Handling SQL Nulls with Two-Valued Logic

Leonid Libkin, Liat Peterfreund

Handling SQL Nulls with Two-Valued Logic

Leonid Libkin
Univ. Edinburgh / ENS-Paris, PSL / Neo4j
libkin@inf.ed.ac.uk

Liat Peterfreund
ENS-Paris, PSL
liatpf.cs@gmail.com

ABSTRACT

The design of SQL is based on a three-valued logic (3VL), rather than the familiar Boolean logic with truth values true and false, to accommodate the additional truth value unknown for handling nulls. It is viewed as indispensable for SQL expressiveness, but is at the same time much criticized for leading to unintuitive behavior of queries and thus being a source of programmer mistakes.

We show that, contrary to the widely held view, SQL could have been designed based on the standard Boolean logic, without any loss of expressiveness and without giving up nulls. The approach itself follows SQL's evaluation which only retains tuples for which conditions in the WHERE clause evaluate to true. We show that conflating unknown, resulting from nulls, with false leads to an equally expressive version of SQL that does not use the third truth value. Queries written under the two-valued semantics can be efficiently translated into the standard SQL and thus executed on any existing RDBMS. These results cover the core of the SQL 1999 Standard, including SELECT-FROM-WHERE-GROUP BY-HAVING queries extended with subqueries and IN/EXISTS/ANY/ALL conditions, and recursive queries. We provide two extensions of this result showing that no other way of converting 3VL into Boolean logic, nor any other many-valued logic for treating nulls could have possibly led to a more expressive language.

These results not only present small modifications of SQL that eliminate the source of many programmer errors without the need to reimplement database internals, but they also strongly suggest that new query languages for various data models do not have to follow the much criticized SQL's three-valued approach.

ACM Reference Format:
Leonid Libkin and Liat Peterfreund. 2020. Handling SQL Nulls with Two-Valued Logic. In . ACM, New York, NY, USA, 13 pages. https://doi.org/...

1 INTRODUCTION

To process data with nulls, SQL uses a three-valued logic (3VL). This is one of the most often criticized aspects of the language, and one that is very confusing to programmers. Database texts are full of damning statements about the treatment of nulls such as the inability to explain them in a "comprehensible" manner [19], their tendency to "ruin everything" [9] and outright recommendations to "avoid nulls" [17]. The latter, however, is often not possible: in large volumes of data, incompleteness is hard to avoid.

Issues related to null handling stem from the fact that we do not naturally think in terms of a three-valued logic; rather we try to categorize facts as true or false. Once the third truth value – in the case of SQL, *unknown* – enters the picture, our usual logic often proves faulty leading to errors and unexpected behavior. We illustrate this by two commonly assumed query rewriting rules.

The first of the rules is the translation of **IN** subqueries into **EXISTS** queries, described in multiple textbooks. For example,

(Q_1): **SELECT** R.A **FROM** R **WHERE** R.A **NOT IN** (**SELECT** S.A **FROM** S)

would be translated into

(Q_2): **SELECT** R.A **FROM** R **WHERE NOT EXISTS** (**SELECT** S.A **FROM** S **WHERE** S.A=R.A)

(see e.g., [41] explaining in detail many translations presented in database texts). This, however, is not an equivalent rewriting: if $R = \{1, \textbf{NULL}\}$ and $S = \{\textbf{NULL}\}$ then the first query produces the empty table while the latter returns R itself. This presumed, but incorrect, equivalence is known to be a trap many SQL programmers are not aware of, see [7, 9].

Next, consider two queries presented as an illustration of the HoTTSQL prover for showing equivalences among queries [12]

(Q_3): **SELECT DISTINCT** X.A **FROM** R X, R Y **WHERE** X.A=Y.A
(Q_4): **SELECT DISTINCT** R.A **FROM** R

While claimed to be equivalent in [12], Q_3 and Q_4 are different: if $R = \{\textbf{NULL}\}$ then Q_3 returns no rows while Q_4 returns a single row with a **NULL** in it. In fairness to [12], it does not consider databases with nulls, but it is illustrative nonetheless that an "easy" equivalence example they chose is that of two non-equivalent queries on the simplest possible database containing **NULL**.

Over the years two lines of thought emerged for dealing with these problems. One is to provide a more complex logic for handling nulls, accounting for more varied types of those than SQL presents [8, 13, 18, 24, 34, 44]. These however did not take off, as the logic is even harder for the programmer. An alternative is to produce a language with no nulls at all, and thus resort to the usual two-valued logic. This proposal found more success, for example in the "3rd manifesto" [16] and the Tutorial D language, and in the LogicBlox system [3] which used the 6th normal form to eliminate nulls. But nulls do occur in many scenarios and need to be handled; the world is not yet ready to dismiss them altogether.

