Making SQL Queries Correct on Incomplete Databases: A Feasibility Study

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 key goal, evaluation of SQL queries disregards the standard notion of correctness on incomplete databases – certain answers – due to its high complexity. As a result, it may produce answers that are just plain wrong. It was recently shown that SQL evaluation can be modified, at least for first-order queries, to return only correct answers. But while these modifications came with good theoretical complexity bounds, they have not been tested in practice.

The goals of this proof-of-concept paper are to understand whether wrong answers can be produced by SQL queries in real-world scenarios, and whether proposed techniques for avoiding them can be made practically feasible.

We use the TPC-H benchmark, and show that for some typical queries involving negation, wrong answers are very common. On the other hand, existing solutions for fixing the problem do not work in practice at all. By analyzing the reasons for this, we come up with a new modified way of rewriting SQL queries that restores correctness. We conduct experiments which show the feasibility of our solution: the small price tag it imposes can be often tolerated to ensure correct results, and we do not miss correct answers that the usual SQL evaluation produces. The overall conclusion is that correct evaluation can be realistically achieved in the presence of nulls, at least for the SQL fragment that corresponds to first-order queries.

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 [7, 9]. The reason behind this behavior is that SQL designers had first and foremost efficient evaluation in mind, but correctness and efficiency do not always get along. The standard theoretical approach to answering queries on incomplete databases, which is widely accepted as providing the right notion of answers, is to compute *certain answers*. These are query answers that do not depend on how the unknown data is interpreted. However, computing them is not easy, in fact coNP-hard for most reasonable semantics, if we deal with relational calculus/algebra queries [2]. SQL's evaluation is very efficient; it is in AC^0 (a small parallel complexity class) for the same class of queries, and thus it provably cannot capture certain answers.

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-4191-2/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2902251.2902297

The gap in complexity is not yet a reason for undesirable behavior; one can easily imagine that SQL evaluation produces an approximation of certain answers. There are two ways in which certain answers and SQL's evaluation could differ:

- 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*.

False negatives can be accepted as the price to be paid for lowering complexity; after all, they just indicate that the correct answer is approximated. False positives, on the other hand, are outright wrong answers and therefore should not be tolerated. The problem with SQL is that it produces both kinds of errors.

To see how false positives can be generated, consider a simple query computing the difference of two relations R and S, each with a single attribute A:

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

When $R = \{1\}$ and $S = \{\texttt{NULL}\}$, the output is $\{1\}$, but it is *not* a certain answer. Indeed, if the missing value represented by **NULL** is interpreted as 1, the difference $R - S$ is empty, and thus so is the set of certain answers.

The reasons behind SQL's incorrect behavior have their roots in the flawed three-valued logic approach it uses for handling nulls. Multiple attempts to fix it have been made in the past (see, e.g., [11, 15, 31]) although none of them came with formal correctness guarantees. Recently, [22] proposed a new approach to fixing SQL's evaluation scheme that provided provable correctness guarantees. It showed how to translate a query Q into a query Q' such that:

- false positives never occur: Q' returns a subset of certain answers to Q;
- data complexity of Q' is still AC^0; and
- on databases without nulls, Q and Q' produce the same results.

Given the attractive theoretical properties of the approach, we want to understand *whether these theoretical* *guarantees can work in practice.* To this end, we need to answer two main questions:

Question 1. Are false positives a real problem? Do they occur in real-life queries over databases with nulls?

Question 2. Can algorithms that correctly evaluate queries on databases with nulls be efficiently implemented? What is the price to pay, in terms of query evaluation performance, for correctness guarantees?

Since algorithms in [22] introduced extra steps to restore correctness, we do not expect, in general, to outperform native SQL evaluation, which was designed exclusively 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 is then the *price of correctness*, and we need to understand what it is.

This work is the first feasibility study in this direction, and it produces promising results that warrant further investment into designing algorithms with correctness guarantees for larger classes of queries. Here we consider relational calculus/algebra queries and provide the following results.

False positives. They are a real problem for queries that involve negation. We look at queries inspired by the TPC-H benchmark [28] and show that false positives are always present. Sometimes they constitute almost 100% of answers; for other queries, they quickly grow with the null rate (the probability that a null occurs in a given position), accounting for between 1.5% and 20% of answers when the null rate is 2% and rising to at least 15% even for the best behaved queries when the null rate is 10%.

Algorithms with correctness guarantees. Those presented in [22] have a very good theoretical complexity but they cannot be implemented *as-is* due to the extensive use of Cartesian product in translated queries. To overcome this, we design an alternative way of transforming queries that still guarantees correctness while producing queries that are much more implementation-friendly. For positive queries (i.e., not involving negation in any form) and on databases without nulls, it coincides with the usual SQL evaluation.

We then test our new algorithm on databases with nulls, for our sample TCP-H queries with negation, and observe two kinds of behavior. Sometimes, the performance of translated queries with correctness guarantees is very good, with the price of correctness – the overhead of executing the translated query – not exceeding 4% (and sometimes in fact making queries up to 10,000 times faster).

For other queries, the translation may "confuse" the optimizer. That is, the optimizer generates plans with astronomical costs (although in some cases these are not reflected by the actual running time). The reason is that some conditions of the form `R.A = S.B` get translated into `R.A = S.B` **`OR`** `R.A` **`IS NULL`** to enforce correctness, but this causes the optimizer to badly overestimate the execution time, up to the point of resorting to nested-loop implementation of joins. However, we show that simple syntactic manipulations of queries can restore sanity to the optimizer, and result in reasonable performance, with translated queries running at roughly half the speed of the original ones.

Precision and recall. We need to address them to say how good our evaluation techniques are. Precision refers to how many answers are certain, and recall to the proportion of certain answers returned. The formal correctness guarantees we prove imply that precision of our algorithms is 100% (while for some of the queries we experiment with, SQL's precision is close to zero).

It is easy to achieve high precision with low recall: just return nothing, and there are no false positives. Since SQL returns many more answers than necessary (false positives), it should not be surprising that sometimes it also returns a certain answer our procedure misses. However, in all our experiments, recall stands at 100%: our procedure returns precisely certain answers that are also returned by SQL evaluation.

The key conclusion of this study is that *achieving correctness in SQL query evaluation on databases with nulls is feasible.* As we said, this is a proof-of-concept paper, whose goal is to demonstrate that the idea of rewriting queries to achieve correctness is implementable. We have done this for first-order queries and missing-information interpretation of nulls. Of course much more is needed before one can introduce a second, fully correct, evaluation mode for SQL queries (e.g., by saying **`SELECT CERTAIN`**); we shall discuss particular tasks (such as handling bag semantics, aggregates, non-applicable nulls, etc.) at the end of the paper.

