On the Codd Semantics of SQL Nulls

Paolo Guagliardo and Leonid Libkin

School of Informatics, University of Edinburgh

Abstract. Theoretical models used in database research often have subtle differences with those occurring in practice. One particular mismatch that is usually neglected concerns the use of *marked nulls* to represent missing values in theoretical models of incompleteness, while in an SQL database these are all denoted by the same syntactic **NULL** object. It is commonly argued that results obtained in the model with marked nulls carry over to SQL, because SQL nulls can be interpreted as *Codd nulls*, which are simply marked nulls that do not repeat. This argument, however, does not take into account that even simple queries may produce answers where distinct occurrences of **NULL** do in fact denote the same unknown value. For such queries, interpreting SQL nulls as Codd nulls would incorrectly change the semantics of query answers.
To use results about Codd nulls for real-life SQL queries, we need to understand which queries preserve the Codd interpretation of SQL nulls. We show, however, that the class of relational algebra queries preserving Codd interpretation is not recursively enumerable, which necessitates looking for sufficient conditions for such preservation. Those can be obtained by exploiting the information provided by **NOT NULL** constraints on the database schema. We devise mild syntactic restrictions on queries that guarantee preservation, do not limit the full expressiveness of queries on databases without nulls, and can be checked efficiently.

1 Introduction

Query evaluation is a fundamental task in data management, and very often it has to be performed on databases that have incomplete information. This is especially true in applications such as data integration [7, 4], data exchange [1] and ontology-based data access [2, 6] that rely on the standard tools offered by existing relational database technology, in particular SQL, in order to take advantage of its efficiency. There is much theoretical research on query answering and its applications that uses well established models of incompleteness. However, there is an important and often neglected mismatch between theoretical models and real-life SQL, which has an impact on the semantics of query answers.

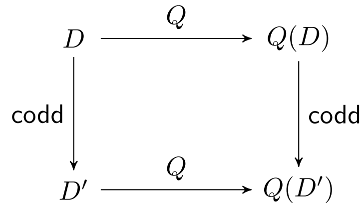
Theoretical work traditionally adopts a model of incompleteness where missing values in a database are represented by *marked* (also called *naive* or *labeled*) *nulls*. In SQL, however, missing values are all represented by the same syntactic object: the infamous **NULL**. Marked nulls are more expressive than SQL nulls: two unknown values can be asserted to be the same just by denoting them with the same null. For example, in a relation R with attributes *name* and *age*, one

can put tuples $(\text{"Alice"}, \bot_1)$ and $(\text{"Bob"}, \bot_1)$ to express the fact that Alice and Bob have the same age, even though their actual age, represented by the null $\bot_1$, is unknown. However, there is a standard argument often found in the literature: SQL nulls are modeled by *Codd nulls*, that is, non-repeating marked nulls. Then each **NULL** is interpreted as a fresh marked null that does not appear anywhere else in the database.

But do Codd nulls properly model SQL nulls? It is sometimes argued (e.g., in [8]) that they are different, but this assumes a special type of evaluation – called *naive* [5] – under which marked nulls are simply viewed as new constants, e.g., $\bot_1 = \bot_1$ but $\bot_1 \neq \bot_2$ etc. Under naive evaluation, the conjunctive query $q(x) :- R(x, y),\ R(z, y'),\ y = y'$ will produce {"Alice", "Bob"}, while the same query in SQL, `SELECT R1.name FROM R R1, R R2 WHERE R1.age = R2.age`, will produce the empty set.

This mismatch is due to the SQL's use of 3-valued logic (3VL): comparisons involving nulls in SQL result in the truth value *unknown*, even if a null is compared to itself, which happens in the above example. But this is easy to fix by using the same comparison semantics for Codd nulls: every condition $x = y$ or $x \neq y$ evaluates to *unknown* if x or y is a null. So, is this sufficient to reconcile the mismatch between Codd nulls and SQL nulls?