What is missing in this picture is a different line of thought: namely, a language that handles nulls but in doing so, uses the familiar two-valued Boolean logic, rather than a many-valued logic. In this proof-of-concept paper we show that SQL indeed could have been designed along these lines. To have a language that uses nulls and handles them with the familiar two-valued logic, we would need to fulfill the following criteria.

, ,

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/...

(1) On databases without nulls queries would be written exactly as before, and return the same results (*do not make changes unless necessary*).
(2) The version of SQL with nulls and two-valued logic will have exactly the same expressiveness as its version based on 3VL (*do not lose any queries; do not invent new ones*).
(3) For each query currently expressible in SQL, the size of the equivalent query in the two-valued language should be of the same order, e.g., at most linear (*do not make queries overly complicated*).

Why do we think this is achievable? After all, many years of SQL practice taught generations of programmers that one needs a 3VL to handle nulls. The main reason to believe that this is not so is two recent results, that made steps in the right direction, albeit for simpler languages. First, [28] showed that in the most basic fragment of SQL corresponding to relational algebra (selection-projection-join-union-difference), the truth value *unknown* can be eliminated from conditions in **WHERE**. Essentially, it rewrote conditions by adding **IS NULL** or **IS NOT NULL**, in a way that they could ever evaluate to *unknown*. Following that, [15] considered many-valued first-order predicate calculi under set semantics, and showed that no many-valued logic gives us extra power over the Boolean logic.

Our goal is to see if these purely theoretical results are applicable to the language that programmers actually use, SQL[1]. We start by looking at the core of it formalized in the 1992 version of the Standard, that includes the following features:

- full relational algebra;
- arithmetic functions and comparison predicates ($+, \cdot, \leq$ etc.);
- aggregate functions and **GROUP BY**;
- comparisons involving aggregates (**HAVING**);
- comparisons involving subqueries (**IN**, **EXISTS**, **ALL**, **ANY**);
- set operations (**UNION**, **INTERSECT**, **EXCEPT**, with or without the **ALL** keyword);
- conditional statements.

A significant extension of the 1999 version of the Standard, also known as SQL3, added recursion in the form of

- **WITH RECURSIVE** clause.

Our main result is as follows: *for these languages – SQL 1992 or SQL 1999 – our key goals 1–3 above are achievable, and SQL's 3VL can be eliminated in favor of the usual Boolean logic.*

The core idea To explain it, consider where *unknown* appears in SQL query evaluation. This happens when one evaluates a predicate, such as R.A=S.A, and one or more arguments are **NULL**. Thus, if we were to move to the Boolean logic, a minimal change is to assigned one of the Boolean truth values to such comparisons. And SQL already does so, in a way. In fact, SQL conflates *unknown* with *false* upon exiting the **WHERE** clause. Indeed, only tuples for which the condition in **WHERE** is *true* are selected, and thus at the end of evaluating the condition, *unknown* is merged with *false*; in other words, 3VL only exists while the **WHERE** clause is evaluated, and afterwards it is all back to the standard Boolean logic.

[1] A note from the authors to the reviewers: this is the reason we chose category 3 in the rather loose classification of papers into three categories, that is likely to be revamped or abolished soon.

Our main result says that changing the evaluation of conditions in this way leads to a two-valued version of SQL that satisfies our desiderata. Specifically, a language with the support for the usual relational algebra operators, plus aggregation, plus recursion, remains equally expressive to SQL based on the three-valued logic, the conversions of queries are easy, and none are necessary on databases without nulls.

We go even further and prove a number of extensions of this result. First, we show that other ways of changing the evaluation of conditions give us equally expressive languages. One of them is of particular note as it is used in SQL. This is the *syntactic equality*, in which **NULL** = **NULL** results in *true*. It is used in in set operations and grouping. For example, $\{1, \textbf{NULL}\}$ **EXCEPT** $\{1\}$ produces $\{\textbf{NULL}\}$, and on a relation $T = \{(\textbf{NULL}, 2), (\textbf{NULL}, 3)\}$, the query **SELECT** A, **SUM**(B) **FROM** T **GROUP BY** A results in $\{(\textbf{NULL}, 5)\}$, applying the syntactic equality to nulls.

Another result that we prove stems from the question whether a different many-valued logic could have given us a more expressive version of SQL. We provide a negative answer to this, further strengthening the argument for using the familiar Boolean logic for handling nulls.

Applicability of the results While the main conclusion is that SQL *could have been designed* without the recourse to a many-valued logic, we would like to use it as a proof of concept showing that:

- future languages can be designed using the familiar two-valued logic; and
- they need not do it by eliminating nulls altogether.

There are multiple languages under design at all times, SQL itself included as it constantly changes and each release of the Standard adds features. There is much activity in the field of graph databases [2, 20, 23, 42] with a new unifying standard called GQL emerging [43]. This could be a good testbed, as well as addition of nulls to languages that deliberately omitted them in order to avoid the abnormalities of the three-valued logic [3, 16].