Organization. Basic notions and definitions are given in Section 2. Section 3 discusses the setup for our experiments. Section 4 presents experimental results on false positives in SQL query evaluation. Section 5 describes the translation of [22] and explains why it cannot be efficiently implemented. Section 6 presents our improved implementation-friendly translation. Section 7 contains the experimental results for the improved translation and describes the price of correctness for SQL query evaluation. Section 8 discusses extensions to cover other language aspects of SQL. The query translations used in our experiments are in the appendix.

2. PRELIMINARIES

We consider incomplete databases with nulls interpreted as missing information. Much of the following is standard in the literature on databases with incomplete information, see, e.g., [1, 12, 14, 22, 30]. The usual way of modeling SQL's nulls under this interpretation is to use *Codd nulls* which are a special case of the more general *marked*, or labeled, nulls. 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 $\bot$, 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 (and our experiments), 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, or vocabulary, is a set of relation names with associated arities. 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.

In instances with Codd nulls, it is assumed that nulls do not repeat, i.e., each element of Null appears at most once in a database. When repetition of nulls is allowed, we speak of *labeled* or *marked* nulls; these often appear in applications such as data integration and exchange [4, 19]. Our translations work correctly on databases with Codd nulls as well as on databases with the more general marked nulls.

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, i.e., $\mathsf{Const}(D) \cup \mathsf{Null}(D)$. 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 $\bot$ with $v(\bot)$ in D. The *semantics* of an incomplete database D is the set of all complete databases it can possibly represent. Under the missing value interpretation of nulls, it is defined as $\{v(D) \mid v \text{ is a valuation}\}$, and is referred to as the closed-world semantics of incompleteness [26].

Query languages. As our query language, we consider the basic fragment of SQL, corresponding to relational calculus/algebra (i.e., first-order queries). That is, we have the usual **`SELECT-FROM-WHERE`** queries, with (correlated) subqueries preceded by **`IN`** and **`EXISTS`**, as well as their negations.

For translations that give us correctness guarantees, we use relational algebra, with the standard operations of selection σ, projection π, Cartesian product $\times$ (or join $\bowtie$), union $\cup$, difference $-$ and intersection $\cap$. 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 $(A \neq B) \wedge (B = 1)$.

In the translations of queries, 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 SQL's `A` **`IS NOT NULL`** and `A` **`IS NULL`**.

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, these are typically defined as $\bigcap\{Q\big(v(D)\big) \mid v \text{ is a valuation}\}$; see [1, 14].

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 $R = \{(1, \bot), (2, 3)\}$, and a query returning R, then the certain answer contains only $(2, 3)$, while intuitively it should give us the entire relation. In light of this, for correctness guarantees we use a closely-related but more general notion from [23], called certain answers with nulls in [22].

Formally, for a query Q and a database D, the *certain answer with nulls*, denoted by $\mathsf{cert}(Q, D)$, is the set of all tuples $\bar{a}$ over $\mathsf{adom}(D)$ such that $v(\bar{a}) \in Q\big(v(D)\big)$ for every valuation v on D. In the above example, certain answer with nulls contains both tuples $(1, \bot)$ and $(2, 3)$. The standard certain answers are exactly the null-free tuples in $\mathsf{cert}(Q, D)$ [22].

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 Q into another query Q', and then run it on D. If $Q'(D) \subseteq \mathsf{cert}(Q, D)$, 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.

FACT 1 ([12, 14, 22]). *Naïve evaluation has correctness guarantees for positive relational algebra, i.e., relational algebra without the difference operator and without disequalities in selection conditions. In fact it computes exactly certain answers with nulls. 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. For SQL, the evaluation procedure is different, as it is based on a 3-valued logic (3VL); see [9]. In particular, comparisons such as $\bot = c$, as well as comparisons between two nulls, evaluate to *unknown*, which is then propagated through conditions using the rules of 3VL.

More precisely, selection conditions can evaluate to true ($\mathbf{t}$), false ($\mathbf{f}$), or unknown ($\mathbf{u}$). If at least one attribute in a comparison is null, the result of the comparison is $\mathbf{u}$. The interaction of $\mathbf{u}$ with Boolean connectives is as follows: $\neg\mathbf{u} = \mathbf{u}$, $\mathbf{u} \wedge \mathbf{t} = \mathbf{u} \wedge \mathbf{u} = \mathbf{u}$, $\mathbf{u} \wedge \mathbf{f} = \mathbf{f}$, and dually by De Morgan's law for $\vee$. Then, σ_θ selects tuples on which θ evaluates to $\mathbf{t}$ (that is, $\mathbf{f}$ and $\mathbf{u}$ 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 ([22]). $\mathsf{Eval}_{\mathsf{SQL}}$ *has correctness guarantees for the positive fragment of relational algebra.*

The positive fragment of relational algebra 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.

3. SETUP: QUERIES AND INSTANCES

We have seen that what breaks correctness guarantees is queries with negation; the example used in the introduction was based on a **`NOT EXISTS`** subquery. To choose concrete queries for our experiments, we use the well established and common TPC-H benchmark that models a business application scenario and typical decision support queries [28]. Its schema contains information about customers who place orders consisting of several items, and suppliers who supply parts for those orders. There are also small relations describing geographical information (nations and regions). In terms of size, `lineitem` is by far the largest table, which records the items constituting an order and associated parts and suppliers, followed by `order` itself.

Given the decision support nature of TPC-H queries, many of them involve aggregation. However, aggregation is not important for our purposes: if a tuple without an aggregate value is a false positive, it remains false positive when an extra attribute value is added. 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.

Most of the TPC-H queries do not have negation; they are essentially aggregate queries on top of multi-way joins. Two of them, queries 21 and 22, do use **NOT EXISTS** so we choose them for our experiments. We further supplement them with two very typical database textbook [10] queries (slightly modified to fit the TPC-H schema) that are designed to teach subqueries. We provide all these queries below, to give the reader an idea of the features involved.

Query Q_1. It is a TPC-H query meant to identify suppliers who were not able to ship required parts in a timely manner. It returns suppliers from a given nation, and multi-supplier *finalized* orders (i.e., with status 'F') where the supplier was the only one who failed to meet the committed delivery date. The query is:

```
SELECT s_suppkey, o_orderkey
FROM   supplier, lineitem l1, orders, nation
WHERE  s_suppkey = l1.l_suppkey
  AND  o_orderkey = l1.l_orderkey
  AND  o_orderstatus = 'F'
  AND  l1.l_receiptdate > l1.l_commitdate
  AND  EXISTS (
       SELECT *
       FROM   lineitem l2
       WHERE  l2.l_orderkey = l1.l_orderkey
         AND  l2.l_suppkey <> l1.l_suppkey )
  AND  NOT EXISTS (
       SELECT *
       FROM   lineitem l3
       WHERE  l3.l_orderkey = l1.l_orderkey
         AND  l3.l_suppkey <> l1.l_suppkey
         AND  l3.l_receiptdate > l3.l_commitdate )
  AND  s_nationkey = n_nationkey
  AND  n_name = $nation
```

where `$nation` is a randomly chosen value among the keys of table `nation`.