The answer is still negative, because we have not taken into account the role of queries. 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 $\mathsf{codd}(R)$ be the result of replacing SQL nulls in R with distinct marked nulls, e.g., for $R = \{(\text{"Alice"}, \textbf{NULL}), (\text{"Bob"}, \textbf{NULL})\}$, we would have $\mathsf{codd}(R) = \{(\text{"Alice"}, \bot_1), (\text{"Bob"}, \bot_2)\}$. Technically, $\mathsf{codd}(R)$ is the set of such relations, because we can choose distinct marked nulls arbitrarily (e.g., $\{\bot_3, \bot_4\}$ instead of $\{\bot_1, \bot_2\}$), but they 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 $\mathsf{codd}\big(Q(D)\big)$, i.e., there must be a way of assigning Codd nulls to SQL nulls in $Q(D)$ that will result in $Q(D')$.

Does this condition hold for queries in some sufficiently expressive language like relational algebra? Unfortunately, the answer is negative, even for very simple queries. Take for example a database D with $T = \{\textbf{NULL}\}$ and $S = \{1, 2\}$, and consider the query Q that computes the Cartesian product of T and S. The answer to Q on D is $Q(D) = \{(\textbf{NULL}, 1), (\textbf{NULL}, 2)\}$. The Codd interpretation of D is

a database D' where $T = \{\bot_1\}$ and $S = \{1, 2\}$, and $Q(D') = \{(\bot_1, 1), (\bot_1, 2)\}$. Since $\bot_1$ is repeated, $Q(D')$ *cannot* be obtained by replacing SQL nulls in $Q(D)$ with Codd nulls.

The culprit in the above example is that the query, evaluated on the Codd interpretation of the original SQL database, produces an answer that is not a Codd table, as it contains repeated nulls. While in this particular example it is easy to detect such an undesirable behavior, we show that in general the class of relational algebra queries that transform SQL databases into Codd databases is not recursively enumerable, and thus it is impossible to capture it by a syntactic fragment of the language.

Given the lack of an effective syntax for queries that preserve Codd semantics in query answers, we can only hope for sufficient conditions, i.e., finding reasonable syntactic fragments that guarantee this property. However, simply choosing a subset of relational algebra operations will not give us a useful restriction: as we saw before, one of the problematic constructs is Cartesian product, and we obviously do not want a fragment that prohibits joins. Thus, we must look for a more refined solution.

The idea is to consider database schemas with **NOT NULL** constraints, which are very common in practice: base relations in real SQL databases will almost always have a **PRIMARY KEY** declared on them. We then exploit these constraints to come up with mild restrictions on queries, which have the following properties:

- they guarantee the preservation of Codd semantics of SQL nulls in query answers on all incomplete databases;
- they can be checked efficiently; and
- they do not restrict the full expressiveness of relational algebra queries on databases without nulls, where all attributes are declared as **NOT NULL**.

2 Preliminaries

We assume three countably infinite and pairwise disjoint sets of elements: names (Names), constants (Const), and nulls (Null). Elements of Null are denoted by $\bot$, possibly with sub- or superscripts, and we call a *signature* any finite subset of Names. A *naive record* is a map from some signature to $\mathsf{Const} \cup \mathsf{Null}$. To be as close as possible to SQL's data model, we define a *naive table* as a **bag** of naive records (which may occur multiple times) over the same signature.

Schemas and databases. A relational schema consists of a signature of relation names and a function sign that maps each relation name R to a signature $\mathsf{sign}(R)$, whose elements are called the *attributes* of R. A *naive database* D associates each relation name R with a naive table R^D over $\mathsf{sign}(R)$. We denote the sets of constants and nulls occurring in D by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$, respectively. A *Codd database* (respectively, table) is a naive one in which nulls do not repeat, that is, there can be at most one occurrence of each element from Null. Note that a Codd database is a set of Codd tables, but the converse need not hold.

SQL databases. We use the special symbol $\mathbf{N} \notin \mathsf{Const} \cup \mathsf{Null}$ to denote SQL's **NULL** value. *SQL databases* (as well as tables and records) are defined in a similar way as naive ones, but they are populated with elements of $\mathsf{Const} \cup \{\mathbf{N}\}$ rather than $\mathsf{Const} \cup \mathsf{Null}$.

