Correctness of SQL Queries on Databases with Nulls

Paolo Guagliardo
School of Informatics
The University of Edinburgh
pguaglia@inf.ed.ac.uk

Leonid Libkin
School of Informatics
The University of Edinburgh
libkin@inf.ed.ac.uk

ABSTRACT

Multiple issues with SQL's handling of nulls have been well documented. Having efficiency as its main goal, SQL disregards the standard notion of correctness on incomplete databases – certain answers – due to its high complexity. As a result, the evaluation of SQL queries on databases with nulls may produce answers that are just plain wrong. However, SQL evaluation can be modified, at least for relational algebra queries, to approximate certain answers, i.e., return only correct answers. We examine recently proposed approximation schemes for certain answers and analyze their complexity, both theoretical bounds and real-life behavior.

1. INTRODUCTION

The way incomplete information is handled in commercial DBMSs, specifically by SQL, has been heavily criticized for producing counter-intuitive and just plain incorrect answers [4, 9]. This is often blamed on SQL's 3-valued logic (3VL), and there are multiple discussions in the literature on the relative merits of SQL's 3VL and some alternatives; see, e.g., [11, 33, 6]. They often try to justify a logic within itself, without having an external yardstick definition of correctness. Given the futility of such an approach, we first need to settle on what constitutes the notion of correctness.

For this, we adapt the standard approach found in the database literature: correct answers are those that we are *certain* about. Intuitively, this means that such answers will be true no matter how we interpret incomplete information that is present in the database. This approach, first proposed in the late 1970s [13, 26], is now dominant in the literature and it is standard in all applications where incomplete information appears (data integration, data exchange, ontology-based data access, data cleaning, etc.).

Why cannot SQL then just compute certain answers? The reason is that SQL's designers had first and foremost efficient evaluation in mind, but correctness and efficiency do not always get along. Computing certain answers is coNP-hard for most reasonable semantics, if we deal with relational calculus/algebra queries [2]. On the other hand, SQL evaluation is very efficient; it is in AC^0 (a small parallel complexity class) for the same class of queries, and so it provably cannot compute certain answers.

If SQL provably cannot produce what is assumed to be *the* correct answers, then what kinds of errors can it generate? To understand this, consider the simple database in Figure 1. It shows orders for books, information about customers paying for them, and basic information about customers themselves.

Decision support queries against such a database may include finding *unpaid orders*:

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

or finding customers *who have not placed an order*:

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

As expected, the first query produces a single answer Ord3, while the second returns the empty table. But now assume that just a single entry in these tables is replaced by `NULL`: specifically, the value of `order_id` in the second tuple of `Payments` changes from Ord2 to `NULL`. Then the answers to queries change drastically, and in *different ways*: now the unpaid orders query returns the empty table, and the customers without an order query returns Cust2. That is, due to the presence of nulls, we can both miss answers, and invent new answers!

Let us analyze this in more detail. If we consider certain answers as the correct behavior of query answering over incomplete databases, then SQL evaluation can differ from it in two ways:

- SQL can miss some of the tuples that belong to certain answers, thus producing *false negatives*; or
- it can return some tuples that do not belong to certain answers, that is, *false positives*.

In the previous example, Cust2 returned by the second

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

query is a false positive. The unpaid orders query does not generate any false negatives: certain answers are actually empty since we cannot know which order was unpaid. But a simple query

```
SELECT cust_id FROM Payments
WHERE order_id = 'Ord2' OR order_id <> 'Ord2'
```

returns only Cust1 in the database with null as described above, while the certain answer is { Cust1, Cust2 }.

To sum up, SQL cannot compute certain answers due to the complexity gap. Furthermore, it can produce *both* false positives and false negatives. However, the gap in complexity does not yet justify such a behavior: it leaves open the possibility that a query evaluation scheme produces *only one type of undesirable results*.

For now we take the view that false positives are the worst of the two: after all, they produce an outright lie as opposed to hiding some of the truth. We admit that an alternative point of view has merits too [6], and in fact we shall address it later. One can accept one type of errors – false negatives – as the price to be paid for lowering complexity.

This idea is not new: it was first explored more than 30 years ago [29, 32]. Those papers assumed the model of databases as logical theories and could not lead to implementations that would handle familiar relational databases with nulls. Some ad hoc translations of SQL queries were studied later but without any formally proved correctness guarantees [19].

The first approach to fixing SQL's evaluation scheme that provides provable correctness guarantees was presented surprisingly recently, in [25] (with the conference version appearing in 2015). It showed how to translate a relational algebra query Q into a query $Q^{\mathbf{t}}$ of **t**rue answers such that:

- false positives never occur: $Q^{\mathbf{t}}$ returns a subset of certain answers to Q;
- data complexity of $Q^{\mathbf{t}}$ is still AC^0; and
- on databases without nulls, Q and $Q^{\mathbf{t}}$ coincide.

Given the attractive theoretical properties of the approach, it was natural to ask two questions. First, do we address a real problem, that is, do false positives occur in real queries? And second, do theoretical guarantees of the approach translate into good behavior in practice? What is the price to pay, in terms of query evaluation performance, for correctness guarantees?

These questions were addressed in [15]. It provided experimental evidence that false positives are indeed a real-life problem. It then noticed that the translation of [25] cannot be implemented as-is: queries in translations tend to build very large Cartesian products and are thus impractical, despite very good theoretical complexity bounds.

To remedy this, [15] proposed a new translation $Q \mapsto (Q^+, Q^?)$ of relational algebra queries. The query Q^+ shares the desirable properties of $Q^{\mathbf{t}}$, and $Q^?$ addresses the alternative point of view that false negatives are evil: it eliminates false negatives but can produce false positives instead. The translations are by mutual recursion, hence one cannot define Q^+ without $Q^?$ and vice versa.

Algorithms in [15] introduced extra steps to restore correctness. We do not, therefore, expect them to outperform native SQL evaluation, which was designed to optimize execution time. We can hope, however, that the overhead is sufficiently small. If this is so, one can envision two modes of evaluation: the standard one, where efficiency is the only concern, and an alternative, perhaps slightly more expensive one, that provides correctness guarantees. The difference between the two is the *price of correctness*.

The goal of this short survey is to report recent developments in finding efficient approximations of SQL queries on databases with nulls that come with correctness guarantees. Experimental evidence shows that false positives are a real issue. We present the approximation schemes of [25] and [15] and provide theoretical guarantees for both. For the latter, we also present an experimental evaluation showing that its real-life behavior in terms of the price of correctness falls into three major categories:

- for the first, and largest, group of queries, the price of correctness is small, as was indeed hoped (the overhead ranges between 1% and 4%);
- for another group, somewhat surprisingly, there is a significant improvement in performance despite the query performing additional checks;
- for the last group, performance becomes an issue, but it has to do with the well documented issues in the way commercial optimizers handle disjunctions in queries [5]; depending on the size of the data-

base, the approximating query Q^+ runs at between one quarter to half the speed of Q.

These results point to a real opportunity to fix many of the issues related to the handling of nulls in RDBMSs, at a reasonable cost in terms of query evaluation.