Query Q_2. It is another TPC-H query, which aims at identifying countries where there are customers who may be likely to make a purchase. It returns customers within a specific range of countries who have not recently placed orders but have a greater than average positive account balance. The query is:

```
SELECT c_custkey, c_nationkey
FROM   customer
WHERE  c_nationkey IN ($countries)
  AND  c_acctbal > (
       SELECT avg(c_acctbal)
       FROM   customer
       WHERE  c_acctbal > 0.00
         AND  c_nationkey IN ($countries) )
  AND  NOT EXISTS (
       SELECT *
       FROM   orders
       WHERE  o_custkey = c_custkey )
```

where `$countries` is a list of 7 distinct values randomly chosen among the keys of table `nation`.

Query Q_3. It is a classical textbook query that finds all orders supplied entirely by a specific supplier:

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT *
       FROM   lineitem
       WHERE  l_orderkey = o_orderkey
         AND  l_suppkey <> $supp_key )
```

where `$supp_key` is a randomly chosen value among the keys of table `supplier`.

Query Q_4. It is another standard textbook query illustrating a correlated subquery with **NOT EXISTS**. The subquery uses multiple relations and a complex join condition. It asks for orders not supplied with any part of a specific color by any supplier from a specific country.

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT *
       FROM   lineitem, part, supplier, nation
       WHERE  l_orderkey = o_orderkey
         AND  l_partkey = p_partkey
         AND  l_suppkey = s_suppkey
         AND  p_name LIKE '%'||$color||'%'
         AND  s_nationkey = n_nationkey
         AND  n_name = $nation )
```

Here `$nation` is a randomly chosen value among the keys of table `nation` and `$color` is a randomly chosen string from a list of 92 possibilities provided by TPC-H.

Generating test instances. The TPC-H benchmark comes with its own standard tool, DBGen, for generating instances. However, DBGen generates only *complete* instances (i.e., without nulls) so we need to go over them and insert nulls to make them fit for our purpose.

For that, we separate attribute 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 for each tuple to decide whether the corresponding value is to be replaced by a null. As a result, for each nullable attribute, the instance will contain a percentage of nulls 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 (in line with the prescriptions of the TPC-H standard) is about 1GB in size, containing just under $9 \cdot 10^6$ tuples. We shall measure the relative performance of our translated queries w.r.t. the original ones on instances of 1GB, 3GB, 6GB, and 10GB size.

To estimate the amount of false positives in query answers, we shall generate a high number of incomplete instances, on which our sample queries are executed multiple times. False positives are detected by algorithms that are quite expensive. To speed up the process, and since in any case we are interested only in the *percentage* of false positives, for this experiment we use smaller instances. These are generated by means of a configurable data generator, DataFiller [8], and are compliant with the TPC-H specification in everything but size, which we scale down by a factor of 10^3.

All our experiments were run using a local installation of PostgreSQL 9.5.1 on a dedicated machine with an Intel Core i5-3470 quad-core CPU @ 3.20GHz and 8GB of RAM. Note that since we measure the *percentage* of false positive answers and the *relative* performance of our scheme for obtaining correct answers, the exact hardware configuration and choice of a DBMS are of less importance.

4. HOW MANY FALSE POSITIVES?

A false positive answer is a tuple that is returned by 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.

However, certain answers are expensive to compute: the problem is coNP-hard for queries with negation, thus a naïve computation will require exponential time. As a way around it, we design specialized algorithms to detect (some of the) false positives for queries Q_1–Q_4, and compare their results with SQL outputs. This will tell us that at least some percentage of SQL answers are false positives.

Algorithms for detecting false positives. Such algorithms take as input the bindings for the parameters of the query, a database D and an answer tuple $\bar{a}$, and return *true* if $\bar{a}$ is a false positive, thereby giving us a lower bound on the number of false positives. The underlying idea is the same for all queries: we look for the presence of null values in relevant comparisons involving nullable attributes, because in such a case the comparison can be made true or false at need in order to falsify the answer tuple. For instance, to falsify an order id k in the answer to query Q_3, we look for a tuple in table `lineitem` where the value of attribute `l_orderkey` is k and the value of `l_suppkey` is null. Intuitively, if there is a lineitem for order k where the supplier is unknown, then that supplier may well be different from the one specified by the value of parameter `$supp_key`.

Detecting false positives in query Q_2 is also simple: it suffices to check whether there is a tuple in table `orders` for which the attribute `o_custkey` is null; in that case, all answers produced by the query are false positives. Intuitively, if there is an order for which the customer is unknown, then that customer could be anybody, including the one in the answer tuple.

For the remaining two sample queries, the pseudocode to detect false positives is given in Algorithms 1 and 2.

Algorithm 1 Detect false positives in Q_1

```
for $\bar{t} \in$ lineitem where $\bar{t}$[l_orderkey] = $\bar{a}$[o_orderkey]:
    $x \leftarrow \bar{t}$[l_suppkey]
    if $x$ is not null and $x = \bar{a}$[s_suppkey]:
        continue
    $d_1 \leftarrow \bar{t}$[l_commitdate] ; $d_2 \leftarrow \bar{t}$[l_receiptdate]
    if $d_1$ is null or $d_2$ is null or $d_2 > d_1$:
        return true
return false
```

Algorithm 2 Detect false positives in Q_4

```
for $\bar{t} \in$ lineitem where $\bar{t}$[l_orderkey] = $\bar{a}$[o_orderkey]:
    $P, S \leftarrow$ false
    for $\bar{p} \in$ part where $\bar{t}$[l_partkey] is null or
                                or equal to $\bar{p}$[p_partkey]:
        if $\bar{p}$[p_name] is null or has substring $\sigma$[color]:
            $P \leftarrow$ true ; break
    if $P$ is not true: continue
    for $\bar{s} \in$ supplier where $\bar{t}$[l_suppkey] is null or
                                      equal to $\bar{s}$[s_suppkey]:
        $x \leftarrow \bar{s}$[s_nationkey]
        if $x$ is null: $S \leftarrow$ true ; break
        for $\bar{n} \in$ nation where $\bar{n}$[n_nationkey] = $x$:
            if $\bar{n}$[n_name] equals $\sigma$[nation]:
                $S \leftarrow$ true ; break
    if $P$ is true and $S$ is true: return true
return false
```

False positives: experimental results. Recall that null values in instances are randomly generated: each nullable attribute can become null with the same fixed probability, referred to as the null rate. We let null rates range from 0.5% to 6% in steps of 0.5% and from 6% to 10% in steps of 1%.

To get good estimates, we generate 100 instances for each null rate, and run 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 described above. The results of the experiment are shown in Figure 1, reporting the average over all executions.

The outcome of the experiment shows that the problem of incorrect query answers in SQL is not just theoretical but it may well occur in practical settings: all of the queries we tested produce false positives on incomplete databases with as low as 0.5% of null values.