Crucially, for RDBMSs, the changes we propose do not necessitate changes to the underlying implementation. A user can write a query under a two-valued Boolean semantics. Then it is translated into an equivalent query under the standard SQL semantics, which any of the existing engines can evaluate. The translated query itself only marginally exceeds the size of the original in the worst case, and in many cases, as we shall see, no changes are even needed.

Another potential application is in the design and verification of query optimizers. As already mentioned, 3VL invalidates optimization rules that one takes for granted in research papers. Thus, one needs to build tools for verifying actual optimizers, to ensure their rules are correct. This need is well recognized [11, 12]. However, most existing verification tools are based on Boolean logic, and it thus appears that such tools would be better suited for verifying a query language itself based on a two-valued logic. In addition, we shall see that two-valued languages are actually better behaved in terms of query equivalences: some "false equivalences" (i.e., those true only on databases without nulls) become true on all databases when we pass from 3VL to two-valued logic. This is an additional argument for using the logic programmers and DBMS implementors are more familiar with.

The choice of language We need to prove results formally, and thus we need a language as closely resembling SQL as possible and yet having a formal compositional semantics one can reason about. For this purpose, we choose an extended relational algebra that resembles very much the algebra into which SQL is translated into in RDBMS implementations. It expands the standard textbook operations of relational algebra with several features. First, it is interpreted under bag semantics, and duplicate elimination is added. Second, selection conditions are expanded significantly. They introduce testing for nulls, use SQL 3VL and SQL's rules for the introduction of *unknown*, and have conditions of the form $\bar{t} \in E$ and $\texttt{empty}(E)$ for directly encoding **IN** and **EXISTS** subqueries, as well as conditions $t = \texttt{any}(E)$ and $t = \texttt{all}(E)$ (and likewise for other comparisons) for encoding **ANY** and **ALL** subqueries. Third, the algebra has aggregate functions and grouping operation. Fourth, it allows function application to mimic expressions in the **SELECT** clauses. And finally, it has an iteration operation with the semantics of SQL recursive queries. For someone familiar with SQL, it should be clear that the language faithfully captures SQL's semantics while allowing us to prove results formally.

Related work The idea of using Boolean logic for nulls predates SQL; it actually appeared in QUEL [39] (see details in the latest manual [32]). Afterwards however the main direction was in making the logic of nulls more rather than less complicated, with proposals ranging from three to six values [13, 18, 24, 34, 44] or producing more complex classifications of nulls, e.g., [8, 45]. Elaborate many-valued logics for handling incomplete and inconsistent data were also considered in AI literature, see for example [4, 22, 25]. Proposals for eliminating nulls have appeared in [3, 16].

There is a large body of work on achieving correctness of query results on databases with nulls where correctness assumes the standard notion of certain answers [31]. Among such works are [21, 26, 27, 35]. They assumed either SQL's 3VL, or the Boolean logic of marked nulls [31], and showed how query evaluation could be modified to achieve correctness, but they did not question the underlying logic of nulls. To the contrary, we are concerned with finding a logic that makes it more natural for programmer to write queries; once this is achieved, one will need to modify evaluation schemes to produce subsets of certain answers if one so desires. For connection between theoretical models such as marked or Codd nulls used in much of such work and real SQL nulls, see [29].

Some papers looked into handling nulls and incomplete data in bag-based data models as employed by SQL [14, 30, 37] but none concentrated on the underlying logic of nulls.

Organization In Section 2 we present the syntax and the semantics of the language. In Section 3 we explain how to eliminate *unknown* to achieve our desiderata. Section 4 looks into other two-valued semantics, while Section 5 shows that no other many-valued logic could have achieved additional expressiveness. Conclusions are in Section 6. Complete proofs are in the appendix.

2 QUERY LANGUAGE: $\mathrm{RA}_{\mathrm{SQL}}$

We now settle on the language. Given the idiosyncrasies of SQL's syntax, it is not the ideal language – syntactically – to reason about. We know however that its queries are all translatable into an extended relational algebra; indeed, this is what is done inside every RDBMS, and multiple such translations are described in the literature [5, 10, 28, 36, 41].

Thus, we shall work with relational algebra, but not the textbook version of it. Rather we look at the version of the language called $\mathrm{RA}_{\mathrm{SQL}}$ that is very close to what real-life SQL queries are translated into. In particular it will be a typed relational algebra, as we need to distinguish numerical attributes over which aggregation is performed. It will include constructs for grouping and computing aggregates, as well as comparing aggregates. Its conditions will include **IN** and **EXISTS** for direct expression of subqueries rather than translating them via joins. This will cover the essential SQL 1992 features. Then we add an iteration operation that works in the same way as SQL's **WITH RECURSIVE**, added in SQL 1999.