2. PRELIMINARIES

We consider incomplete databases with nulls interpreted as missing values. Much of the following is standard in the literature on databases with incomplete information; see, e.g., [1, 18, 31]. The usual way of modeling missing values in a database is to use *marked* (or labeled) nulls, which often appear in applications such as data integration and exchange [3, 22]. In this model, databases are populated by two types of elements: *constants* and *nulls*, coming from countably infinite sets denoted by Const and Null, respectively. Nulls are denoted by $\perp$, sometimes with sub- or superscripts. For the purpose of the general model we follow the textbook approach assuming one domain Const for all non-null elements appearing in databases. In real life, such elements can be of many different types, and those appearing in the same column must be of the same type. Adjusting results and translations of queries for this setting is completely straightforward.

A relational schema is a set of relation names with associated arities (numbers of attributes). With each k-ary relation symbol S from the vocabulary, an incomplete relational instance D associates a k-ary relation S^D over $\mathsf{Const} \cup \mathsf{Null}$, that is, a finite subset of $(\mathsf{Const} \cup \mathsf{Null})^k$. When the instance is clear from the context, we write S instead of S^D for the relation itself. We denote the arity of S by $\mathsf{ar}(S)$, and use the same notation for queries. Note that we now assume *set* semantics of queries; we shall comment on *bag* semantics, which is used in real-life DBMSs, in Section 5.

The sets of constants and nulls that occur in a database D are denoted by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$, respectively. The *active domain* of D is the set $\mathsf{adom}(D)$ of all elements occurring in it, that is, $\mathsf{Const}(D) \cup \mathsf{Null}(D)$. If D has no nulls, we say that it is *complete*. A *valuation* v on a database D is a map $v : \mathsf{Null}(D) \to \mathsf{Const}$. We denote by $v(D)$ the result of replacing each null $\perp$ with $v(\perp)$ in D. An incomplete database represents the collection of complete databases $\{v(D) \mid v \text{ is a valuation}\}$; this is often referred to as the closed-world semantics of incompleteness [28].

Query languages. As our query language, we consider relational algebra with the standard operations of selection σ, projection π, Cartesian product $\times$ (or join $\bowtie$), union $\cup$, and difference $-$. This corresponds to the basic fragment of SQL – which we use in the experiments of Section 4 – consisting of the usual `SELECT-FROM-WHERE` queries, with (correlated) subqueries preceded by `IN` and `EXISTS`, as well as their negations. We shall comment in more detail about the correspondence between SQL and relation algebra in Section 5.

We assume that selection conditions are positive Boolean combinations of equalities of the form $A = B$ and $A = c$, where A and B are attributes and c is a constant value, and disequalities $A \neq B$ and $A \neq c$. Note that these conditions are closed under negation, which can simply be propagated to atoms: e.g., $\neg\big((A = B) \vee (B \neq 1)\big)$ is equivalent to $(A \neq B) \wedge (B = 1)$.

We also use conditions $\mathsf{const}(A)$ and $\mathsf{null}(A)$ in selections, indicating whether the value of an attribute is a constant or a null. These correspond to `A IS NOT NULL` and `A IS NULL` in SQL.

Correctness guarantees. The standard notion of correct query answering on incomplete databases is *certain answers*, that is, tuples that are present in the answer to a query regardless of the interpretation of nulls. For a query Q and a database D, they are typically defined as tuples $\bar{a}$ that are present in $Q(v(D))$ for all valuations v; see [1, 18].

This definition has a serious drawback, though, as tuples with nulls cannot be returned, while standard query evaluation may well produce such tuples. For instance, if we have a relation $R = \{(1, \perp), (2, 3)\}$, and a query returning R, then the only certain answer according to the above definition is $(2, 3)$, while intuitively we would expect the entire relation.

In light of this, we use a closely-related but more general notion from [27], called *certain answers with nulls* in [25]. Formally, for a query Q and a database D, these are tuples $\bar{a}$ over $\mathsf{adom}(D)$ such that $v(\bar{a}) \in Q\big(v(D)\big)$ for every valuation v on D. The set of all such tuples is denoted by $\mathsf{cert}(Q, D)$. In the above example, the certain answers with nulls are $(1, \perp)$ and $(2, 3)$. The standard certain answers are exactly the null-free tuples in $\mathsf{cert}(Q, D)$ [25].

Definition 1. A query evaluation algorithm has *correctness guarantees* for query Q if for every database D it returns a subset of $\mathsf{cert}(Q, D)$.

In other words, with correctness guarantees, false positives are not allowed: all returned tuples must be certain answers.

Often our evaluation algorithms will be of the following form: translate a query Q into another query Q', and then run Q' on D. If $Q'(D) \subseteq \mathsf{cert}(Q, D)$ for every D, then we say that Q' has correctness guarantees for Q.

Some results concerning correctness guarantees are known. By *naïve evaluation* for a fragment of relational algebra we mean the algorithm that treats elements of Null as if they were the usual database entries, i.e., each

evaluation $\bot = c$ for $c \in \mathsf{Const}$ is false and $\bot = \bot'$ is true iff $\bot$ and $\bot'$ are the same element in Null.

Recall that the *positive* fragment of relational algebra is the fragment without the difference operator and without disequalities in selection conditions. It corresponds to the fragment of SQL in which negation does not appear in any form, i.e., **EXCEPT** is not allowed, there are no negations in **WHERE** conditions and the use of **NOT IN** and **NOT EXISTS** for subqueries is prohibited.

FACT 1 ([12, 18, 25]). *For positive relational algebra queries, naïve evaluation computes exactly certain answers with nulls, and thus it has correctness guarantees. This remains true even if we extend the language with the division operator as long as its second argument is a relation in the database.*

Recall that division is a derived relational algebra operation; it computes tuples in a projection of a relation appearing in all possible combinations with tuples from another relation (e.g., 'find students taking all courses').

SQL evaluation. The query evaluation procedure in SQL is different from naïve evaluation: it is based on a 3-valued logic. Comparisons such as $\bot = c$, or $\bot = \bot'$, evaluate to *unknown*, which is then propagated through conditions using the rules of 3VL.

More precisely, selection conditions can evaluate to true (**t**), false (**f**), or unknown (**u**). If at least one attribute in a comparison is null, the result of the comparison is **u**. The interaction of **u** with Boolean connectives follows the rules of SQL's 3VL (which is Kleene's 3-valued logic) shown below for the cases when **u** is involved:

$$\neg\mathbf{u} = \mathbf{u} \qquad \mathbf{u} \wedge \mathbf{f} = \mathbf{f} \qquad \mathbf{u} \vee \mathbf{t} = \mathbf{t}$$
$$\mathbf{u} \wedge \mathbf{t} = \mathbf{u} \qquad \mathbf{u} \wedge \mathbf{u} = \mathbf{u}$$
$$\mathbf{u} \vee \mathbf{f} = \mathbf{u} \qquad \mathbf{u} \vee \mathbf{u} = \mathbf{u}$$

Then, σ_θ selects tuples on which θ evaluates to **t** (that is, **f** and **u** tuples are not selected). We refer to the result of evaluating a query Q in this way as $\mathsf{Eval}_{\mathsf{SQL}}(Q, D)$.