Query language. We use relational algebra on bags, with the usual operations of projection π, selection σ, Cartesian product $\times$, union $\cup$, difference $-$, intersection $\cap$, renaming ρ, and duplicate elimination ε. These are all defined in the standard way [3]. Selection conditions are Boolean combinations of comparisons $x = y$ and $\mathsf{null}(x)$, where each x and y is either a constant or an attribute. The condition $\mathsf{null}(x)$, corresponding to **IS NULL** in SQL, tests whether the value of x is null. We use $x \neq y$ and $\mathsf{const}(x)$ for $\neg(x = y)$ and $\neg\,\mathsf{null}(x)$, respectively.

This language captures the basic fragment of SQL: **SELECT** [**DISTINCT**]-**FROM**-**WHERE** queries, with (correlated) subqueries preceded by possibly negated **IN** and **EXISTS**, combined by **UNION**, **INTERSECT** and **EXCEPT**, possibly with **ALL**.

The answer to a query Q on a (naive or SQL) database D is denoted by $Q(D)$. Selection conditions are evaluated using SQL's three-valued logic: (in)equality comparisons result in *unknown* if at least one of the arguments is null, and truth values are propagated through the connectives $\wedge$, $\vee$, $\neg$ following the well known truth tables of Kleene logic. Then, for a table T, the operation $\sigma_\theta(T)$ returns all occurrences of each record r in T for which the condition θ evaluates to *true*.

3 Codd Interpretation of SQL Nulls

Differently from naive databases, where nulls are elements of Null, missing values in SQL are all denoted by the special symbol $\mathbf{N}$. To reconcile this mismatch, the occurrences of $\mathbf{N}$ in an SQL database are typically interpreted as *non-repeating* elements of Null. That is, an SQL database can be seen as a Codd database where each occurrence of $\mathbf{N}$ is replaced by a fresh distinct marked null. Obviously, these nulls can be chosen arbitrarily as long as they do not repeat, so an SQL database may admit infinitely many Codd interpretations in general.

To make this notion more precise, let us denote by $\mathsf{sql}(D)$ the SQL database obtained from D by replacing each null (either $\mathbf{N}$ or an element of Null) with $\mathbf{N}$. Note that $\mathsf{sql}(D)$ is uniquely determined for any given (naive or SQL) database D. Then, for an SQL database D, we define

$$\mathsf{codd}(D) = \{D' \mid D' \text{ is a Codd database such that } \mathsf{sql}(D') = D\} \tag{1}$$

Even though this set may be infinite, its elements are all isomorphic. Thus, with some abuse of terminology, we can speak of *the* Codd interpretation of an SQL database, which is unique up to renaming of nulls.

Answers to queries are tables, which may of course contain nulls. So, if SQL nulls are interpreted as Codd nulls, this should apply also to query answers, not just input databases. More precisely, when computing the answer to a query Q on an SQL database D, there should always be a way of assigning Codd nulls to SQL nulls in $Q(D)$ so as to obtain $Q(D')$, where D' is the Codd interpretation of

D. In other words, the Codd interpretation of $Q(D)$ should be indistinguishable from $Q(D')$, that is, isomorphic to it. This requirement was shown in the diagram in the introduction and it is formally defined below.

Definition 1. *A query Q* preserves Codd semantics *if, for every SQL database D and for every $D' \in \mathsf{codd}(D)$, it holds that*

$$Q(D') \in \mathsf{codd}\big(Q(D)\big)\,. \tag{2}$$

The above definition can also be equivalently formulated as follows: a query Q preserves Codd semantics if, for every Codd database D,

$$\mathsf{sql}\big(Q(D)\big) = Q\big(\mathsf{sql}(D)\big) \text{ and} \tag{2a}$$

$$Q(D) \text{ is a Codd table.} \tag{2b}$$

Intuitively, in order to preserve Codd semantics a query must (2a) preserve the *occurrences* of nulls, regardless of repetitions, and (2b) avoid repetitions of nulls in the answers.