2.1 Data Model

The usual presentation of relational algebra assumes a countably infinite domain of values. Since we handle languages with aggregations, we need to distinguish columns of numerical types. As not to over-complicate the model, we assume two types: a numerical and non-numerical one (we call it the *ordinary* type as it corresponds to the presentation of relational algebra one ordinarily finds in textbooks). This will be without any loss of generality as the treatment of nulls as values of all types is the same, except numerical as nulls behave differently with respect to aggregation.

Towards that goal, we assume the following pairwise disjoint countable infinite sets:

- Name of attribute *names*, and
- Num of *numerical values*, and
- Val of *(ordinary) values*.

Each of these has a *type* whose value is either o (ordinary) or n (numerical). If $N \in \mathrm{Name}$, then $\mathsf{type}(N)$ indicates whether columns contain elements of ordinary type or numerical type; one can think of this as the usual declarations in **CREATE TABLE** statements in this simple type system. Furthermore, $\mathsf{type}(e) = \mathsf{n}$ if $e \in \mathrm{Num}$ and $\mathsf{type}(e) = \mathsf{o}$ if $e \in \mathrm{Val}$.

We use the fresh symbol **NULL** to denote the null value.

Typed records and relations are defined as follows. Let $\tau := \tau_1 \cdots \tau_n$ be a word over the alphabet $\{\mathsf{o}, \mathsf{n}\}$. A *$\tau$-record* $\bar{a}$ with *arity* n is a tuple $(a_1, \cdots, a_n)$ where

- $a_i \in \mathrm{Num} \cup \{\textbf{NULL}\}$ whenever $\tau_i = \mathsf{n}$, and
- $a_i \in \mathrm{Val} \cup \{\textbf{NULL}\}$ otherwise (whenever $\tau_i = \mathsf{o}$).

Each n-ary relation symbol R in the schema has an associated sequence $\ell(R) = N_1 \cdots N_n \in \mathrm{Name}^n$ of its attribute names. The *type* of R is then the sequence $\mathsf{type}(R) = \mathsf{type}(N_1) \cdots \mathsf{type}(N_n)$. A *relation* R over R is then a *bag* of $\mathsf{type}(R)$-records, i.e., records compatible with the type of R. As in SQL, we use bag semantics, i.e., a record may appear more than once in a relation. We also speak of bags of τ-records as τ-relations.

For a τ-relation R write $\bar{a} \in_k \mathrm{R}$ if $\bar{a}$ occurs k times in R. In particular, $\bar{a} \in_0 \mathrm{R}$ means that $\bar{a}$ does not occur in $\mathbf{R}$.

A *relation schema* $\mathcal{S}$ is a set of relation symbols and their types, i.e., a set of pairs $(R, \mathsf{type}(R))$. A *database* D over a relation schema $\mathcal{S}$ associate with each $(R, \mathsf{type}(R)) \in \mathcal{S}$ a relation of $\mathsf{type}(R)$-records.

The *duplicate eliminating operator* $\varepsilon(\mathrm{R})$ turns R into *set* that contains every τ-record in R once. Formally, $\bar{a} \in \varepsilon(\mathrm{R})$ iff $\bar{a} \in_k \mathrm{R}$ for some $k > 0$. The *cardinality* $\mathsf{Card}(\mathrm{R})$ of R as the sum of

Terms

$$t := n \mid c \mid \textbf{NULL} \mid N \mid f(t_1, \cdots, t_k) \quad n \in Num,\ c \in \mathrm{Val},\ N \in \mathrm{Name},\ f \in \Omega$$

Expressions

$E :=$	R	(base relation)
	$\pi_{t_1[\to N'_1], \cdots, t_m[\to N'_m]}(E)$	(generalized projection with optional renaming)
	$\sigma_\theta(E)$	(selection)
	$E \times E$	(product)
	$E \cup E$	(union)
	$E \cap E$	(intersection)
	$E - E$	(difference)
	$\varepsilon(E)$	(duplicate elimination)
	$\mathtt{Group}_{\bar N}\langle F_1(N_1)[\to N'_1], \cdots, F_m(N_m)[\to N'_m]\rangle(E)$	(grouping/aggregation with optional renaming)

Atomic conditions

$$ac := \mathbf{t} \mid \mathbf{f} \mid \mathtt{isnull}(t) \mid \bar t \doteq \bar t' \mid \bar t \in E \mid \mathtt{empty}(E) \mid P(\bar t) \mid t\ \omega\ \mathtt{any}(E) \mid t\ \omega\ \mathtt{all}(E)$$

$$P \in \Omega \quad \omega \in \{=, \neq, <, >, \leq, \geq\}$$

Conditions

$$\theta := ac \mid \theta \vee \theta \mid \neg\theta \mid \theta \wedge \theta$$

Figure 1: Syntax of $\mathrm{RA}_{\mathrm{SQL}}$

the number of occurrences of different τ-records in it. Formally, $\mathsf{Card}(\mathrm{R}) := \sum_{\bar a \in_k \mathrm{R}} k$. In what follows, we omit the τ from τ-record or τ-relation if it is not significant or clear from the context.