FACT 2 ([25]). $\mathsf{Eval}_{\mathsf{SQL}}$ *has correctness guarantees for positive relational algebra.*

Thus, it is the negation in queries – that may appear in various forms – that causes SQL's behavior to deviate from correct answers. Not surprisingly, all the example queries in the introduction used some form of negation.

3. APPROXIMATION SCHEMES WITH CORRECTNESS GUARANTEES

Due to the high complexity of certain answers, we must settle for approximations that can be computed efficiently. As we have seen, although efficient, standard SQL evaluation may produce answers that are not certain, so we need alternative evaluation schemes that have correctness guarantees and tractable complexity.

One such scheme was first devised in [25], but despite its promising complexity bounds it was not effectively applicable in practice. For this reason, [15] proposed a new evaluation scheme with correctness guarantees and the same theoretical complexity of the previous one, but that can also be implemented efficiently.

We will now present and discuss these two schemes in more detail for queries expressed in relational algebra.

3.1 A simple translation

The key idea of the approximation scheme of [25] is to translate a query Q into a pair $(Q^{\mathbf{t}}, Q^{\mathbf{f}})$ of queries that have correctness guarantees for Q and its complement $\overline{Q}$, respectively. That is, tuples in $Q^{\mathbf{t}}(D)$ are certainly true, and tuples in $Q^{\mathbf{f}}(D)$ are certainly false:

$$Q^{\mathbf{t}}(D) \subseteq \mathsf{cert}(Q, D) \tag{1}$$
$$Q^{\mathbf{f}}(D) \subseteq \mathsf{cert}(\overline{Q}, D) \tag{2}$$

To describe the translation, we need the following.

Definition 2. Two tuples $\bar{r}$ and $\bar{s}$ of the same length over $\mathsf{Const} \cup \mathsf{Null}$ are *unifiable*, written as $\bar{r} \Uparrow \bar{s}$, if there exists a valuation v of nulls such that $v(\bar{r}) = v(\bar{s})$.

For example, $(\bot, 2, \bot') \Uparrow (2, \bot, 3)$ with the valuation $v(\bot) = 2$ and $v(\bot') = 3$, but $(\bot, 3, \bot')$ and $(2, \bot, 3)$ do not unify. Checking whether tuples unify is very efficient: it can be done in linear time, and in fact can be expressed by a condition in **WHERE**.

The translations of [25] are shown in Figure 2, where adom refers to the query computing the active domain. For a single relation R with attributes $A_1, \ldots, A_n$, this is $\mathsf{adom}(R) = \pi_{A_1}(R) \cup \ldots \cup \pi_{A_n}(R)$, and for a database D with relations $R_1, \ldots, R_m$, it is $\mathsf{adom}(D) = \mathsf{adom}(R_1) \cup \ldots \cup \mathsf{adom}(R_m)$. Recall that $\mathsf{ar}(Q)$ is the arity of Q, so $\mathsf{adom}^{\mathsf{ar}(Q)}$ refers to the Cartesian product $\mathsf{adom} \times \ldots \times \mathsf{adom}$ taken $\mathsf{ar}(Q)$ times.

The translation also uses conditions θ^* which are obtained by translating selection conditions θ as defined inductively by the following rules:

$$\begin{aligned}
(A = B)^* &= (A = B) \\
(A = c)^* &= (A = c) \quad \text{if } c \text{ is a constant} \\
(A \neq B)^* &= (A \neq B) \wedge \mathsf{const}(A) \wedge \mathsf{const}(B) \\
(A \neq c)^* &= (A \neq c) \wedge \mathsf{const}(A) \\
(\theta_1 \vee \theta_2)^* &= \theta_1^* \vee \theta_2^* \\
(\theta_1 \wedge \theta_2)^* &= \theta_1^* \wedge \theta_2^*
\end{aligned}$$

THEOREM 1 ([25]). *The translations of Figure 2 have correctness guarantees: (1) and (2) hold. Moreover, both queries $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ have AC^0 data complexity, and $Q^{\mathbf{t}}(D) = Q(D)$ for complete databases.*

$$
\begin{array}{rcl@{\qquad}rcl}
R^{\mathbf{t}} & = & R & R^{\mathbf{f}} & = & \{\, \bar{s} \in \mathsf{adom}^{\mathsf{ar}(R)} \mid \nexists\, \bar{r} \in R\colon \bar{r} \Uparrow \bar{s} \,\} \\
(Q_1 \cup Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \cup Q_2^{\mathbf{t}} & (Q_1 \cup Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \cap Q_2^{\mathbf{f}} \\
(Q_1 - Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}} & (Q_1 - Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \cup Q_2^{\mathbf{t}} \\
\big(\sigma_\theta(Q)\big)^{\mathbf{t}} & = & \sigma_{\theta^*}(Q^{\mathbf{t}}) & (\sigma_\theta(Q))^{\mathbf{f}} & = & Q^{\mathbf{f}} \cup \sigma_{(\neg\theta)^*}\big(\mathsf{adom}^{\mathsf{ar}(Q)}\big) \\
(Q_1 \times Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \times Q_2^{\mathbf{t}} & (Q_1 \times Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \times \mathsf{adom}^{\mathsf{ar}(Q_2)} \cup \mathsf{adom}^{\mathsf{ar}(Q_1)} \times Q_2^{\mathbf{f}} \\
\big(\pi_\alpha(Q)\big)^{\mathbf{t}} & = & \pi_\alpha(Q^{\mathbf{t}}) & \big(\pi_\alpha(Q)\big)^{\mathbf{f}} & = & \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha\big(\mathsf{adom}^{\mathsf{ar}(Q)} - Q^{\mathbf{f}}\big)
\end{array}
$$

Figure 2: Relational algebra translations of [25].

While (1) and (2) ensure correctness guarantees for all relational algebra queries, and queries $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ have good theoretical complexity, they suffer from a number of problems that severely hinder their practical implementation. Crucially, they require the computation of active domains and, even worse, their Cartesian products. While expressible in relational algebra, the $Q^{\mathbf{f}}$ translations for selections, products, projections, and even base relations become prohibitively expensive. Several optimizations have been suggested in [25] (at the price of missing some certain answers), but the cases of projection and base relations do not appear to have any reasonable alternatives. Yet another problem is the complicated structure of the queries $Q^{\mathbf{f}}$. When translations are applied recursively, this leads to very complex queries $Q^{\mathbf{t}}$ if Q used difference.

In fact we tried a simple experiment with the translations in Figure 2, and found that they are already infeasible for very small databases: some of the queries start running out of memory on instances with fewer than 10^3 tuples.

All this tells us that good theoretical complexity is not yet a guarantee of real-life efficiency, and we need an implementable alternative, which we present next.

3.2 An implementation-friendly translation

To overcome the practical difficulties posed by the translation in Figure 2, [15] proposed an alternative translation that is implementation-friendly and comes with sufficient correctness guarantees. This translation does not produce a second query $Q^{\mathbf{f}}$ that underapproximates certain answers to the negation of the query, which was the main source of complexity. To see what we can replace it with, note that, in the $Q^{\mathbf{t}}$ translation, $Q^{\mathbf{f}}$ was only used in the rule for difference: a tuple $\bar{a}$ is a certain answer to $Q_1 - Q_2$ if

1. $\bar{a}$ is a certain answer to Q_1, and
2. $\bar{a}$ is a certain answer to the complement of Q_2.