In fact for some queries the percentage of false positives is very high: for Q_2, almost all answers are such, even when few nulls are present. For Q_3, at least a quarter of all answers are wrong when the null rate is just 3%, rising to at least half incorrect answers for the null rate of 8%. Other queries, such as Q_1 and Q_4, appear to be more robust (as we only find a lower bound on the number of false positives), but false positives are always present, even at the lowest null rate tested, and in fact they constitute at least 10% of all answers at null rates of 7% and above.

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.

5. CORRECTNESS: A SIMPLE TRANSLATION

Since false positives are a real problem, we want to devise evaluation strategies that avoid it, if correctness of query re-

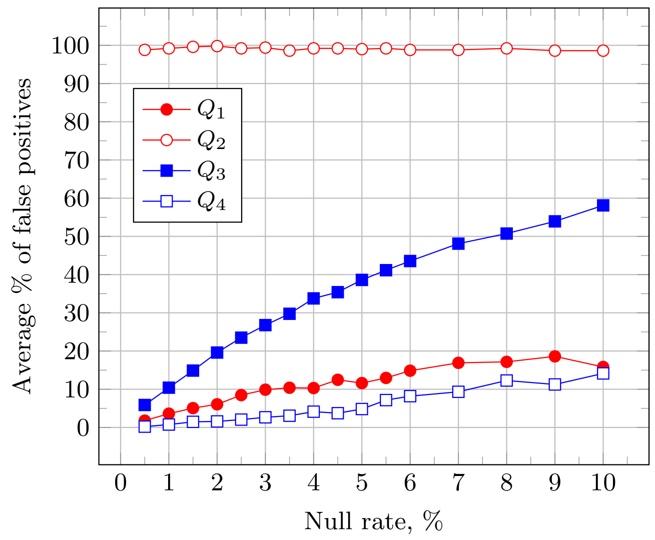

Figure 1: Average percentage of false positives produced by each query for increasing null rates.

sults is a concern. One such evaluation scheme was proposed recently [22]. We now review it, and explain why in its original form it cannot be implemented efficiently, despite good theoretical bounds.

We present it at the level of relational algebra. The key idea 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, for every database D, the tuples in $Q^{\mathbf{t}}(D)$ are certainly true, and those in $Q^{\mathbf{f}}(D)$ are certainly false:

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

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 $v(\bar{r}) = v(\bar{s})$.

The translations of [22] are shown in Figure 2. Here adom refers to the query computing the active domain (the union of all projections on each attribute of all relations), and θ^* refers to the translation of selection conditions, which is defined inductively as follows:

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

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 (AC^0), 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 [22] (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 much smaller databases than the smallest TPC-H compliant instance: some of the queries start running out of memory already on instances with fewer than 10^3 tuples.

All this tells us that we need an implementable alternative, which we present next.

6. AN IMPLEMENTATION-FRIENDLY TRANSLATION

To overcome the practical difficulties posed by the translation in Figure 2, we propose an alternative translation that is implementation-friendly and comes with sufficient correctness guarantees. In this translation, we do 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: tuples that are certain answers to $Q_1 - Q_2$ are those that are certainly answers to Q_1, and certainly not answers to Q_2. That necessitated working with the complex $Q^{\mathbf{f}}$ translation.

But we can use a slightly different rule: if a tuple is a certain answer to Q_1, and it does not match any tuple that could possibly be an answer to Q_2, then it is a certain answer to $Q_1 - Q_2$. 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.

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, 30] – 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).

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

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

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

Figure 2: Relational algebra translations of [22].

$$R^+ = R \tag{3.1}$$
$$(Q_1 \cup Q_2)^+ = Q_1^+ \cup Q_2^+ \tag{3.2}$$
$$(Q_1 \cap Q_2)^+ = Q_1^+ \cap Q_2^+ \tag{3.3}$$
$$(Q_1 - Q_2)^+ = Q_1^+ \mathbin{\overline{\ltimes}_{\Uparrow}} Q_2^? \tag{3.4}$$
$$\big(\sigma_\theta(Q)\big)^+ = \sigma_{\theta^*}(Q^+) \tag{3.5}$$
$$(Q_1 \times Q_2)^+ = Q_1^+ \times Q_2^+ \tag{3.6}$$
$$\big(\pi_\alpha(Q)\big)^+ = \pi_\alpha(Q^+) \tag{3.7}$$

$$R^? = R \tag{4.1}$$
$$(Q_1 \cup Q_2)^? = Q_1^? \cup Q_2^? \tag{4.2}$$
$$(Q_1 \cap Q_2)^? = Q_1^? \ltimes_{\Uparrow} Q_2^? \tag{4.3}$$
$$(Q_1 - Q_2)^? = Q_1^? - Q_2^+ \tag{4.4}$$
$$\big(\sigma_\theta(Q)\big)^? = \sigma_{\theta^{**}}\big(Q^?\big) \tag{4.5}$$
$$(Q_1 \times Q_2)^? = Q_1^? \times Q_2^? \tag{4.6}$$
$$\big(\pi_\alpha(Q)\big)^? = \pi_\alpha\big(Q^?\big) \tag{4.7}$$

Figure 3: The improved translation with correctness guarantees.

and the *left unification anti-semijoin* is

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

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 [22]. Note that, in this notation, $R^{\mathbf{f}} = \mathsf{adom}^{\mathsf{ar}(R)} \mathbin{\overline{\ltimes}_{\Uparrow}} R$.

Now, we can see why queries that represent potential answers are useful.

Lemma 1. *Consider the translations $Q \mapsto Q^+$ given in Figure 3 by (3.1)–(3.7), where the only assumption on $Q_2^?$ in (3.4) is that it represents potential answers to Q_2. Then Q^+ has correctness guarantees for Q.*

Proof sketch. The proof is by induction on the structure of the query; here, we show the important case of set difference. Let $Q = Q_1 - Q_2$, let D be a database, let $\bar{r}$ be in $Q^+(D) = Q_1^+(D) \mathbin{\overline{\ltimes}_{\Uparrow}} Q_2^?(D)$, and let v be a valuation on D. We need to show that $v(\bar{r}) \in Q\big(v(D)\big)$. As $\bar{r}$ is in $Q_1^+(D)$, we get that $v(\bar{r}) \in Q_1\big(v(D)\big)$ by the induction hypothesis. Now, suppose that $v(\bar{r}) \in Q_2\big(v(D)\big)$. Since $Q_2^?$ represents potential answers to Q_2 by assumption, we have $v(\bar{r}) \in v\big(Q_2^?(D)\big)$. Hence, there exists a tuple $\bar{s} \in Q_2^?(D)$ that unifies with $\bar{r}$ and, as $\bar{r} \in Q_1^+(D)$, this implies that $\bar{r} \in Q_1^+(D) \ltimes_{\Uparrow} Q_2^?(D)$, which contradicts our assumption that $\bar{r} \in Q_1^+(D) \mathbin{\overline{\ltimes}_{\Uparrow}} Q_2^?(D)$. This shows that $v(\bar{r}) \notin Q_2\big(v(D)\big)$ and, in turn, $v(\bar{r}) \in Q\big(v(D)\big)$. $\square$

Given this lemma, our next goal is to produce a translation of queries that represent potential answers. As for queries Q^+, this can be done almost by mimicking the structure of queries and using a query with correctness guarantees when it comes to translating the difference operation. We also need to modify selection conditions: the new translation $\theta \mapsto \theta^{**}$ 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 give it here:

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

Lemma 2. *Consider the translations $Q \mapsto Q^?$ given in Figure 3 by (4.1)–(4.7),where the only assumption on Q_2^+ in (4.4) is that it has correctness guarantees for Q. Then $Q^?$ represents potential answers to Q.*

Proof sketch. The proof is by induction on the structure of the query; here, we present two cases for illustration.