Since we are dealing with bags rather than sets, we interpret the operators union $\cup$, intersection $\cap$, difference $-$, and Cartesian product $\times$ with the standard bag semantics:

- Union: $\bar a \in_k \mathbf{R} \cup \mathbf{S}$ iff $\bar a \in_n \mathbf{R}$ and $\bar a \in_m \mathbf{S}$ and $k = n + m$;
- Intersection: $\bar a \in_k \mathbf{R} \cap \mathbf{S}$ iff $\bar a \in_n \mathbf{R}$ and $\bar a \in_m \mathbf{S}$ and $k = \min(n, m)$;
- Difference: $\bar a \in_k \mathbf{R} - \mathbf{S}$ iff $\bar a \in_n \mathbf{R}$ and $\bar a \in_m \mathbf{S}$ and $k = \max(n - m, 0)$;
- Cartesian Product: $(\bar a, \bar b) \in_k \mathbf{R} \times \mathbf{S}$ iff $\bar a \in_n \mathbf{R}$ and $\bar b \in_m \mathbf{S}$ and $k = n \cdot m$.

Note that the first three correspond to SQL's **UNION ALL**, **INTERSECT ALL**, and **EXCEPT ALL**; without **ALL**, these are followed by applying duplicate elimination.

2.2 Syntax

To define the syntax of our relational algebra, we define a *term* as either a numerical value in Num, or an ordinary value in Val, or **NULL**, or a name in Name, or an element of the form $f(t_1, \cdots, t_k)$ where f is a *k-ary numerical function*, that is, $f : \mathrm{Num}^k \to \mathrm{Num}$, and $t_1, \cdots, t_k$ are terms. For example, addition and multiplication are binary numerical functions. As we shall see in the description of the semantics, it will be well-defined if the argument terms $t_1, \cdots, t_k$ evaluate to values of the numerical type.

A k-ary *numerical predicate* is a relation symbol whose type is n^k for some positive integer k. For example, $\leq$ is a binary numerical predicate. An *aggregate function* F is a function F that maps bags of numerical values into a numerical value (i.e., it maps bags whose elements are from Num into a single element in Num). For example, SQL's aggregates **COUNT**, **AVG**, **SUM**, **MIN**, **MAX** are such.

Relational algebra considered here is parameterized by a collection Ω of numerical predicates, functions, and aggregate functions. We assume that the standard comparison predicates $=, \neq, <, >, \leq, \geq$ are always present over numbers. Our results on translation will be true for every possible collection of predicates and functions.

Given a schema $\mathcal{S}$ and such a collection Ω, the syntax of $\mathrm{RA}_{\mathrm{SQL}}$ expressions and conditions over $\mathcal{S} \cup \Omega$ is given in Fig. 1, where each t_i is a term, each N_i, N'_i is a name, each $\bar N$ is a tuple of names, and each F_i is an aggregate function. In the generalized projection and in the grouping/aggregation, the parts in the squared brackets (i.e., $[\to N_i]$ and $[\to N'_i]$) are optional renamings.

The *size* of an expression is defined in the standard way as the size of its parse tree.

In what follows, we restrict our attention to expressions with well-defined semantics (e.g., we forbid aggregation over non-numerical columns or functions applied to arguments of wrong types). The next two sections present the semantics: first at the intuitive level, and then formally.

2.3 Informal Semantics

We now offer an informal explanations of the semantics, with the formal semantics presented in the next section. In $\mathrm{RA}_{\mathrm{SQL}}$ expressions, R ranges over relation symbols in $\mathcal{S}$.

Terms are either constants of numerical or ordinary type, or **NULL**, or an attribute name, or function application. For example, A, 2 are terms as are A+2 and A*2.

Generalized projection corresponds to SQL's **SELECT** clause. In generalized projections, each term t_i is evaluated and added as a column to the result. Such terms may refer to names from Name that are among attributes of the result of the expression E. Optional renaming allows us to rename such columns, simulating **AS** in SQL. To see a concrete example, to express

SELECT A , B , A + 2 , A * B **FROM** R

for a relation R with attributes A, B we would use

$$\pi_{A,B,\mathsf{add2}(A),\mathsf{mult}(A,B)}(R)$$

where $\mathsf{add2}(x) = x + 2$ and $\mathsf{mult}(x, y) = x \cdot y$.

Names of columns are unique and can be specified explicitly by N_i in case the optional $[\to N_i]$ part appears. The content of these square brackets corresponds to whats comes right after SQL's renaming key word **AS**. (Names cam also be derived implicitly by the function `Name` that we discuss later.) For example

SELECT A, B, A+2 **AS** C, A*B **AS** D **FROM** R

would be translated into

$$\pi_{A,B,\mathsf{add2}(A)\to C,\mathsf{mult}(A,B)\to D}(R)$$

Projection follows SQL's bag semantics. That is, for tuple $(a_1, \cdots, a_n)$ in the result of E, it computes the values of terms $t_1, \cdots, t_m$ and outputs them as values of (optionally renamed) attributes.