These two requirements are in fact independent of each other. For example, the query $Q = R \times S$ satisfies (2a) but not (2b), because on the Codd database D where $R = \{\bot_1\}$ and $S = \{\bot_2, \bot_3\}$ the answer $Q(D) = \{(\bot_1, \bot_2), (\bot_1, \bot_3)\}$ contains repeated nulls. On the other hand, the query $Q' = R \cap S$ satisfies (2b) but not (2a), because $\mathsf{sql}\big(Q'(D)\big) = \varnothing \neq \{\mathbf{N}\} = Q'\big(\mathsf{sql}(D)\big)$.

Can we syntactically capture the class of queries preserving Codd semantics? Unfortunately, the answer is no.

Proposition 1. *For every schema with at least one binary relation symbol, the set of queries that preserve Codd semantics is not recursively enumerable.*

To prove this, we can show that the set of preserving queries is not recursive, by reduction to the undecidability of emptiness of relational algebra expressions; since the complement of the set is clearly r.e., the result follows. Thus, we should look for syntactic restrictions that provide sufficient conditions for the preservation of Codd semantics. This is what we do in the next section.

4 Queries Preserving Codd Semantics

Since the set of queries that preserve Codd semantics cannot be captured syntactically, we can only hope for syntactic restrictions that are sufficient to guarantee this property. However, simply choosing a subset of relational algebra operations would be too restrictive to be useful: as we have seen in the examples, Cartesian product is one of the operations causing problems with the preservation of Codd semantics, and we certainly do not want a fragment that forbids joins altogether.

This suggests that, to come up with useful restrictions, we need some additional information beyond the query itself. Real databases typically must satisfy some integrity constraints, and when it comes to incomplete SQL databases the most basic form of constraint amounts to declaring some of the attributes in the

schema as **NOT NULL**. The use of this kind of constraints is extremely common in practice, because each base table in a real-life SQL database has almost always a subset of its attributes declared as **PRIMARY KEY**, which implies **NOT NULL**.

We model **NOT NULL** constraints as follows: the signature $\mathsf{sign}(R)$ of each base relation R is partitioned into two sets $\mathsf{sign}_{\mathsf{null}}(R)$ and $\mathsf{sign}_{\neg\mathsf{null}}(R)$ of *nullable* and *non-nullable* attributes, so that $\pi_{\mathsf{sign}_{\neg\mathsf{null}}(R)}(R^D)$ contains only elements of Const. That is, nulls are not allowed as values of the attributes in $\mathsf{sign}_{\neg\mathsf{null}}(R)$, while there are no restrictions on the values of the attributes in $\mathsf{sign}_{\mathsf{null}}(R)$. We extend these notions to relational algebra queries as follows:

$$
\begin{aligned}
\mathsf{sign}_{\mathsf{null}}\big(Q_1 \star Q_2\big) &= \mathsf{sign}_{\mathsf{null}}(Q_1) \cup \mathsf{sign}_{\mathsf{null}}(Q_2) \text{ for } \star \in \{\times, \cup\}\\
\mathsf{sign}_{\mathsf{null}}\big(Q_1 \cap Q_2\big) &= \mathsf{sign}_{\mathsf{null}}(Q_1) \cap \mathsf{sign}_{\mathsf{null}}(Q_2)\\
\mathsf{sign}_{\mathsf{null}}\big(Q_1 - Q_2\big) &= \mathsf{sign}_{\mathsf{null}}(Q_1)\\
\mathsf{sign}_{\mathsf{null}}\big(\pi_\alpha(Q)\big) &= \mathsf{sign}_{\mathsf{null}}(Q) \cap \alpha\\
\mathsf{sign}_{\mathsf{null}}\big(\sigma_\theta(Q)\big) &= \mathsf{sign}_{\mathsf{null}}\big(\varepsilon(Q)\big) = \mathsf{sign}_{\mathsf{null}}(Q)\\
\mathsf{sign}_{\mathsf{null}}\big(\rho_{A\to B}(Q)\big) &= \begin{cases} \big(\mathsf{sign}_{\mathsf{null}}(Q) - \{A\}\big) \cup \{B\} & \text{if } A \in \mathsf{sign}_{\mathsf{null}}(Q)\\ \mathsf{sign}_{\mathsf{null}}(Q) & \text{otherwise}\end{cases}\\
\mathsf{sign}_{\neg\mathsf{null}}(Q) &= \mathsf{sign}(Q) - \mathsf{sign}_{\mathsf{null}}(Q)
\end{aligned}
$$