That necessitated working with the complex $Q^{\mathbf{f}}$ translation.

But we can use a slightly different rule: a tuple $\bar{a}$ is a certain answer to $Q_1 - Q_2$ if

1. $\bar{a}$ is a certain answer to Q_1, and
2. $\bar{a}$ does not match any tuple that could possibly be an answer to Q_2.

The advantage of this is that the query that approximates possible answers can be built in a much simpler way than $Q^{\mathbf{f}}$. For instance, for a base relation R, it will be just R itself, as opposed to the complex expression involving adom we used before. Then the rule for $Q_1 - Q_2$ involves a left anti-semijoin (to be defined soon) of the approximation of certain answers to Q_1 and possible answers to Q_2.

We need to formally say what "(not) matching possible answers" means. To this end, we define approximations of possible answers and two matching-based semijoin operators. There already exists a notion of *maybe-answers* [2, 31] – answers that appear in $Q\big(v(D)\big)$ for at least one valuation v – but those can be infinite, and include arbitrary elements outside of $\mathsf{adom}(D)$. What we need instead is a compact representation.

Definition 3. Given a k-ary query Q and an incomplete database D, we say that a set $A \subseteq \mathsf{adom}(D)^k$ *represents potential answers* to Q on D if $Q\big(v(D)\big) \subseteq v(A)$ for every valuation v. A query Q' *represents potential answers* to Q if $Q'(D)$ represents potential answers to Q on D, for every D.

Obviously, there are trivial ways of representing potential answers: take, e.g., $\mathsf{adom}(D)^k$. But we shall be looking for good approximations, just as we are looking for good approximations of $\mathsf{cert}(Q, D)$, for which bad ones can also be found easily (e.g., the empty set). In general, testing if a set A represents potential answers to a query is computationally hard:

PROPOSITION 1 ([15]). *There is a fixed relational algebra query Q such that the following problem is* coNP*-complete: given a database D and a set A of tuples over $\mathsf{adom}(D)$, does A represent potential answers to Q on D?*

$$R^+ = R \tag{3.1}$$
$$(Q_1 \cup Q_2)^+ = Q_1^+ \cup Q_2^+ \tag{3.2}$$
$$(Q_1 - Q_2)^+ = Q_1^+ \mathbin{\overline{\ltimes}_{\Uparrow}} Q_2^? \tag{3.3}$$
$$\big(\sigma_\theta(Q)\big)^+ = \sigma_{\theta^*}(Q^+) \tag{3.4}$$
$$(Q_1 \times Q_2)^+ = Q_1^+ \times Q_2^+ \tag{3.5}$$
$$\big(\pi_\alpha(Q)\big)^+ = \pi_\alpha(Q^+) \tag{3.6}$$

$$R^? = R \tag{4.1}$$
$$(Q_1 \cup Q_2)^? = Q_1^? \cup Q_2^? \tag{4.2}$$
$$(Q_1 - Q_2)^? = Q_1^? - Q_2^+ \tag{4.3}$$
$$\big(\sigma_\theta(Q)\big)^? = \sigma_{\theta^{**}}\big(Q^?\big) \tag{4.4}$$
$$(Q_1 \times Q_2)^? = Q_1^? \times Q_2^? \tag{4.5}$$
$$\big(\pi_\alpha(Q)\big)^? = \pi_\alpha\big(Q^?\big) \tag{4.6}$$

Figure 3: Improved relational algebra translations of [15].

However, we shall see that potential answers can be efficiently approximated, which is what we need for the translation.

To express conditions involving matching, we shall need two semijoin operations based on unifiable tuples (see Definition 2).

Definition 4. For relations R, S over $\mathsf{Const} \cup \mathsf{Null}$, with the same set of attributes, the *left unification semijoin* is

$$R \ltimes_{\Uparrow} S = \big\{\, \bar{r} \in R \mid \exists\, \bar{s} \in S \colon \bar{r} \Uparrow \bar{s} \,\big\}$$

and the *left unification anti-semijoin* is

$$R \mathbin{\overline{\ltimes}_{\Uparrow}} S = R - (R \ltimes_{\Uparrow} S) = \big\{\, \bar{r} \in R \mid \nexists\, \bar{s} \in S \colon \bar{r} \Uparrow \bar{s} \,\big\}$$

These are similar to the standard definition of (anti) semijoin; we simply use unifiability of tuples as the join condition. They are definable operations: we have that $R \ltimes_{\Uparrow} S = \pi_R\big(\sigma_{\theta_\Uparrow}(R \times S)\big)$, where the projection is on all attributes of R and condition $\theta_\Uparrow$ is true for a tuple $\bar{r}\bar{s} \in R \times S$ iff $\bar{r} \Uparrow \bar{s}$. The unification condition $\theta_\Uparrow$ is expressible as a selection condition using predicates const and null [25]. Note that, in this notation, $R^{\mathbf{f}}$ of Figure 2 is $\mathsf{adom}^{\mathsf{ar}(R)} \mathbin{\overline{\ltimes}_{\Uparrow}} R$.

We now define the translation $Q \mapsto (Q^+, Q^?)$. For Q^+ with correctness guarantees, all of the rules are the same as in Figure 2, except the one for difference, which becomes

$$(Q_1 - Q_2)^+ \;=\; Q_1^+ \mathbin{\overline{\ltimes}_{\Uparrow}} Q_2^?$$

This is precisely the set of tuples certainly in Q_1 that do not match potential answers to Q_2.

For queries $Q^?$, the translation follows the structure of the query closely, but it needs a different translation of selection conditions: $\theta \mapsto \theta^{**}$ is given by $\theta^{**} = \neg(\neg\theta)^*$. Recall that negating selection conditions means propagating negations through them, and interchanging $=$ and $\neq$, and const and null. For completeness, we provide it below:

$$\begin{aligned}
(A \neq B)^{**} &= (A \neq B)\\
(A \neq c)^{**} &= (A \neq c) \quad \text{if } c \text{ is a constant}\\
(A = B)^{**} &= (A = B) \vee \mathsf{null}(A) \vee \mathsf{null}(B)\\
(A = c)^{**} &= (A = c) \vee \mathsf{null}(A)\\
(\theta_1 \vee \theta_2)^{**} &= \theta_1^{**} \vee \theta_2^{**}\\
(\theta_1 \wedge \theta_2)^{**} &= \theta_1^{**} \wedge \theta_2^{**}
\end{aligned}$$

The full translation is given in Figure 3.

THEOREM 2 ([15]). *For the translation $Q \mapsto (Q^+, Q^?)$ in Figure 3, the query Q^+ has correctness guarantees for Q, and $Q^?$ represents potential answers to Q.*

In particular, $Q^+(D) \subseteq \mathsf{cert}(Q, D)$ and

$$v\big(Q^+(D)\big) \;\subseteq\; Q\big(v(D)\big) \;\subseteq\; v\big(Q^?(D)\big) \tag{5}$$

for every database D and every valuation v.