When $Q = Q_1 \cap Q_2$, we have that $Q^? = Q_1^? \ltimes_{\Uparrow} Q_2^?$. Let $\bar{r} \in Q\big(v(D)\big)$; then, by the induction hypothesis, $\bar{r}$ is in $v\big(Q_i^?(D)\big)$ for $i = 1, 2$. So, there are tuples $\bar{r}_i \in Q_i^?(D)$ such that $v(\bar{r}_1) = v(\bar{r}_2) = \bar{r}$. Hence, $\bar{r}_1 \Uparrow \bar{r}_2$ and thus $\bar{r} = v(\bar{r}_1) \in v\big(Q_1^?(D)\big) \ltimes_{\Uparrow} v\big(Q_2^?(D)\big) = v\big(Q^?(D)\big)$.

Next, consider $Q = Q_1 - Q_2$ and $Q^? = Q_1^? - Q_2^+$. Let $\bar{r} \in Q\big(v(D)\big)$. Then, $\bar{r} \in Q_1\big(v(D)\big)$ and $\bar{r} \notin Q_2\big(v(D)\big)$. By the induction hypothesis, $\bar{r} \in v\big(Q_1^?(D)\big)$ and so there is $\bar{s} \in Q_1^?(D)$ such that $\bar{r} = v(\bar{s})$. Assume $\bar{s} \in Q_2^+(D)$. Since Q_2^+ has correctness guarantees for Q by assumption, we have $\bar{r} = v(\bar{s}) \in Q_2\big(v(D)\big)$, which is a contradiction. Hence, $\bar{s} \notin Q_2^+(D)$ and so $\bar{s} \in Q^?(D)$. Therefore, $\bar{r} \in v\big(Q^?(D)\big)$ as required. $\square$

Lemmas 1 and 2 tell us that we can now combine the two translations in Figure 3 to obtain correctness guarantees. Using the lemmas, mutual induction on the expressions in Figure 3 shows the following.

THEOREM 1. *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, $Q^+(D) \subseteq \mathsf{cert}(Q, D)$ for every database D.*

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. As an example, consider the query

$$Q = R - \big(\pi_\alpha(T) - \sigma_\theta(S)\big)$$

and suppose it has arity k. Its corresponding translation $Q^{\mathbf{t}}$ that follows the rules in Figure 2 is

$$R \cap \big((\pi_\alpha(\mathsf{adom}^k \,\overline{\ltimes}_{\Uparrow}\, T) - \pi_\alpha(\mathsf{adom}^k \ltimes_{\Uparrow} T)) \cup \sigma_{\theta^*}(S)\big)$$

while the translation Q^+ we propose is much simpler:

$$R \,\overline{\ltimes}_{\Uparrow}\, \big(\pi_\alpha(T) - \sigma_{\theta^*}(S)\big)$$

and avoids Cartesian products of very large sets, computed multiple times, as in $Q^{\mathbf{t}}$.

We conclude this section with a few remarks. First, the translation of Figure 3 is really a family of translations. The proof of Theorem 1 applies to show the following.

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)–(3.7), and*
- *containing those listed in(4.1)–(4.7),*

then the resulting translation continues to satisfy the claim of Theorem 1.

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.

Second, the reason why queries $Q^?$ produce approximations of sets that represent potential answers is the same as for queries Q^+ to approximate certain answers, namely complexity. It can be easily seen that checking whether a set A represents potential answers to a given query Q on D is in coNP, and for some queries the problem is coNP-hard as well.

PROPOSITION 1. *There is a fixed 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?*

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 a 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 $\mathsf{Eval}_{\mathsf{SQL}}(Q_2, D_2) = \varnothing$. Nonetheless, in all our experiments Q^+ will always produce *all* of the certain answers returned by SQL.

As a final remark, note that (4.3) in Figure 3 is not the only possibility, since intersection is a commutative operation, but left unification semijoin is not. Correctness guarantees hold if we replace the left unification semijoin with the right one that keeps unifiable tuples from the second argument.

7. THE PRICE OF CORRECTNESS

Now that we have established the correctness of the translation $Q \mapsto Q^+$, our goal is to test it. For this, we take queries Q_1—Q_4 from Section 3, generate incomplete TPC-H instances, and then run Q_1—Q_4 as well as their translations to compare their performance.

Translations of SQL queries

The translation $Q \mapsto Q^+$ was given at the level of relational algebra. While there are multiple relational algebra simulators freely available, we want to carry out our experiments using a real DBMS on instances of realistic size (which rules out relational algebra simulators). Thus, we shall take SQL queries Q_1—Q_4, apply the translation $Q \mapsto Q^+$ to their relational algebra equivalents, and then run the results of the translation as SQL queries.

Expressing Q_1—Q_4 in relational algebra is standard. We remark though that traditional database texts tend to provide only simplified translations from SQL to algebra; for a full one, including nested subqueries that can themselves contain subqueries, a good source is [29] and we follow it here. As an example, consider query Q_3. Its relational algebra translation is

$$\pi_{\texttt{o\_orderkey}}\big(\texttt{orders} - \pi[\sigma_\theta(\texttt{lineitem} \times \texttt{orders})]\big)$$

where the inner projection is on the attributes of relation `orders` and the condition θ is `l_orderkey` = `o_orderkey` $\wedge$ `l_suppkey` $\neq$ `$supp_key`.

Before computing Q_3^+, we need to address one more technical issue. Often, at least in the theoretical literature, SQL nulls are identified with Codd nulls (non-repeating marked nulls). While in many cases this way of modeling SQL nulls is proper, it does not always work. The main issue is that comparing a null with itself results in *true* for Codd nulls, but *unknown* for SQL nulls. For instance, computing the self-join of $R = \{$**NULL**$\}$ by

```
SELECT R1.A FROM R R1, R R2 WHERE R1.A = R2.A
```

results in the empty set, while for the Codd database $R = \{\bot\}$, the evaluation of $R \bowtie R$ is $\{\bot\}$. This tells us that in full generality we cannot guarantee correctness with the SQL implementation of nulls, as it is too coarse to see when null refers to the same value. However, the situation that causes this problem – when a nullable attribute is compared with itself in a self-join – is not very common, and does not affect us here as long as we make a minor adjustment of the translations Q^+ and $Q^?$ to work correctly when evaluated as SQL queries.

