) \wedge \mathsf{const}(\sharp n) \wedge \mathsf{const}(\sharp m)$.
- $(\sharp n \neq c)^* = (\sharp n \neq c) \wedge \mathsf{const}(\sharp n)$.
- $(\theta_1 \vee \theta_2)^* = \theta_1^* \vee \theta_2^*$.
- $(\theta_1 \wedge \theta_2)^* = \theta_1^* \wedge \theta_2^*$.

Active domain query We use adom as an RA query that returns the active domain of a database; clearly it can be written as a $\pi, \cup$-query, that takes the union of all projections of all relations in the database.

Relative complement query The *relative complement* of a k-ary relation R in database D is

$$R^\ominus \;=\; \{\bar{u} \in \mathrm{adom}(D)^k \mid \neg\exists \bar{t} \in R : \; \bar{u} \Uparrow \bar{t}\,\}.$$

It is not hard to see that $R^\ominus$ is expressible in $\mathsf{RA}_{\mathsf{null}}$. We show this formally in the proof of Theorem 11. In fact this is the only expression where conditions $\mathsf{null}(\sharp n)$ are used in selections.

With these, translations of relational algebra are given by inductive rules presented in Figure 1. We use abbreviation $\mathrm{ar}(Q)$ for the arity of Q, and $\boldsymbol{\alpha}$ refers to a list of attributes.

▶ **Theorem 11.** *The translation $Q \;\mapsto\; (Q^+, Q^-)$ in Figure 1 provides correctness guarantees.*

Proof sketch. Again, we show the following, by induction on relational algebra expressions:

$$\bar{u} \in Q^+(D) \;\Rightarrow\; \forall \text{ homomorphism } h : \; h(\bar{u}) \in Q(h(D)) \qquad (\checkmark)$$

$$\bar{u} \in Q^-(D) \;\Rightarrow\; \forall \text{ homomorphism } h : \; h(\bar{u}) \notin Q(h(D)) \qquad (\checkmark\checkmark)$$

We provide a couple of sample cases. Consider, for instance, the case when Q is R. Then $Q^- = R^\ominus$. Assume $\bar{u} \in R^\ominus$ and let h be a homomorphism. By definition, $\bar{u}$ does not unify with any of $\bar{t} \in R$, in particular, $h(\bar{u})$ cannot equal $h(\bar{t})$, thus implying $h(\bar{u}) \notin h(R)$.

Let $\theta = (\sharp n \neq \sharp m)$, and assume $Q = \sigma_\theta(Q_1)$ (so that $Q^+ = \sigma_{\theta^*}(Q_1^+)$). Suppose $\bar{u} \in \sigma_{\theta^*}(Q_1^+(D))$, and let h be a homomorphism. Since $\bar{u} \in Q_1^+(D)$, we see, by the hypothesis,

$$
\begin{array}{rclrcl}
R^+ & = & R & R^- & = & R^{\ominus} \\
(Q_1 \cup Q_2)^+ & = & Q_1^+ \cup Q_2^+ & (Q_1 \cup Q_2)^- & = & Q_1^- \cap Q_2^- \\
(Q_1 \cap Q_2)^+ & = & Q_1^+ \cap Q_2^+ & (Q_1 \cup Q_2)^- & = & Q_1^- \cup Q_2^- \\
(Q_1 - Q_2)^+ & = & Q_1^+ \cap Q_2^- & (Q_1 - Q_2)^- & = & Q_1^- \cup Q_2^+ \\
(\sigma_\theta(Q))^+ & = & \sigma_{\theta^*}(Q^+) & (\sigma_\theta(Q))^- & = & Q^- \cup \sigma_{(\neg\theta)^*}(\mathrm{adom}^{\mathrm{ar}(Q)}) \\
(Q_1 \times Q_2)^+ & = & Q_1^+ \times Q_2^+ & (Q_1 \times Q_2)^- & = & Q_1^- \times \mathrm{adom}^{\mathrm{ar}(Q_2)} \\
 & & & & & \cup\ \mathrm{adom}^{\mathrm{ar}(Q_1)} \times Q_2^- \\
(\pi_{\boldsymbol{\alpha}}(Q))^+ & = & \pi_{\boldsymbol{\alpha}}(Q^+) & (\pi_{\boldsymbol{\alpha}}(Q))^- & = & \pi_{\boldsymbol{\alpha}}(Q^-) - \pi_{\boldsymbol{\alpha}}(\mathrm{adom}^{\mathrm{ar}(Q)} - Q^-)
\end{array}
$$