The theoretical complexity bounds for queries Q^+ and $Q^{\mathbf{t}}$ are the same: both have the low AC^0 data complexity. However, the real world performance of Q^+ will be significantly better, as it completely avoids large Cartesian products.

We conclude this section with a few remarks. First, the translation of Figure 3 is really a family of translations: our result is more general.

COROLLARY 1. *If in the translation in Figure 3 one replaces the right sides of rules by queries*

- *contained in those listed in* (3.1)–(3.6)*, and*
- *containing those listed in* (4.1)–(4.6)*,*

then the resulting translation continues to satisfy the claim of Theorem 2.

This opens up the possibility of optimizing translations (at the expense of potentially returning fewer tuples). For instance, if we modify the translations of selection conditions so that θ^* is a stronger condition than the original and θ^{**} is a weaker one, we retain overall correctness guarantees. In particular, the unification condition $\theta_\Uparrow$ is expressed by a case analysis that may become onerous for tuples with many attributes; the above observation can be used to simplify the case analysis while retaining correctness.

Next, we turn to the comparison of Q^+ with the result of SQL evaluation, i.e., $\mathsf{Eval}_{\mathsf{SQL}}(Q, D)$. Given that the

latter can produce both types of errors – false positives and false negatives – it is not surprising that the two are in general incomparable. To see this, consider first a database D_1 where $R = \{(1,2),(2,\bot)\}$, $S = \{(1,2), (\bot,2)\}$ and $T = \{(1,2)\}$, and the query $Q_1 = R - (S \cap T)$. The tuple $(2,\bot)$ belongs to $\mathsf{Eval}_{\mathsf{SQL}}(Q_1, D)$ and it is a certain answer, while $Q_1^+(D) = \varnothing$. On the other hand, for D_2 with $R = \{(\bot,\bot)\}$ over attributes A, B, and $Q_2 = \sigma_{A=B}(R)$, the tuple $(\bot,\bot)$ belongs to $Q_2^+(D_2)$, but $\mathsf{Eval}_{\mathsf{SQL}}(Q_2, D_2) = \varnothing$.

4. EXPERIMENTAL EVALUATION

We now report on the experiments carried out in [15] that answer two questions posed in the introduction: Do false positives occur in real-life queries? Does the approximation scheme $(Q^+, Q^?)$ perform well in practice?

We have seen that what breaks correctness guarantees is queries with negation; the example in the introduction was based on a `NOT EXISTS` subquery. To choose concrete SQL queries for our experiments, we consider the well established TPC-H benchmark that models a business application scenario and typical decision support queries [30]. Its schema contains information about customers who place orders consisting of several items, and suppliers who supply parts for those orders.

Only few TPC-H queries use `NOT EXISTS`, so we supplement them with very typical database textbook [10] queries (slightly modified to fit the TPC-H schema) that are designed to teach subqueries.

Another issue is that the standard TPC-H data generator, DBGen, only produces instances without nulls, so we need to insert nulls to make them fit for our purpose. To this end, we separate attributes into nullable and non-nullable ones; the latter are those where nulls cannot occur (due to primary key constraints, or `NOT NULL` declarations). For nullable attributes, we choose a probability, referred to as the *null rate* of the resulting instance, and simply flip a coin to decide whether the corresponding value is to be replaced by a null. The resulting instances contain a percentage of nulls in nullable attributes that is roughly equal to the null rate with which nulls are generated. We consider null rates in the range 0.5%–10%.

The smallest instance DBGen generates is about 1GB in size, containing just under $9 \cdot 10^6$ tuples. We measured the relative performance of our translated queries w.r.t. the original ones on instances of size comprised between 1GB and 10GB.

Estimating the amount of false positives in query answers in queries is trickier, since finding certain answers is computationally hard. We overcome this difficulty by using ad hoc algorithms for the specific queries we experiment with, and by using smaller instances generated by a configurable data generator, DataFiller [7]. These instances are compliant with the TPC-H specification in everything but size, which we scale down by a factor of 10^3. For additional details of the experimental setup, we refer to [15].

4.1 How many false positives?

A false positive answer is a tuple that is returned by the SQL evaluation and yet is not certain; that is, the set of false positives produced by a query Q on a database D is $Q(D) - \mathsf{cert}(Q, D)$. They only occur on databases with nulls; on complete databases, $Q(D) = \mathsf{cert}(Q, D)$. A simple example was given in the introduction; our goal now is to see whether real-life queries indeed produce false positives. For this, we shall run our test queries on generated instances with nulls and compare their output with certain answers. As explained above, for each test query we designed a specialized algorithm to detect (some of the) false positives. This will tell us that at least some percentage of SQL answers are false positives.

Recall that null values in instances are randomly generated: each nullable attribute can be null with the same fixed probability, referred to as the null rate. To get good estimates, we generated 100 instances for each null rate in the range 0.5%–10%, and we ran each query 5 times, with randomly generated values for its parameters. At each execution, a lower bound on the percentage of false positives is calculated by means of the algorithms mentioned above.

The outcome of the experiment showed that the problem of incorrect query answers in SQL is not just theoretical but it may well occur in practical settings: every single query we tested produced false positives on incomplete databases with as low as 0.5% of null values. In extreme cases, false positives constitute almost the totality of answers, even when few nulls are present. Other queries appear to be more robust (as we only find a lower bound on the number of false positives), but the overall conclusion is clear: *false positives do occur in answers to very common queries with negation, and account for a significant portion of the answers*.

4.2 The price of correctness

Our goal was to test whether the translation $Q \mapsto Q^+$ works in practice. For this, we executed our test queries and their translations with correctness guarantees on randomly generated incomplete TPC-H instances to compare their performance.

The translation $Q \mapsto Q^+$ was given at the level of relational algebra. While there are multiple relational algebra simulators freely available, we carried out our experiments using a real DBMS on instances of realistic size (which rules out relational algebra simulators). Thus, we took test SQL queries, applied the translation $Q \mapsto Q^+$ to their relational algebra equivalents, and

then ran the results of the translation as SQL queries.

Note that we measured the *relative performance* of the correct translations Q^+s, i.e., the ratio between the running times of Q^+ and of the original queries Q. We used the DBGen tool to generate instances and populated them with nulls, depending on the prescribed null rate. For each null rate in the range 1%–5%, in steps of 1%, we generate multiple incomplete instances, and ran queries multiple times for randomly generated values of their parameters. The reported results were averages of those runs.

Regarding the size of instances, it seems, intuitively, that the *ratio* of execution times of Q^+ and Q should not significantly depend on the size of the generated instances. With this hypothesis in mind, we first did a detailed study for the smallest allowed size of TPC-H instances (roughly 1GB). After that, we tested our hypothesis using instances up to 10GB. For the majority of queries relative performances indeed remained about the same for all instance sizes as we expected, although we did find an exception (we shall discuss this later).

One of the key changes that our translation introduces is to convert conditions of the form `A=B` to

```
A=B OR A IS NULL OR B IS NULL
```

inside correlated **NOT EXISTS** subqueries. The reason for this should be clear when one looks at the translation $\theta \mapsto \theta^{**}$ of conditions in queries $Q^?$. This is the translation that is applied to negated subqueries, due to the rule $(Q_1 - Q_2)^+ = Q_1^+ \,\overline{\ltimes}_{\Uparrow}\, Q_2^?$, thus resulting in such disjunctions.