As expected, the adjustment occurs in selection conditions. For the Q^+ translation, we need to ensure that attributes compared for equality are not nulls (the existing translation θ^* already ensures that for disequality comparisons). 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 θ^{**} already does it for equalities). 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 $(A \neq c)^{**}$. For all the queries considered here, these ensure that Q^+ continues to under-approximate certain answers and $Q^?$ continues to represent possible answers even when SQL evaluation rules are used for conditions with nulls.

Applying then the translations to Q_3 gives us Q_3^+ as

$$\pi_{\texttt{o\_orderkey}}\big(\texttt{orders} - \pi[\sigma_{\theta'}(\texttt{lineitem} \times \texttt{orders})]\big)$$

where the inner projection is on the attributes of `orders` and θ' is $\texttt{l\_orderkey} = \texttt{o\_orderkey} \wedge \big(\texttt{l\_suppkey} \neq \texttt{\$supp\_key} \vee \mathsf{null}(\texttt{l\_suppkey})\big)$. The left unification anti-semijoin produced by the translation is simplified to difference in Q_3^+ due to the following observation: if R is a relation that has a key, and $S \subseteq R$, then $R \,\overline{\ltimes}_{\Uparrow}\, S = R - S$. In the above query, this applies since the inner projection is contained in `orders`. Summing up, Q_3^+ is expressed in SQL as

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT *
       FROM   lineitem
       WHERE  l_orderkey = o_orderkey
       AND    ( l_suppkey <> {s_key}
                OR l_suppkey IS NULL ) )
```

In fact, a key feature of all the translations is that they change some conditions of the form `A=B` to `A=B OR B IS NULL`. 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. One can in fact observe that for a query of the form

```
SELECT * FROM R WHERE NOT EXISTS
    ( SELECT * FROM   S, ..., T
      WHERE  ( A=B OR B IS NULL ) AND ··· AND
             ( X=Y 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. One can then we can 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 $\neg\exists\phi_i(\bar{x})$, ending up with a query of the form

```
SELECT * FROM R WHERE NOT EXISTS
    ( SELECT * FROM $S_i$, $i \in I_1$ WHERE $\bigwedge_j \psi_j^1$ )
AND ··· AND NOT EXISTS
    ( SELECT * FROM $S_i$, $i \in I_k$ WHERE $\bigwedge_j \psi_j^k$ )
```

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.

Translating additional features. Queries Q_1—Q_4 on which we test the approach go slightly beyond relational algebra as used in the previous section: they use $>$ and **`LIKE`** comparisons, and Q_2 refers to an aggregate subquery. As for the first two, looking at the SQL-adjusted translations of selection conditions, we can see that there is nothing special about (dis)equality. The same translations can be applied to other comparisons, and this is what we do. For the aggregate subquery, we just treat it as a black box, that is, we view the result of that subquery as a value c and apply the translation to condition $\texttt{c\_acctbal} > c$.

Experimental results

Note that we measure *relative performance* of correct translations Q_i^+s, that is, the ratio of running times of Q_i^+s and the original queries Q_is. Intuitively, this ratio should not significantly depend on the size of generated instances. With this hypothesis in mind, we first report detailed results for the smallest allowed size of TPC-H instances (roughly 1GB). After that, we test our hypothesis using instances of 3GB, 6GB, and 10GB size, and show that indeed relative performances remain about the same for all instance sizes for queries Q_1, Q_2, and Q_3, although they can decrease slightly for Q_4 (we shall discuss this later).

For the experiments described below, we use the DBGen tool of TPC-H to generate databases of about 1GB each, and then we populate them with nulls, depending on the prescribed null rate. For each null rate in the range 1%–5%, in steps of 1%, we generate 10 incomplete databases. On each such database, we instantiate our test queries 5 times with randomly generated values for their parameters, and we run each query instance 3 times. The results that we report are averages of those runs.

Since we are interested in the *cost* of correctness, we report *relative values* of the parameters: one for the original query, and the other for the translation. Thus, if t is the time it takes a query Q to run, and t^+ is the running time of Q^+, we report the ratio t^+/t. In particular, staying close to 1 means that the price of correctness is low as correctness guarantees do not affect running time; if it drops below 1, we actually win by running a correct version of the query.

Based on the experiments we conduct, we observe three types of behavior, reported in Figure 4.

1. For queries Q_1 and Q_3, the price of correctness is *negligible* for most applications: under 4% for both.

2. For query Q_2, the translation with correctness guaranteed is *significantly faster* than the original query; in fact it is more than 3 orders of magnitude faster on average. Note that the relative performance scale in the graph for Q_2^+ ranges from $2 \cdot 10^{-4}$ to $8 \cdot 10^{-4}$.

3. For query Q_4, the behavior is the worst among those we observed, as the running time of Q_4^+ almost doubles the running time of Q_4. This is still tolerable though if correctness of results is very important.

Larger database instances. The previous results were obtained for the smallest allowed TPC-H instances. As we