Figure 1 Relational algebra translations

that $h(\bar{u}) \in Q_1(h(D))$. Furthermore, since θ^* holds, we know that u_n and u_m, the nth and the mth components of $\bar{u}$, are constants, and $u_n \neq u_m$. This means $h(u_n) \neq h(u_m)$, proving $h(\bar{u}) \in \sigma_\theta(Q_1(h(D)))$. □

The translation in Figure 1 is not just one translation but rather a family of translations, due to the following observation. A translation can be viewed as a mapping $\mathcal{F}$ that assigns to each relational algebra operation ω (including nullary operations for base relations) two queries F_ω^+ and F_ω^-. These queries are simply the queries that appear on the right in the translation; for instance, for the translation scheme we used, $F_\cap^+$ is the intersection (since the result of $(Q_1 \cap Q_2)^+$ is the intersection of Q_1^+ and Q_2^+) and $F_\cap^-$ is the union (since the result of $(Q_1 \cap Q_2)^-$ is the union of Q_1^- and Q_2^-).

Such a mapping $\mathcal{F}$ results in a translation $Q \mapsto \mathcal{F}_Q^+, \mathcal{F}_Q^-$, where $\mathcal{F}_Q^+$ and $\mathcal{F}_Q^-$ are queries of the same type as Q (i.e., they operate on databases of the same schema and have the same arity). Intuitively, these are analogs of Q^+ and Q^- that we had for the translation in Figure 1.

Formally, they are defined as follows.

- If ω is a base relation R, then F_R^+ and F_R^- take no arguments and $\mathcal{F}_R^+ = F_R^+$ and $\mathcal{F}_R^- = F_R^-$.

  That is, F_R^+ and F_R^- are queries that give us certainly positive and certainly negative information about R.
- If ω is a unary operation (σ or π), then F_ω^+ and F_ω^- take two arguments and $\mathcal{F}_{\omega(Q)}^+ = F_\omega^+(\mathcal{F}_Q^+, \mathcal{F}_Q^-)$ and $\mathcal{F}_{\omega(Q)}^- = F_\omega^-(\mathcal{F}_Q^+, \mathcal{F}_Q^-)$.

  That is, if we already have queries $\mathcal{F}_Q^+$ and $\mathcal{F}_Q^-$ describing certainly positive and certainly negative answers for Q, the queries describing such answers for $\omega(Q)$ are obtained by applying F_ω^+ and F_ω^- to those.
- If ω is a binary operation ($\cup, \cap, -, \times$), then F_ω^+ and F_ω^- take four arguments and $\mathcal{F}_{\omega(Q_1,Q_2)}^+ = F_\omega^+(\mathcal{F}_{Q_1}^+, \mathcal{F}_{Q_2}^+, \mathcal{F}_{Q_1}^-, \mathcal{F}_{Q_2}^-)$ and $\mathcal{F}_{\omega(Q_1,Q_2)}^- = F_\omega^-(\mathcal{F}_{Q_1}^+, \mathcal{F}_{Q_2}^+, \mathcal{F}_{Q_1}^-, \mathcal{F}_{Q_2}^-)$.

  That is, if we already have queries $\mathcal{F}_{Q_i}^+$ and $\mathcal{F}_{Q_i}^-$ describing certainly positive and certainly negative answers for Q_i, with $i = 1, 2$, the queries describing such answers for $\omega(Q_1, Q_2)$ are again obtained by applying F_ω^+ and F_ω^- to those.

Given a translation $\mathcal{F}$ and another translation $\mathcal{G}$ that assigns to each operation ω queries G_ω^+ and G_ω^-, we say that $\mathcal{F}$ is *contained* in $\mathcal{G}$ if $F_\omega^+ \subseteq G_\omega^+$ and $F_\omega^- \subseteq G_\omega^-$, where $\subseteq$ refers to the usual query containment.