In general, and this has nothing to do with our translation, when several such disjunctions occur in a subquery, they may not be handled well by the optimizer [5]. One could in fact observe that for a query of the form

```
SELECT * FROM R WHERE NOT EXISTS
  ( SELECT * FROM   S, ..., T
    WHERE  ( A=B OR A IS NULL OR B IS NULL )
             AND ··· AND
           ( X=Y OR X IS NULL OR Y IS NULL ) )
```

the estimated cost of the query plan can be thousands of times higher than for the same query from which the **IS NULL** conditions are removed.

One way to overcome this is quite simple and takes advantage of the fact that such disjunctions will occur inside **NOT EXISTS** subqueries. We can then propagate disjunctions in the subquery, which results in a **NOT EXISTS** condition of the form $\neg\exists\bar{x} \bigvee \phi_i(\bar{x})$, where each ϕ_i now is a conjunction of atoms. This in turn can be split into conjunctions of $\neg\exists\bar{x}\,\phi_i(\bar{x})$, ending up with a query of the form

```
SELECT * FROM R WHERE NOT EXISTS
    ( SELECT * FROM S_i, i ∈ I_1 WHERE ⋀_j ψ_j^1 )
AND ··· AND NOT EXISTS
    ( SELECT * FROM S_i, i ∈ I_k WHERE ⋀_j ψ_j^k )
```

where formulae ψ_j^l are comparisons of attributes and statements that an attribute is or is not null, and relations S_i for $i \in I_l$ are those that contain attributes mentioned in the ψ_j^ls.

Based on the experiments we conduct, we observe three types of behavior, discussed below.

Small overhead. In half of the queries, the price of correctness is *negligible* for most applications, under 4%. The **IS NULL** disjunctions introduced by our translation are well handled by the optimizer, resulting in small overheads. In some cases, these overheads get lower as the null rate gets higher. This is most likely due to the fact that with a higher null rate it is easier to satisfy the **IS NULL** conditions in the **WHERE** clause of the **NOT EXISTS** subquery. As a result, a counterexample to the **NOT EXISTS** subquery can be found earlier, resulting in an overall faster evaluation.

Significant speedup. The translation with correctness guarantees is *much faster* than the original query; in fact we observed that it could be more than 3 orders of magnitude faster on average. This behavior arises when the translation with correctness guarantees results in decorrelated subqueries, which allows one to quickly detect that the correct answer is empty and terminate execution early, while the original query, on the other hand, spends most of its time looking for incorrect answers. In fact, this behavior was observed for queries with a rate of false positive answers close to 100%. As instances grow larger, the speedup of the translated query increases, since the original query is forced to spend more time looking for incorrect answers.

Moderate slowdown. The translated queries with correctness guarantees run at roughly half the speed of the original ones on 1GB databases. The slowdown is worse for bigger instances, increasing to about a quarter of the speed on 10GB databases, but it may still be tolerable if correctness of results is very important.

This behavior may arise when there are complex multi-way joins with large tables in **NOT EXISTS** subqueries. Without splitting the **IS NULL** disjunctions introduced by our translation, PostgreSQL produces query plans with astronomical costs, as it resorts to nested-loop joins even for large tables. This is due to the fact that it underestimates the size of joins, which is a known issue for major DBMSs [21]. In order to make the optimizer produce better estimates and a reasonable query plan, the direct translation of these queries may also require some additional hand-tuning involving common table expressions.

We conclude our experimental evaluation by addressing the standard measures for assessing the quality of approximation algorithms, namely precision and recall.

The first refers to the percentage of correct answers given. With the correctness guarantees proven in Section 3, we can state that the precision of our algorithms is 100%. Recall refers to the fraction of relevant answers returned. In our case, we can look at the certain answers returned by the standard SQL evaluation of a query Q, and see how many of them are returned by Q^+. The ratio of those is what we mean by recall in this scenario.

In some artificial examples, Q^+ may miss several, or even all, certain answers returned by Q. Thus, we cannot state a theoretical bound on the recall, but we can see what it is in the scenarios represented by our test queries. For this, we could use algorithms for identifying false positives, as explained in Section 4.1, on smaller TPC-H instances generated by DataFiller. In all those cases, the behavior we observed was that the translated queries returned precisely the answers to the original queries except false positive tuples. That is, for those instances, the recall rate was 100%, and no certain answers were missed.

5. THEORETICAL MODELS VS. REAL LIFE

We saw that good theoretical complexity bounds do not guarantee efficiency in real systems: the evaluation schemes with correctness guarantees presented in Section 3 are both very efficient in theory, yet only one of them performs well in practice. The mismatch between theoretical results and their practicality is not limited to efficiency. Before our approach [15] could be successfully applied in real life scenarios, several other important factors must be taken into account. We discuss them below.

5.1 Bag semantics and certain answers

As prescribed by the SQL Standard, relational database management systems use bag semantics in query evaluation. With bags, a tuple $\bar{a}$ can have a *multiplicity* (number of occurrences) $\#(\bar{a}, R)$ in a table R, which is a number in $\mathbb{N}$. Thus, instead of saying that a tuple is certainly in the answer, we have more detailed information: namely, the range of the numbers of occurrences of the tuple in query answers. This is captured by the following definitions, that extend the notion of certain answers with nulls:

$$\min_Q(D, \bar{a}) = \min_v \#\big(v(\bar{a}), Q(v(D))\big) \quad (6a)$$

$$\max_Q(D, \bar{a}) = \max_v \#\big(v(\bar{a}), Q(v(D))\big) \quad (6b)$$

where v ranges over valuations. Note that, if $\bar{a}$ has no nulls, $\min_Q(D, \bar{a})$ and $\max_Q(D, \bar{a})$ are simply the minimum and the maximum numbers of occurrences of $\bar{a}$ in the answer to Q over all databases $v(D)$ represented by D. Then we know with certainty that every query answer must contain at least $\min_Q(\bar{a}, D)$ occurrences of $\bar{a}$, and no answer will contain more than $\max_Q(\bar{a}, D)$ of them. When a query is evaluated under set semantics, $\min_Q(D, \bar{a}) = 1$ means that $\bar{a} \in \mathsf{cert}(Q, D)$.

Relational algebra operations under bag semantics are interpreted in a way that is consistent with SQL evaluation: union, for example, adds up occurrences and, for difference, $\#(\bar{a}, R-S) = \max\big(\#(\bar{a}, R)-\#(\bar{a}, S), 0\big)$. We refer to [14] for a survey on the subject and the full definition of all operations of relational algebra $(\sigma, \pi, \times, \cup, -)$ under bag semantics.

The complexity of the bounds (6a) and (6b) mimics analogous results for set semantics: for every relational algebra query Q interpreted under bag semantics, and for every $m \in \mathbb{N}$, checking whether $\min_Q(D, \bar{c}) > m$ or whether $\max_Q(D, \bar{c}) < m$ can be done in coNP with respect to data complexity, and the problems could be coNP-hard already without duplicates.