Selection, as usual, evaluates the condition θ for each tuple, and only keeps tuples for which the condition is *true* (i.e., not *false* or *unknown*). Operations of generalized projection and selection correspond to sequential scans in query plans (with filtering in the case of selection).

Other operations have the standard meaning under the bag semantics: for union, intersection, difference, and Cartesian product, it was described above. The operation ε eliminates duplicates and keeps one copy of each record.

We follow SQL's semantics of functions: if one of its arguments is **NULL**, then the result is null. For example, $3 + 2$ gives 5, but **NULL** $+ 2$ gives **NULL**.

Before looking at grouping/aggregation, we deal with the conditions. For each predicate $P \in \Omega$ we assume its meaning is well defined when its arguments are not **NULL** (e.g., $\leq$ on numbers). Then this is the meaning that is used when all arguments are not **NULL**, and if one is **NULL**, then the value is unknown (**u**). A special case of this is equality, in fact equality of tuples of terms $(t_1, \ldots, t_m) \doteq (t'_1, \ldots, t'_m)$, which is the conjunction of $t_i \doteq t'_i$ for all $i \leq m$. The condition $\mathtt{isnull}(t)$ tests if the value of term t is **NULL**.

The condition $\bar{t} \in E$, not typically included in relational algebra, tests whether a tuple belongs to the result of a query, and corresponds to SQL's **IN** subqueries. The condition $\mathtt{empty}(E)$ checks if the result of E is empty, and corresponds to SQL's **EXISTS** subqueries.

One might be tempted to say that these are expressible via joins in traditional relational algebra. There is a good reason to include them directly. First, we want to stay as close to SQL as possible. Even more importantly, these conditions behave *differently* in the presence of nulls, and their expressibility via joins would require complex conditions checking which attributes values are **NULL**. Indeed, as we shall see, they behave differently with respect to treatment of nulls; in fact **EXISTS** subqueries follow the two-valued logic, which **IN** subqueries are based on the three-valued logic.

Other predicates not typically included in relational algebra presentation, though included here for direct correspondence with SQL, are **ALL** and **ANY** comparisons. Let E be an expression that returns a table with a single numerical column, t a term, and ω a comparison. Then $t\ \omega\ \mathtt{any}(E)$ means that there exists a value t' in E so that $t\ \omega\ t'$ holds, and $t\ \omega\ \mathtt{all}(E)$ means that $t\ \omega\ t'$ holds for each value t' in E (in particular, if E returns no tuples, this condition is true). If ω is $=$ or $\neq$, conditions with `any` and `all` are applicable at either ordinary or numerical type; if ω is one of $<, \leq, >, \geq$, then t and E must be of numerical type.

Finally, we describe the operator $\mathtt{Group}_{\bar{N}}\langle F_1(N_1), \cdots, F_m(N_m)\rangle(E)$. The tuple $\bar{N}$ lists attributes in **GROUP BY**, $F_i(N_i)$ are aggregate functions F_i over numerical columns N_i possibly named N'_i (when $[\to N'_i]$ is specified). For example, to express

SELECT A, **COUNT**(B) **AS** C, **SUM**(B) **FROM** R **GROUP BY** A

we would use

$$\mathtt{Group}_A\langle F_{\mathsf{count}}(B)[\to C], F_{\mathsf{sum}}(B)\rangle(R)$$

where $F_{\mathsf{count}}(\{a_1, \ldots, a_n\}) = n$ and $F_{\mathsf{sum}}(\{a_1, \ldots, a_n\}) = a_1 + \cdots + a_n$. Note that $\bar{N}$ could be empty; this corresponds to computing aggregates over the entire table, without grouping, for example, as in **SELECT COUNT** (B), **SUM** (B) **FROM** R.

Example 1. We start by showing how queries Q_1–Q_4 from the introduction are expressible in $\mathrm{RA}_{\mathrm{SQL}}$:

$$\begin{array}{lcl} Q_1 & = & \sigma_{\neg(R.A\in S)}(R) \\ Q_2 & = & \sigma_{\mathtt{empty}(\sigma_{R.A=S.A}(S))}(R) \\ Q_3 & = & \varepsilon\Big(\pi_{X.A}(\sigma_{X.A=Y.A}(\rho_{R.A\to X.A}(R) \times \rho_{R.A\to Y.A}(R)))\Big) \\ Q_4 & = & \varepsilon(\pi_{R.A\to X.A}(R)) \end{array}$$