▶ **Proposition 12.** *Every translation that is contained in the translation of Figure 1 provides correctness guarantees.*

This proposition lets us adjust translations for the sake of efficiency without having to worry about correctness guarantees. For instance, consider the rule

$$(Q_1 \times Q_2)^- \;=\; Q_1^- \times \mathrm{adom}^{\mathrm{ar}(Q_2)} \;\cup\; \mathrm{adom}^{\mathrm{ar}(Q_1)} \times Q_2^-$$

in Figure 1. This results in a rather expensive query, as one needs to compute a power of the active domain. But we can replace it with the much simpler rule $(Q_1 \times Q_2)^- = Q_1^- \times Q_2^-$, since $Q_1^- \times Q_2^-$ is contained in the above query, giving us a more efficient query. Another possible replacement is of the rule

$$(\sigma_\theta(Q))^- \;=\; Q^- \cup \sigma_{(\neg\theta)^*}(\mathrm{adom}^{\mathrm{ar}(Q)})$$

that again requires computing the active domain with the very simple rule $(\sigma_\theta(Q))^- = Q^-$. In both cases the result is that the translated queries are significantly more efficient and they still guarantee correctness of the overall translation in the sense that they produce subsets of certain answers with nulls, or the usual certain answers if tuples with nulls are removed. There is a price to pay for the efficiency though: we can get fewer answers in the result. Hence one should decide how to resolve the efficiency vs the quality of approximation tradeoff.

Another corollary concerns positive relational algebra, even extended with inequalities, and it just follows from examining the basic translation of Figure 1. Define $\mathsf{PosRA}^{\neq}$ as the positive fragment of RA (i.e., $\sigma, \pi, \times, \cup$) where conditions in selections are allowed to use inequalities. In terms of its expressiveness, this fragment corresponds to $\mathsf{UCQ}^{\neq}$.

▶ **Corollary 13.** *Let Q be a $\mathsf{PosRA}^{\neq}$ query, and let Q^* be obtained from it by changing each selection condition θ to θ^*. Then, for every database D, we have $Q^*(D) \subseteq \mathsf{certain}_\perp(Q, D)$.*

7 Conclusions

We have shown that small changes to the 3-valued query evaluation used in SQL produce sound query answers, i.e., answers without false positives. We have presented such evaluation procedures at the levels of both relational calculus and algebra, and also specialized them for unions of conjunctive queries with inequalities.

The theoretical complexity of these procedures is very low, in fact it is as low as evaluating relational calculus and algebra themselves, in terms of data complexity. The next obvious step is to implement these algorithms to study their real-life applicability. As indicated at the end of the last section, our translations – especially at the procedural level – are really families of algorithms, with the efficiency vs quality of approximation tradeoff, so there is a lot to play with, to find those that provide a good combination of both. Another natural question is to consider other features of SQL. They include not only such common features as aggregation and grouping, but also derived operations of relational algebra that are used in implementation of SQL queries: for instance, the division operation for the implementation of some universal queries, or semi-joins and anti-joins that can be used for implementing subqueries.

Acknowledgment I thank Cristina Sirangelo and the reviewers for their helpful comments and suggestions. Work partially supported by EPSRC grant J015377.

References

1 S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

2 S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78(1):158–187, 1991.

3 M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

4 G. Bruns and P. Godefroid. Model checking with multi-valued logics. In *ICALP*, pages 281–293, 2004.

5 K. Compton. Some useful preservation theorems. *Journal of Symbolic Logic*, 48(2):427–440, 1983.

6 C. J. Date. *Database in Depth - Relational Theory for Practitioners*. O'Reilly, 2005.

7 C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.

8 A. Gheerbrant, L. Libkin, and C. Sirangelo. Naïve evaluation of queries over incomplete databases. *ACM Transactions on Database Systems*, 39(4): 34 (2014).

9 A. Gheerbrant, L. Libkin, and T. Tan. On the complexity of query answering over incomplete XML documents. In *ICDT*, pages 169–181, 2012.

10 A. Gurfinkel and M. Chechik. Multi-valued model checking via classical model checking. In *CONCUR*, pages 263–277, 2003.

11 L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, pages 805–810, 2005.

12 T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

13 M. Lenzerini. Data integration: a theoretical perspective. In *PODS*, pages 233–246, 2002.

14 H. J. Levesque. A completeness result for reasoning with incomplete first-order knowledge bases. In *Principles of Knowledge Representation and Reasoning (KR)*, pages 14–23, 1998.

15 L. Libkin. Certain answers as objects and knowledge. In *Principles of Knowledge Representation and Reasoning (KR)*, 2014.

16 L. Libkin. Incomplete information: what went wrong and how to fix it. In *PODS*, pages 1–13, 2014.

17 W. Lipski. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, 1979.

18 Y. Liu and H. J. Levesque. A tractability result for reasoning with incomplete first-order knowledge bases. In *IJCAI*, pages 83–88, 2003.

19 B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an opensource tool for second-generation schema mapping and data exchange. *PVLDB*, 4(12):1438–1441, 2011.

20 M. Paterson and M. N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

21 R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.

22 R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the ACM*, 33(2):349–347, 1986.