From the above definition we immediately obtain:

Proposition 2. *Let Q be a query and D be a database. Then, $\pi_{\mathsf{sign}_{\neg\mathsf{null}}(Q)}\big(Q(D)\big)$ consists only of elements of Const.*

That is, independently of the data, the answer to a query Q can only have null values for the attributes in $\mathsf{sign}_{\mathsf{null}}(Q)$. We say that a query Q is *non-nullable* if $\mathsf{sign}_{\mathsf{null}}(Q) = \varnothing$.

To explain the restrictions that ensure preservation of Codd semantics, we need two definitions.

Definition 2. *The* parse tree *of a query Q is constructed as follows:*

- *Each relation symbol R is a single node labeled R.*
- *For each unary operation op_1, the parse tree of $\mathsf{op}_1(Q)$ has root labeled op_1 and the parse tree of Q rooted at its single child.*
- *For each binary operation op_2, the parse tree of Q op_2 Q' has root labeled op_2 and the parse trees of Q and Q' rooted at its left child and right child, respectively.*

Note that each node in the parse tree of Q defines a subquery of Q, so we can associate properties of such queries with properties of parse tree nodes.

Definition 3. *The* base *of a query is the set of relation names appearing in it. A node in the parse tree of a query satisfies:*

`NNC` (non-nullable child) *if it is not a leaf and at least one of its children is a non-nullable query;*

`NNA` (non-nullable ancestor) *if either itself or one of its ancestors is a query that is non-nullable;*

`DJB` (disjoint base) *if it corresponds to a binary operation and the base of its left child is disjoint with the base of its right child.*

We now state the main result.

Theorem 1. *Let Q be a relational algebra query whose parse tree is such that:*

(a) *each ε, $\cap$, and $-$ node satisfies* `NNC`*;*

(b) *each $\times$ node satisfies* `NNA`*;*

(c) *each $\cup$ node satisfies* `NNC` *or* `DJB` *or* `NNA`*.*

Then Q preserves Codd semantics.

These conditions do not restrict the full expressiveness of relational algebra on databases without nulls: when all attributes in the schema are non-nullable, every query is such, hence the restrictions are trivially satisfied.

Corollary 1. *On database schemas without nullable attributes, every relational algebra query satisfies the conditions of Theorem 1.* □

Also, the restrictions are easy to check:

Proposition 3. *Deciding whether a query satisfies the conditions of Theorem 1 can be done in linear time w.r.t. the number of nodes in its parse tree.*

We will now outline the main ideas behind the restrictions. If a query is non-nullable, then it trivially satisfies (2b). But there exist non-nullable queries for which (2a) does not hold: e.g., $\pi_A\big(\varepsilon(R)\big)$ when R has attributes A, B, of which only A is non-nullable. Thus, non-nullability needs to be used more carefully, on the subqueries of Q.

Duplicate elimination, intersection and difference cause problems with (2a). What these operations have in common is that they match nulls *syntactically*, i.e., as if they were constants. Now, SQL nulls are all syntactically the same, but this is not the case for Codd nulls. So, for these operations, it makes a difference whether we are using Codd nulls or SQL nulls. On the other hand, the remaining operations are not affected by this: projection, union and Cartesian product do not rely on syntactic matching of nulls, and nulls in selection conditions are not compared syntactically (equality and inequality conditions involving at least one null result in unknown). Thus, we only need to take care of ε, $-$ and $\cap$, and here is where the `NNC` condition comes into play. Requiring it for ε, $\cap$ and $-$ nodes is enough to ensure that the query satisfies (2a). Intuitively, when at least one of the input subqueries to each of these operations is non-nullable, no syntactic matching of nulls can occur, simply because one of the operands (the only one for ε) will not have nulls at all.