The difference with the set case comes when we look at *positive* relational algebra which, as before, excludes difference.[1]

THEOREM 3 ([8]). *For each positive relational algebra query Q, under bag semantics, $\min_Q(D, \bar{c})$ can be computed in polynomial time (in fact,* DLOGSPACE*) with respect to data complexity.*

However, there is a positive relational algebra query Q such that checking, for given D, $\bar{a}$, and m, whether $\max_Q(D, \bar{a}) < m$ is coNP*-complete.*

In fact, coNP-hardness is witnessed by an extremely simple query that returns a relation in a database, that is, `SELECT * FROM R`.

Next, we look at possible extensions of the approximation schemes of Section 3 to bag semantics. A simple analysis of the definition of queries $Q^{\mathbf{t}}, Q^{\mathbf{f}}$ shows that for every tuple $\bar{a}$,

$$\begin{aligned} \#\big(\bar{a}, Q^{\mathbf{t}}(D)\big) &\leq \min_Q(D, \bar{a}) \\ \#\big(\bar{a}, Q^{\mathbf{f}}(D)\big) &\leq \big(1 + \max_Q(D, \bar{a})\big) \mod 2 \end{aligned}$$

This suggests a natural extension of the translation scheme $(Q^{\mathbf{t}}, Q^{\mathbf{f}})$ to bags: we simply omit modulo 2 from addition, since it was only needed to force multiplicities to be either 0 or 1. But this is suddenly very problematic, as $\max_Q(D, \bar{a})$ is hard computationally, for *all* queries, since we cannot even compute it efficiently for base relations! Thus, implementing this approximation scheme in a real-life RDBMS (which is bag-based) is infeasible not only practically but also *theoretically* when we use bag semantics.

[1] Please note that Theorems 3 and 4, as stated in [8], referred to languages that also erroneously included duplicate elimination. However, the claims hold only when this operation is not part of the language.

On the other hand, (5) suggests a natural extension of the correctness criterion for the translation scheme $(Q^+, Q^?)$, namely:

$$\#(\bar{a}, Q^+(D)) \;\leq\; \min_Q(\bar{a}, D) \leq\; \#(\bar{a}, Q^?(D)) \quad (7)$$

for every database D and every tuple $\bar{a}$ of elements of D. Indeed, for bags B_1 and B_2, we have that $B_1 \subseteq B_2$ iff $\#(b, B_1) \leq \#(b, B_2)$ for every element b.

THEOREM 4 ([8]). *The translation $Q \mapsto (Q^+, Q^?)$ in Figure 3 satisfies (7) when queries are interpreted under bag semantics.*

In summary, the translation of Figure 2 loses its good theoretical complexity bounds and becomes intractable under bag semantics, while the approximation scheme of Figure 3 remains provably feasible also under bag semantics, thus strengthening the claim of its efficiency, practicality, and robustness.

5.2 Relational algebra vs SQL

The translations [15] in Section 3 work at the level of relational algebra, while the experimental evaluation in Section 4 was carried out with concrete SQL queries on a real DBMS. This was achieved by first translating an SQL query Q to relational algebra, applying the translation with correctness guarantees, and then translating the resulting RA query Q^+ back to SQL.

Unfortunately, database textbooks provide only a few examples of translations between SQL and RA, and detailed translations that appeared in the literature made simplifying assumptions that deviate significantly from the behavior of SQL specified by the Standard, such as the use of set semantics and the omission of nulls along with the associated three-valued logic.

Recently, [16] proposed a formal semantics of SQL that captures the core of the real language and that was experimentally validated on a very large number of randomly generated queries and databases. The semantics was applied to provide precise translations between the core fragment of SQL and relational algebra, yielding the first formal proof that they have the same expressive power. Using this formal semantics, [16] also showed that the three-valued logic of SQL is not really necessary for query evaluation, despite what is commonly believed, and that the usual Boolean logic with only true and false suffices.

The test queries we used in Section 4 go slightly beyond relational algebra as used in the translations of Figure 3. Given their decision support nature, many TPC-H queries involve aggregation, but this is not important for our purposes: if a tuple without an aggregate value is a false positive, it remains so even when an extra attribute value is added. Thus, since we only need to measure the ratio of false positives, and the *relative* change of speed in query evaluation, we can safely drop aggregates from the output of those queries. As for aggregate subqueries, we just treated them as a black box, that is, we viewed the result of such a subquery as a constant value c.

5.3 Marked nulls vs SQL nulls

The approximation schemes of [25] and [15] rely on the standard theoretical model of incompleteness where missing values in a database are represented by marked nulls. In SQL, however, we only have a single syntactic object for this purpose: **NULL**. Marked nulls are more expressive than SQL nulls, in that two unknown values can be asserted to be the same simply by denoting them with the same null. Indeed, $\bot_1 = \bot_1$ is *true* independently of which concrete value is assigned to $\bot_1$. On the other hand, the comparison **NULL** = **NULL** in SQL evaluates to *unknown*, because we do not know whether the two occurrences of **NULL** refer to the same value.

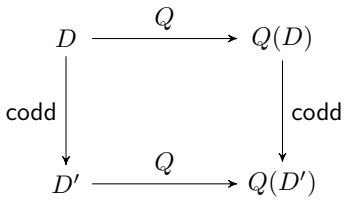
Due to the coarseness of SQL nulls, the translations Q^+ and $Q^?$ must be slightly adjusted to work correctly when evaluated as SQL queries. As expected, the adjustment occurs in selection conditions. For the θ^* translation in Q^+, we need to make sure that attributes compared for equality are not nulls (the existing translation already does that for disequality). For the θ^{**} translation in $Q^?$, the situation is symmetric: we need to include the possibility of attributes being nulls for disequality comparisons (the existing translation already does that for equality). That is, we change the translations as follows:

$$\begin{aligned}(A = B)^* &= (A = B) \wedge \mathsf{const}(A) \wedge \mathsf{const}(B)\\ (A \neq B)^{**} &= (A \neq B) \vee \mathsf{null}(A) \vee \mathsf{null}(B)\end{aligned}$$

and likewise for $(A = c)^*$ and $(A \neq c)^{**}$. Observe that, as stronger conditions are used for equality rules in θ^* and weaker ones for disequality rules in θ^{**}, by Corollary 1 the adjusted translations ensure that Q^+ continues to underapproximate certain answers and $Q^?$ continues to represent potential answers on databases with marked nulls, but now we also take into account SQL's behavior in comparisons with nulls.

There is one more issue we need to address. Usually, at least in the theoretical literature, SQL nulls are identified with *Codd nulls*, that is, marked nulls that do not repeat. The idea is to interpret each occurrence of **NULL** as a fresh marked null that does not appear anywhere else in the database. However, [17] recently showed that this way of modeling SQL nulls does not always work. If SQL nulls are to be interpreted as Codd nulls, this interpretation should apply to input databases as well as query answers, which are incomplete databases themselves. To explain this point, let $\mathsf{codd}(D)$ be the result of replacing SQL nulls in D with distinct marked nulls; as this choice is arbitrary, technically $\mathsf{codd}(D)$ is a set of databases, but these are all isomorphic. To ensure that

Codd nulls faithfully represent SQL nulls for a query Q, we need to enforce the condition in the diagram below:



Intuitively, it says the following: take an SQL database D, and compute the answer to Q on it, i.e., $Q(D)$. Now take some D' in $\mathsf{codd}(D)$, and compute $Q(D')$. Then $Q(D')$ must be in $\mathsf{codd}\big(Q(D)\big)$, that is, there must be a way of assigning Codd nulls to SQL nulls in $Q(D)$ that will result in $Q(D')$.