As a more complex example we look at query Q_5, which is a slightly simplified (to fit in one column) query 22 from the TPC-H benchmark [40]:

```
SELECT c_nationkey, COUNT(c_custkey)
FROM customer
WHERE c_acctbal >
(SELECT avg(c_acctbal)
FROM customer WHERE c_acctbal > 0.0 AND
c_custkey NOT IN (SELECT o_custkey FROM orders) )
GROUP BY c_nationkey
```

In translations below, we use abbreviations C for `customer` and O for `orders`, and abbreviations for attributes like c_n for `c_nationkey` etc. The **NOT IN** condition in the subquery is then translated as $\neg(c\_c \in \pi_{o\_c}(O))$, the whole condition is translated as $\theta := (c\_a > 0) \wedge \neg(c\_c \in \pi_{o\_c}(O))$ and the aggregate subquery becomes

$$Q_{agg} = \mathtt{Group}_{\emptyset}\langle F_{\mathsf{avg}}(c\_a)\rangle\Big(\pi_{c\_a}(\sigma_{\theta}(C))\Big).$$

Notice that there is no grouping for this aggregate, hence the empty set of grouping attributes. Then the condition in the **WHERE** clause of the query is Next condition $\theta' := c\_a > \mathtt{any}(Q_{agg})$ which is then applied to C, i.e., $\sigma_{c\_a>\mathtt{any}(Q_{agg})}(C)$, and finally grouping by c_n and counting of c_a are performed over it, giving us

$$\mathtt{Group}_{c\_n}\langle F_{\mathsf{count}}(c\_c)\rangle(\sigma_{c\_a>\mathtt{any}(Q_{agg})}(C)).$$

Putting everything together, we arrive at the final translation into $\mathrm{RA}_{\mathrm{SQL}}$:

$$\mathsf{Group}_{c\_n}\langle F_{\mathsf{count}}(c\_c)\rangle\Big(\sigma_{c\_a>\mathtt{any}\Big(\mathsf{Group}_{\emptyset}\langle F_{\mathsf{avg}}(c\_a)\rangle\big(\pi_{c\_a}(\sigma_{\theta}(C))\big)\Big)}(C)\Big).$$

2.4 Formal Semantics

We next define the formal semantics of $\mathrm{RA}_{\mathrm{SQL}}$ expressions. This is done in the spirit of [5, 12, 28] and is necessary for formally proving the results about eliminating three-valued logic. That said, the reader who wants to understand the result and rely on the informal explanation of the semantics given in the previous section can skip these technical details.

We define the semantic function

$$[\![E]\!]_{D,\eta}$$

which is the result of evaluation of expression E on database D under the *environment* η. The environment provides values of parameters of the query. Indeed, to give a semantics of a query with subqueries, we need to define the semantics of subqueries as well. Consider for example a subquery **SELECT** S.A **FROM** S **WHERE** S.A=R.A of query Q_2 from the Introduction. Here R.A is a parameter, and to compute the query we need to provide its value. Thus, an environment η is a partial mapping from the set Name of names to the union $\mathrm{Val} \cup \mathrm{Num} \cup \{\mathbf{NULL}\}$.

Recall that every relation R is associated with a sequence $\ell(R)$ of attribute names. Just as SQL queries do, every $\mathrm{RA}_{\mathrm{SQL}}$ expression E produces a table whose attribute similarly have names. We start by defining those in Fig. 2a. We make the assumption that names do not repeat; this is easy to enforce with renaming. This differs from SQL where names in query results can repeat, and this point was rather extensively discussed in [28]. However, the treatment of repeated names in the definition of the semantics of SQL queries is completely orthogonal to the treatment of nulls, and thus we can make this assumption without loss of generality so as not to clutter the description of our translations with the complexities coming from treating repeated attributes.

Next, we define the semantics of terms: it is given by the environment, see Fig. 2b.

After that we give the semantics of predicates $P \in \Omega$ and equality in Fig. 2c. We follow SQL's three valued logic with true value *true* (**t**), *false* (**f**) and *unknown* (**u**). The usual SQL's rule is: evaluate a predicate normally if no attributes are nulls; otherwise return **u**. That is, for each predicate P such as $<$, we have its interpretation P over $\mathrm{Val} \cup \mathrm{Num}$.

To provide the formal semantics of $\mathrm{RA}_{\mathrm{SQL}}$ expressions, we need some extra notation. We assume that there is a one-to-one function Name that maps terms into (unique) names (i.e., elements in Name). Given $\alpha \in \mathrm{Name}^*$ and $\bar{N}, \bar{N}' \in \mathrm{Name}^*$, the sequence $\alpha_{\bar{N}\to\bar{N}'}$ obtained from α by replacing each N_i with N'_i where $\bar{N} := (N_1, \cdots, N_m)$ and $\bar{N}' := (N'_1, \cdots, N'_m)$.