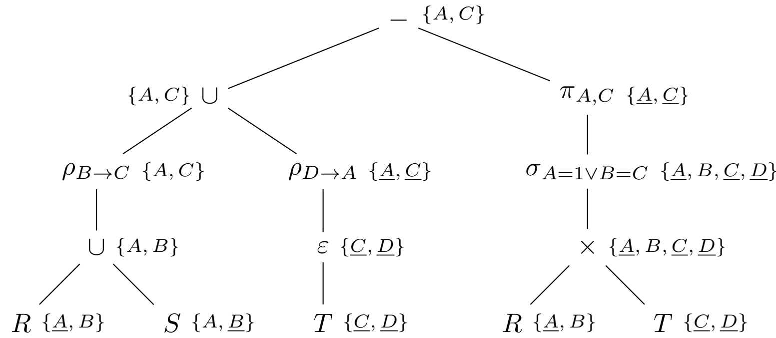
Proposition 4. *Let Q be a query such that each ε, $\cap$ and $-$ node in its parse tree satisfies* `NNC`*. Then Q satisfies property* (2a)*.*

If in addition to the condition of Proposition 4 we also required the query itself to be non-nullable, then preservation of Codd semantics would be guaranteed. However, this is too restrictive, as it would forbid even the simple retrieval of a base relation whenever some of its attributes are nullable. So, we need more refined ways of ensuring (2b) by restricting only the problematic operations.

Duplicate elimination, difference, selection and intersection always produce a table that is contained in at least one of the input tables, so their output may have repeated nulls only if these repetitions were already in the input. The same holds for projection, for a similar reason. The problematic operations w.r.t. (2b) are $\cup$ and $\times$. Both can create repetitions of nulls across records, and the latter can also create repetitions of nulls within a record. To restrict these operations appropriately, we use the `NNA` condition. Requiring this condition for each $\times$ and $\cup$ node is enough to ensure that the query satisfies (2b). Intuitively, repetitions of nulls that may be created by $\cup$ and $\times$ are allowed in the intermediate results of a query, but only as long as they will be eventually discarded, at the latest when the final output is produced.

In fact, the `NNA` condition for $\cup$ can be relaxed. We want to restrict a union operation when it may create "new" repetitions of nulls, but we need not do so when the only repetitions it may produce are those inherited from the input, which were introduced by the application of previous operations. There are two cases in which this happens: when one of the operands is non-nullable, or when the input tables are built from disjoint sets of base relations. The first is captured by the `NNC` condition, and the second by the `DJB` condition. We can then allow for these additional possibilities by requiring each $\cup$ node to satisfy `NNA` *or* `NNC` *or* `DJB`.

This explains the need for the conditions of Definition 3. As an example, let us consider the query $\big(\rho_{B\to C}(R\cup S)\cup\rho_{D\to A}\big(\varepsilon(T)\big)\big)-\pi_{A,C}\big(\sigma_{A=1\vee B=C}(R\times T)\big)$, whose parse tree is shown below:



Next to each node we indicate the corresponding signature, where non-nullable attributes are underlined (for the leaves, this information is given by the schema). On the left side of the tree, we see that the innermost $\cup$ node (lower in the tree) satisfies `DJB` (but not `NNC`), the ε node satisfies `NNC` and so is non-nullable, and in turn the outermost $\cup$ node satisfies `NNC` too. On the right side, we see that the

$\times$ node satisfies NNA because its ancestor π is non-nullable, which also ensures that the root node "$-$" satisfies NNC. Thus, the query preserves Codd semantics.