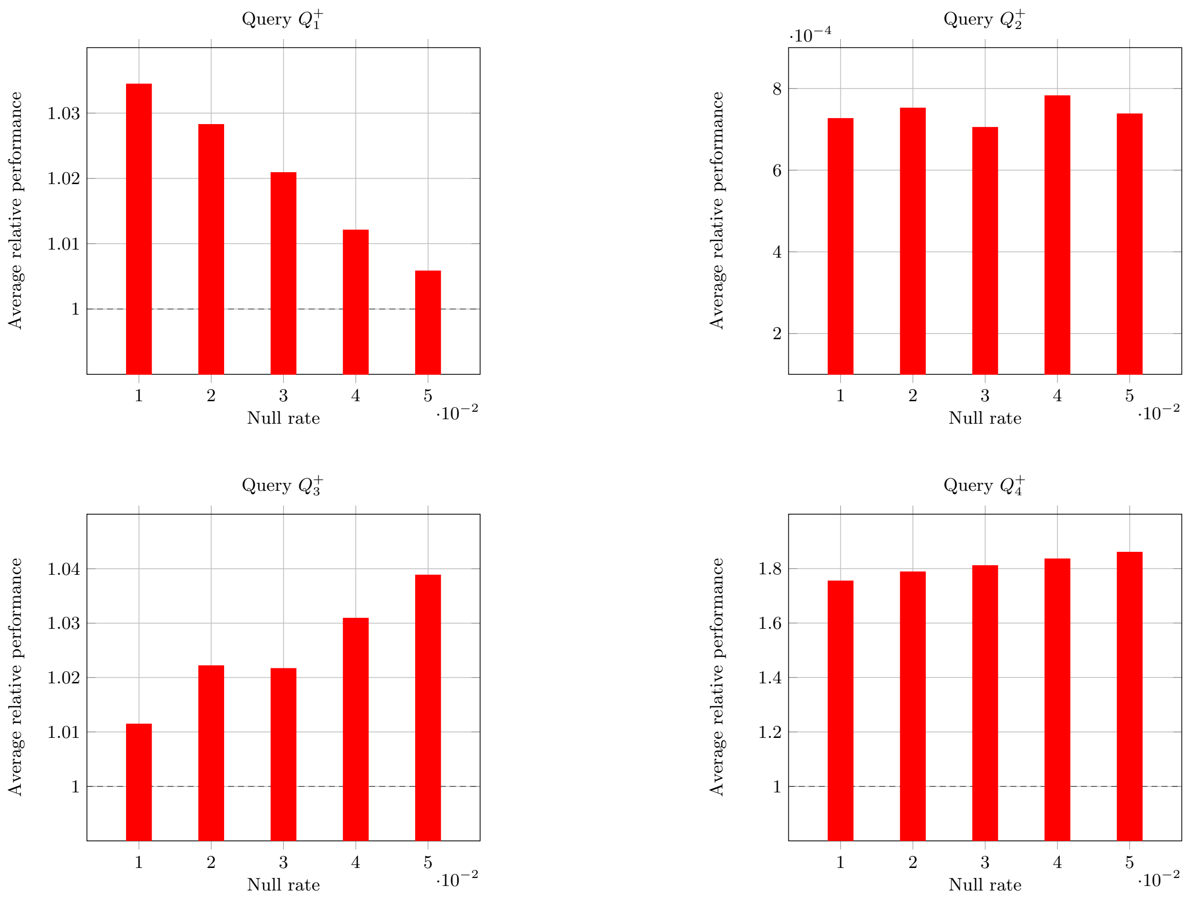

Figure 4: Average relative performance of queries with correctness guarantees.

Query	1GB	3GB	6GB	10GB
Q_1	1.0058 – 1.0344	0.9925 – 1.0148	0.9791 – 1.0113	1.0034 – 1.0201
Q_2	0.0007 – 0.0008	0.0002 – 0.0003	0.0001 – 0.0001	0.0001 – 0.0001
Q_3	1.0114 – 1.0387	1.0105 – 1.0367	0.9873 – 1.0226	1.0007 – 1.0628
Q_4	1.7530 – 1.8587	1.7986 – 1.9296	2.0532 – 2.2536	3.5354 – 3.8900

Table 1: Ranges of average relative performance – Q_i^+ vs Q_i – for instances up to 10GB.

explained, since we measure relative performance, we conjectured that the exact size of the instance should not have much impact: running times for both Q_is and Q_i^+s will increase, but proportionally so. To confirm this experimentally, we generated instances of sizes 3GB, 6GB, and 10GB, and ran similar tests (with fewer test runs for larger instances, as running times increased significantly). Our results, summarized in Table 1, validate our conjecture completely for queries Q_1, Q_2, and Q_3, as relative performances indeed change very little. For Q_4, we observe a decrease in performance from roughly half the speed of the original query for 1GB databases to one quarter of the speed for 10GB databases; we shall comment on this below.

Before analyzing these results, we address the standard measures for evaluating 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 the previous section, we can thus state that precision of our algorithms is 100%. Recall refers to the fraction of relevant answers returned. In our case, we can look at 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.

We saw that, 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, one needs either highly intractable algorithms for computing certain answers, or at least algorithms for identifying false positives. The latter we gave in Section 4 for the SQL evaluation of Q_1—Q_4, and tested them on smaller TPC-H instances generated by DataFiller. Thus, we ran queries Q_i^+s (modified versions for Q_2 and Q_4) on those smaller instances and observed that they returned precisely the answers to the Q_is except false positive tuples. That is, for those instances, the recall rate was 100%, and we did not miss any certain answers.

Discussion

We now discuss the factors that cause the behavior reported in Figure 4. We start with queries Q_1 and Q_3, whose behavior is quite similar. Note that the key change that our translation introduces is the change of comparisons `A` **`op`** `B` to (`A` **`op`** `B`) **`OR`** `B` **`IS NULL`** inside correlated **`NOT EXISTS`** subqueries. The number of such disjunctions is small, and they are well handled by the optimizer, resulting in small overheads. For Q_1^+, 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.

For query Q_2, the translation is similar, but there is one big difference: after we split the disjunction in the correlated **`NOT EXISTS`** subqueries, as explained earlier, one of the resulting **`NOT EXISTS`** subqueries becomes *decorrelated*. It simply tests for the existence of nulls in the attribute `o_custkey` of `orders`, and once it finds it the evaluation of the entire query ends, as we know that the result will be empty. The original query, on the other hand, spends most of its time looking for incorrect answers: this is the query with a rate of false positive answers close to 100%. Hence, in this case, the translation Q_2^+ not only ensures correctness, but also speeds up the execution time by a factor of over 10^3, as it is able to detect early that the correct answer is empty. In fact, as instances grow larger, one wins even more by using the correct query Q_2^+, as the original Q_2 is forced to spend more time looking for incorrect answers.

Query Q_4 is the hardest one to deal with. Without splitting the **`OR`** conditions, PostgreSQL produces astronomical costs of query plans, as it resorts to nested-loop joins, even for large tables (this is due to the fact that it under-estimates the size of joins, which is a known issue for major DBMSs [18]). Hence the direct translation of this query requires some tuning. This is achieved in two steps. First, we split the disjunctions into several **`NOT EXISTS`** conditions, as explained earlier. Even then, **`NOT EXISTS`** subqueries have nested **`EXISTS`** subqueries, each of them appearing twice. We define those queries as views (using **`WITH`**) and then replace subqueries with references to those views. These modifications are sufficient to make the optimizer produce better estimates and a reasonable query plan, which runs at roughly half the speed of the original query (for 1GB databases).

What makes the performance of Q_4 the worst of the four queries is that it is the only one that has a multi-way join in the **`NOT EXISTS`** subquery; all others have no joins in such subqueries at all. This means that absolute running times are significantly higher for Q_4 than for other queries. The translation has four correlated subqueries, three of which use joins that involve the largest `lineitem` relation, which accounts for the decrease in relative performance (as the original query has only one multi-way join subquery). Of course the need to have these multiple subqueries has arisen from the inability of the optimizer to handle disjunctions with **`IS NULL`** conditions. We believe that this problem may be overcome with a proper implementation of marked nulls (see additional comments in Section 8).

Conclusions

Our 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 queries. This could not be achieved with the theoretical solutions presented earlier [22] and required new ways of modifying SQL queries. Depending on the exact translation involved, we saw queries running at roughly half the speed in the worst case, or 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 SQL evaluation returned.

8. FUTURE WORK

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.

Bag semantics. SQL queries use multiset, or bag semantics, and handling duplicates is an important aspect of the language. However, at this point we do not even have a proper theory of certain answers for bag semantics, neither established notions that one can measure against, nor complexity results. We need to understand what the analog of $\mathsf{cert}(Q, D)$ is for queries under bag semantics, and how to define Q^+ in that case.