Unfortunately, [17] showed that this condition does not hold already for simple queries computing the Cartesian product of two relations. Furthermore, the class of relational algebra queries that transform SQL databases into Codd databases is not recursively enumerable, and therefore it is impossible to capture it by a syntactic fragment of the language. Exploiting **NOT NULL** constraints declared on the schema, [17] then proposes mild syntactic restrictions on queries that can be checked efficiently and are sufficient to guarantee the condition in the above diagram (i.e., that SQL nulls behave like Codd nulls).

We remark that the queries – and their translations – used for the experimental evaluation in Section 4 satisfy these restrictions and therefore they work correctly with the SQL implementation of nulls. However, the results of [17] tell us that in full generality we cannot guarantee correctness for all queries unless a proper implementation of marked nulls is available.

6. OUTLOOK & OPEN PROBLEMS

The main conclusion is that it is practically feasible to modify SQL query evaluation over databases with nulls to guarantee correctness of its results. This applies to the setting where nulls mean that a value is missing, and the fragment of SQL corresponds to first-order, or relational algebra, queries. We saw that the modified queries with correctness guarantees run at roughly a quarter of the speed in the worst case, to almost 10^4 times faster in the best case. For several queries, the overhead was small and completely tolerable, under 4%. With these translations, we also did not miss any of the correct answers that the standard SQL evaluation returned.

Given our conclusions that wrong answers to SQL queries in the presence of nulls are not just a theoretical myth – there are real world scenarios where this happens – and correctness can be restored with syntactic changes to queries at a price that is often tolerable, it is natural to look into the next steps that will lift our solution from the first-order fragment of SQL to cover more queries and more possible interpretations of incompleteness. We shall now discuss those.

Aggregate functions. An important feature of real-life queries is aggregation which, in fact, is present in most of the TPC-H queries. However, here our understanding of correctness of answers is quite poor; SQL's rules for aggregation and nulls are rather ad-hoc and have been persistently criticized [4, 9]. Therefore, much theoretical work is needed in this direction before practical algorithms emerge.

Incorporating constraints. In the definition of certain answers we disregarded constraints, even though every real-life database will satisfy some, typically keys and foreign keys. While a constraint ψ can be incorporated into a query ϕ by finding certain answers to $\psi \to \phi$, for common classes of constraints we would like to see how to make direct adjustments to rewritings. One example of this that we actually used in query rewriting is that the presence of a key constraint let us replace $R \,\overline{\ltimes}_{\Uparrow}\, S$ by $R - S$. Ideally such query transformations need to be automated for common classes of constraints.

Other types of incomplete information. We dealt with nulls representing missing values, but there are other interpretations. For instance, non-applicable nulls [23, 34] arise commonly as the result of outer joins. We need to extend the notion of correct query answering and translations of queries to them. One possibility is to adapt the approach of [24] that shows how to define certainty based on the semantics of inputs and outputs of queries. At the level of missing information, we would like to see whether our translations could help with deriving partial answers to SQL queries, when parts of a database are missing, as in [20].

Direct SQL rewriting. We have rewritten SQL queries by a detour via relational algebra. With the assistance of the formal semantics of [16], we should look into direct rewritings from SQL to SQL, without an intermediate language. This would also allow us to run queries with correctness guarantees directly on a DBMS.

Marked nulls in SQL. The results of [17] show us that with standard SQL nulls we can only guarantee correctness for a restricted class of queries, which cannot even be captured syntactically. To overcome this limitation, we are currently working towards extending SQL with a proper implementation of marked nulls.

Acknowledgments

This survey is based on the work originally published in [15, 25], which greatly benefited from discussions with

Marco Console, Chris Date, Hugh Darwen, Ron Fagin, Chris Ré, and Cristina Sirangelo. Work partly supported by EPSRC grants N023056 and M025268.

7. REFERENCES

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

[2] S. Abiteboul, P. C. 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] J. Celko. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 1995.

[5] J. Claußen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 12(2):238–260, 2000.

[6] E. F. Codd and C. J. Date. Much ado about nothing. In C. J. Date, editor, *Relational Database Writings 1991–1994*. 1995.

[7] F. Coelho. DataFiller – generate random data from database schema. `https://www.cri.ensmp.fr/people/coelho/datafiller.html`.

[8] M. Console, P. Guagliardo, and L. Libkin. On querying incomplete information in databases under bag semantics. In *IJCAI*, pages 993–999. ijcai.org, 2017.

[9] C. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.

[10] C. J. Date. *An Introduction to Database Systems*. Pearson, 2003.

[11] G. H. Gessert. Four valued logic for relational database systems. *SIGMOD Record*, 19(1):29–35, 1990.

[12] A. Gheerbrant, L. Libkin, and C. Sirangelo. Naïve evaluation of queries over incomplete databases. *ACM Trans. Database Syst.*, 39(4):31:1–31:42, 2014.

[13] J. Grant. Null values in a relational data base. *Inf. Process. Lett.*, 6(5):156–157, 1977.

[14] S. Grumbach, L. Libkin, T. Milo, and L. Wong. Query languages for bags: expressive power and complexity. *SIGACT News*, 27(2):30–44, 1996.

[15] P. Guagliardo and L. Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *PODS*, pages 211–223. ACM, 2016.

[16] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1), 2017.

[17] P. Guagliardo and L. Libkin. On the Codd semantics of SQL nulls. In *AMW*, 2017.

[18] T. Imielinski and W. Lipski. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[19] H. Klein. How to modify SQL queries in order to guarantee sure answers. *SIGMOD Record*, 23(3):14–20, 1994.

[20] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *SIGMOD*, pages 1275–1286, 2014.

[21] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[22] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[23] N. Lerat and W. Lipski. Nonapplicable nulls. *Theor. Comput. Sci.*, 46(3):67–82, 1986.

[24] L. Libkin. Certain answers as objects and knowledge. *Artificial Intelligence*, 232:1–19, 2016.

[25] L. Libkin. SQL's three-valued logic and certain answers. *ACM TODS*, 41(1):1:1–1:28, 2016.

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

[27] W. Lipski. On relational algebra with marked nulls. In *PODS*, pages 201–203, 1984.

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

[29] 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.

[30] Transaction Processing Performance Council. *TPC Benchmark™ H Standard Specification*, Nov. 2014. Revision 2.17.1.

[31] R. van der Meyden. Logical approaches to incomplete information: A survey. In *Logics for Databases and Information Systems*, pages 307–356, 1998.

[32] M. Vardi. Querying logical databases. *Journal of Computer and System Sciences*, 33(2):142–160, 1986.

[33] K. Yue. A more general model for handling missing information in relational databases using a 3-valued logic. *SIGMOD Record*, 20(3):43–49, 1991.

[34] C. Zaniolo. Database relations with null values. *J. Comput. Syst. Sci.*, 28(1):142–166, 1984.