We conclude this section with an outline of the proof that checking the conditions of Theorem 1 can be done in linear time. Let n be the number of nodes in the binary parse tree of the query. First, we compute the base and the nullable attributes of each node by scanning the tree bottom up (n steps). We then mark all ε, $\cap$, $\cup$, $-$ nodes satisfying NNC, all $\cup$ nodes satisfying DJB, all σ, π, ρ nodes, and all leaves; for each node, we visit up to two child nodes, so this requires at most $2 \cdot n$ steps. Next, by traversing the tree breadth-first from the root, we remove the subtrees rooted in non-nullable nodes; in the worst case (when all nodes are nullable) this takes n steps. Finally, the query satisfies the conditions of Theorem 1 iff all nodes in the resulting tree are marked, which can be checked in one more pass (n steps).

5 Derived Operations and Further Refinements

In our query language we have explicitly included intersection, even though this operation is expressible in terms of difference. While it may seem redundant to have $\cap$ in the syntax of queries, this is not the case w.r.t. our restrictions for the preservation of Codd semantics. To see why, suppose we want the intersection of base relations R and S, but we do not have $\cap$ available as a primitive operation in our query language. Of course, we can always write $R-(R-S)$ or $S-(S-R)$; the problem is that, even though these queries are equivalent, the former satisfies the conditions of Theorem 1 iff R is non-nullable, while the latter satisfies them iff S is non-nullable. In either case, the requirement is stronger than the one for $R \cap S$, which is in fact given by their disjunction: $R \cap S$ satisfies the conditions of Theorem 1 iff R is non-nullable *or* S is non-nullable.

This suggests that adding some derived operations explicitly to the syntax of queries may lead to milder restrictions, allowing us to capture more queries preserving Codd semantics. We will show two such refinements: constant selections and semi/antijoins.

Selections. Looking at σ may seem counterintuitive as this operation did not need to be restricted in any way. However, consider the query $\sigma_{\mathsf{const}(A)}(R) \cap S$, for unary relation symbols R and S over a nullable attribute A. This query does not satisfy the conditions of Theorem 1, because neither $\sigma_{\mathsf{const}(A)}(R)$ nor S are non-nullable. But even though $\mathsf{sign}_{\mathsf{null}}\big(\sigma_{\mathsf{const}(A)}(R)\big) = \{A\}$, we can rest assured that in fact no nulls will appear in $\sigma_{\mathsf{const}(A)}(R^D)$. Here the idea is to include in our query language the operation of *constant selection* $\sigma_{\mathsf{const}(\alpha)}$ which, for a table T and $\alpha \subseteq \mathsf{sign}(T)$, returns all occurrences of each record r in T such that $r(A)$ is a constant for every attribute $A \in \alpha$. Then, we define $\mathsf{sign}_{\mathsf{null}}\big(\sigma_{\mathsf{const}(\alpha)}(Q)\big) = \mathsf{sign}_{\mathsf{null}}(Q) - \alpha$.

Semijoins and antijoins. The *theta-join* of two tables T_1, T_2 with disjoint sets of attributes is $T_1 \bowtie_\theta T_2 = \sigma_\theta\big(T_1 \times T_2\big)$. Adding $\bowtie_\theta$ to the language does not

change anything, but two operations derived from it can be added: *semijoin* $\ltimes_\theta$ and *antijoin* $\overline{\ltimes}_\theta$. These essentially correspond to **EXISTS** and **NOT EXISTS** in SQL. For tables T_1, T_2 with disjoint sets of attributes, they are defined as:

$$T_1 \ltimes_\theta T_2 = T_1 \cap \pi_{\mathsf{sign}(T_1)}(T_1 \bowtie_\theta T_2) \qquad T_1 \overline{\ltimes}_\theta T_2 = T_1 - T_1 \ltimes_\theta T_2$$

That is, $T_1 \ltimes_\theta T_2$ returns all occurrences of each record r in T_1 for which there is a record s in T_2 such that θ is true on $r \cup s$. Similarly, $T_1 \overline{\ltimes}_\theta T_2$ returns all occurrences of each record r in T_1 for which there is no record s in T_2 s.t. θ is true on $r \cup s$. We define $\mathsf{sign}_{\mathsf{null}}(Q_1 \ltimes_\theta Q_2) = \mathsf{sign}_{\mathsf{null}}(Q_1 \overline{\ltimes}_\theta Q_2) = \mathsf{sign}_{\mathsf{null}}(Q_1)$.