Aggregate functions. An important feature of real-life queries is aggregation; in fact it 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 [7, 9]. Thus, much theoretical work is needed in this direction before practical algorithms emerge. There is a better understanding of aggregate queries in neighboring areas such as probabilistic databases [25, 27] or inconsistent databases [5], and this could serve as a starting point.

Marked nulls. The translations $Q \mapsto Q^+, Q^?$ work at the level of marked and Codd nulls, but SQL nulls fall a bit short of Codd nulls, not being able to compare a null with itself. While sample queries used here were not affected, some queries may be. Ideally, one would use marked nulls to overcome this problem. Marked nulls should also be used to overcome issues with **`OR IS NULL`** conditions, as the optimizer will see them as usual disjunctions involving value comparisons. Marked nulls have been implemented in connection with data exchange systems [13, 24], and one has access to multiple querying scenarios involving marked nulls using schema mapping benchmarks [3, 6]; hence we intend to create new base types that use marked nulls and experiment with translations in that setting. If marked nulls are not available, we need to find the precise characterization of queries for which the translations proposed here restore correctness with SQL nulls.

Incorporating constraints. In the definition of certain answers, we disregarded constraints, although every real-life database will satisfy some, typically keys and foreign keys. While constraints ψ can be incorporated into a query ϕ by finding certain answers to $\psi \rightarrow \phi$, for common classes of constraints we would like to see how to make direct adjustments to rewritings. We have seen one example of this: the presence of a key constraint let us replace $R \,\overline{\ltimes}_{\Uparrow}\, S$ by $R - S$. We would like to automate such query transformations based on common classes of constraints.

Other types of incomplete information. So far we dealt with missing-information nulls, but there are other interpretations. For instance, non-applicable nulls [20, 32] 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 [21] 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 [16].

Direct SQL rewriting. We have rewritten SQL queries by a detour via relational algebra. We should look into both running such queries directly on a DBMS (and perhaps take advantage of good properties of semijoins [17] that feature prominently in our translations), and into direct rewriting from SQL to SQL, without an intermediate language.

Acknowledgments

We are grateful to Marco Console for many helpful discussions during the early stages of this work, and to Chris Ré and the anonymous referees for their comments. Work partially supported by EPSRC grants J015377 and M025268.

9. 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] B. Alexe, W. C. Tan, and Y. Velegrakis. STBenchmark: Towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008.

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

[5] M. Arenas, L. E. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. P. Spinrad. Scalar aggregation in inconsistent databases. *TCS*, 296(3):405–434, 2003.

[6] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.

[7] J. Celko. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 1995.

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

[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] 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.

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

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

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

[17] D. Leinders, J. Tyszkiewicz, and J. Van den Bussche. On the expressive power of semijoin queries. *Inf. Process. Lett.*, 91(2):93–98, 2004.

[18] 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.

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

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

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

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

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

[24] 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.

[25] C. Ré and D. Suciu. Efficient evaluation of HAVING queries on a probabilistic database. In *DBLP*, pages 186–200, 2007.

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

[27] R. Ross, V. S. Subrahmanian, and J. Grant. Aggregate operators in probabilistic databases. *J. ACM*, 52(1):54–101, 2005.

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

[29] J. Van den Bussche and S. Vansummeren. Translating SQL into the relational algebra. Course notes, Hasselt University and Université Libre de Bruxelles, 2009.

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

[31] 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.

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

APPENDIX

We present the exact translations of the queries used in our experiments. Query Q_3^+ was already shown in Section 7. Queries Q_1^+, Q_2^+, and Q_4^+ are given below.

Query Q_1^+

```
SELECT s_suppkey, o_orderkey
FROM   supplier, lineitem l1, orders, nation
WHERE  s_suppkey = l1.l_suppkey
  AND  o_orderkey = l1.l_orderkey
  AND  o_orderstatus = 'F'
  AND  l1.l_receiptdate > l1.l_commitdate
  AND  s_nationkey = n_nationkey
  AND  n_name = $nation
  AND  EXISTS (
       SELECT *
       FROM   lineitem l2
       WHERE  l2.l_orderkey = l1.l_orderkey
         AND  l2.l_suppkey <> l1.l_suppkey )
  AND  NOT EXISTS (
       SELECT *
       FROM   lineitem l3
       WHERE  l3.l_orderkey = l1.l_orderkey
         AND  ( l3.l_suppkey <> l1.l_suppkey
                OR l3.l_suppkey IS NULL )
         AND  ( l3.l_receiptdate > l3.l_commitdate
                OR l3.l_receiptdate IS NULL
                OR l3.l_commitdate  IS NULL ) )
```

Query Q_2^+

```
SELECT c_custkey, c_nationkey
FROM   customer
WHERE  c_nationkey IN ($countries)
  AND  c_acctbal > (
       SELECT AVG(c_acctbal)
       FROM   customer
       WHERE  c_acctbal > 0.00
         AND  c_nationkey IN ($countries) )
  AND  NOT EXISTS (
       SELECT *
       FROM   orders
       WHERE  o_custkey = c_custkey )
  AND  NOT EXISTS (
       SELECT *
       FROM   orders
       WHERE  o_custkey IS NULL )
```

Query Q_4^+

```
WITH
part_view  AS (
    SELECT p_partkey
    FROM   part
    WHERE  p_name IS NULL
    UNION
    SELECT p_partkey
    FROM   part
    WHERE  p_name LIKE '%'||$color||'%' ),
supp_view  AS (
    SELECT s_suppkey
    FROM   supplier
    WHERE  s_nationkey IS NULL
    UNION
    SELECT s_suppkey
    FROM   supplier, nation
    WHERE  s_nationkey = n_nationkey
      AND  n_name = '$nation' )
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT *
       FROM   lineitem, part_view, supp_view
       WHERE  l_orderkey = o_orderkey
         AND  l_partkey = p_partkey
         AND  l_suppkey = s_suppkey )
  AND  NOT EXISTS (
       SELECT *
       FROM   lineitem, supp_view
       WHERE  l_orderkey = o_orderkey
         AND  l_partkey IS NULL
         AND  l_suppkey = s_suppkey
         AND  EXISTS ( SELECT * FROM part_view ) )
  AND  NOT EXISTS (
       SELECT *
       FROM   lineitem, part_view
       WHERE  l_orderkey = o_orderkey
         AND  l_partkey = p_partkey
         AND  l_suppkey IS NULL
         AND  EXISTS ( SELECT * FROM supp_view ) )
  AND  NOT EXISTS (
       SELECT *
       FROM   lineitem
       WHERE  l_orderkey = o_orderkey
         AND  l_partkey IS NULL
         AND  l_suppkey IS NULL
         AND  EXISTS ( SELECT * FROM part_view )
         AND  EXISTS ( SELECT * FROM supp_view ) )
```