How do we restrict these operations to ensure (2a) and (2b)? Observe that $\ltimes$ is an intersection and $\overline{\ltimes}$ is a difference, and syntactic matching of nulls must be prevented in general for these operations. Here, however, one of the operands to this intersection/difference is always contained in the other, independently of whether SQL nulls or Codd nulls are used, so syntactic matching of nulls is not a problem in this case. In turn, we need no restriction on $\ltimes$ and $\overline{\ltimes}$ to guarantee (2a). As for (2b), observe that the output of both $\ltimes$ and $\overline{\ltimes}$ is always contained in their left input, so no new repetitions of nulls can be created and, in turn, we do not need any restriction in order to ensure (2b) either. Repetitions of nulls that might have been created in the right operand of $\ltimes$ or $\overline{\ltimes}$ are discarded. Thus we can use an alternative to `NNA`, called `RDS`: a node in a parse tree satisfies this condition if it is the right-descendant of a $\ltimes$ or a $\overline{\ltimes}$ operation.

Theorem 2. *Let Q be a query (extended with $\ltimes$, $\overline{\ltimes}$, σ_{const}) such that:* (a) *each ε, $\cap$, $-$ node satisfies* `NNC`*;* (b) *each $\times$ node satisfies* `NNA` *or* `RDS`*;* (c) *each $\cup$ node satisfies* `NNC` *or* `DJB` *or* `NNA` *or* `RDS`*. Then, Q preserves Codd semantics.*

An analog of Corollary 1 follows, and simple modifications to the algorithm of Proposition 3 show that these conditions are still linear-time testable.

6 Conclusions

The notion of Codd database we adopted in this paper only allows for constant tuples to occur multiple times, as nulls cannot repeat. Relaxing this requirement to also allow for duplicates of non-constant tuples raises several questions: How do we interpret multiple occurrences of a tuple with nulls in an SQL database? What is preservation of Codd semantics in this context? Do our restrictions still guarantee it?

Another interesting direction for future investigation concerns set semantics. Observe that a query Q can always be rewritten into a query Q' where duplicate elimination is suitably applied in each subexpression so that, on set databases (i.e., in which relations are sets), evaluating Q under set semantics is equivalent to evaluating Q' under bag semantics. If Q' satisfies our restrictions, preservation of Codd semantics is guaranteed on all (bag and set) databases and, in turn, Q preserves Codd semantics on set databases. However, this is too restrictive for even simple queries (e.g., $R \cup S$) and weaker conditions may be devised to deal specifically with set semantics.

Acknowledgements

Work partially supported by EPSRC grants EP/N023056/1 and EP/M025268/1. The authors would like to thank the anonymous reviewers for their useful comments.

References

1. Arenas, M., Barceló, P., Libkin, L., Murlak, F.: Foundations of Data Exchange. Cambridge University Press (2014)
2. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. J. Autom. Reasoning 39(3), 385–429 (2007)
3. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems: The complete book. Pearson Education (2009)
4. Halevy, A., Rajaraman, A., Ordille, J.: Data integration: The teenage years. In: VLDB. pp. 9–16 (2006)
5. Imielinski, T., Lipski, W.: Incomplete information in relational databases. Journal of the ACM 31(4), 761–791 (1984)
6. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to ontology-based data access. In: IJCAI. pp. 2656–2661 (2011)
7. Lenzerini, M.: Data integration: a theoretical perspective. In: ACM Symposium on Principles of Database Systems (PODS). pp. 233–246 (2002)
8. Libkin, L.: SQL's three-valued logic and certain answers. ACM Trans. Database Syst. 41(1), 1:1–1:28 (2016)