Next, if $\bar{a} := (a_1, \cdots, a_m)$ is a tuple of values over $\mathrm{Num} \cup \mathrm{Val} \cup \{\mathbf{NULL}\}$ and $\bar{N} := (N_1, \cdots, N_m)$ a tuple of names over Name, we denote by $\eta^{\bar{a}}_{\bar{N}}$ the environment that maps each name N_i into the value a_i. We say that $\bar{a}$ is *consistent* with $\bar{N}$ if $\mathsf{type}(N_i) = \mathsf{n}$ implies $a_i \in \mathrm{Num} \cup \{\mathbf{NULL}\}$ and $\mathsf{type}(N_i) = \mathsf{o}$ implies $a_i \in \mathrm{Val} \cup \{\mathbf{NULL}\}$ for each i. For two environments η and η', by $\eta;\eta'$ we mean η overridden by η'. That is, $\eta;\eta'(N) = \eta(N)$ if η is defined on A and η' is not; otherwise $\eta;\eta'(N) = \eta'(N)$.

For a bag B, let $B_{\neq\mathbf{NULL}}$ be the same as B but with occurrences of **NULL** removed. A tuple is called *null-free* if none of its components is **NULL**.

With these, the semantics of expressions is defined in Fig. 2e. Note that we omit the optional parts in the generalized projection and grouping/aggregation as it do not affect the semantics but is reflected only in the names as appear in Figure 2a.

Now given an expression E of $\mathrm{RA}_{\mathrm{SQL}}$ and a database D, the value of E in D is defined as $[\![E]\!]_{D,\emptyset}$ where $\emptyset$ is the empty mapping (i.e., the top level expression has no free variables).

2.5 Recursive queries

We now incorporate recursive queries, a feature added in the SQL 1999 standard. Extensions of relational algebra with various kinds of recursion exists (e.g., with transitive closure [1] or fixed-point operator [33]). We follow the same approach although stay closer to SQL as it is, which uses a special type of iteration – in fact two kinds depending on the syntactic shape of the query (see [38] as well as explanation below) – to define recursive queries.

Syntax of RA^{REC}_{SQL}. Recall that $\cup$ stands for bag union, i.e., multiplicities of tuples are added up, as in SQL's **UNION ALL**. We also need the operation $B_1 \sqcup B_2$ defined as $\varepsilon(B_1 \cup B_2)$, i.e., union in which a single copy of each tuple is kept. This corresponds to SQL's **UNION**.

An $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$ expression is defined with the grammar of $\mathrm{RA}_{\mathrm{SQL}}$ in Fig. 1) with the addition of the following constructor:

$$\mu R.E$$

where R is a fresh relation symbol (i.e., not in the schema) and E is an expression of the form $E_1 \cup E_2$ or $E_1 \sqcup E_2$ where both E_1 and E_2 are $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$ expressions and E_2 may contain a reference to R.

Note that in SQL, various restrictions are imposed on query E_2. These typically include linearity of recursion (at most one reference to R within E_2, restrictions on the use of recursively defined relation in subqueries, restrictions on the use of aggregation, etc.) The reason for such restrictions is to eliminate some of the common cases of non-terminating queries.

We shall not impose such restrictions, as our result is more general: passing from 3VL to two-valued logic is possible even if such restrictions were not in place. Note that different RDBMSs use different restrictions on recursive queries (and sometimes even different syntax); hence showing this more general result will ensure that it applies to all of them.

Semantics of RA^{REC}_{SQL}. Similarly to the syntactic definition, we distinguish between the two cases.

For $\mu R.E_1 \cup E_2$, the semantics $[\![\mu R.E_1 \cup E_2]\!]_{D,\eta}$ is defined by the following iterative process:

(1) $RES_0, R_0 := [\![E_1]\!]_{D,\eta}$
(2) $R_{i+1} := [\![E_2]\!]_{D\cup R_i,\eta}$, $\quad RES_{i+1} := RES_i \cup R_{i+1}$

with the condition that if $R_i = \emptyset$, then the iteration stops and RES_i is returned.

$$\ell\left(\pi_{t_1[\to N_1],\cdots,t_m[\to N_m]}(E)\right) := \tilde{N}_1 \cdots \tilde{N}_m$$
$$\text{where } \tilde{N}_i := \begin{cases} N_i & \text{if } [\to N_i] \\ \texttt{Name}(t_i) & \text{otherwise} \end{cases}$$
$$\ell\left(\sigma_\theta(E)\right) := \ell(E)$$
$$\ell\left(E_1 \times E_2\right) := \ell(E_1) \cdot \ell(E_2)$$
$$\ell\left(E_1 \,\mathsf{op}\, E_2\right) := \ell(E_1) \text{ for } \mathsf{op} \in \{\cup, \cap, -\}$$
$$\ell\left(\varepsilon(E)\right) := \ell(E)$$
$$\ell(\texttt{Group}_{\bar{N}}\langle F_1(N_1)[\to N_1'], \cdots, F_m(N_m)[\to N_m']\rangle(E)) := \bar{N} \cdot \tilde{N}_1 \cdots \tilde{N}_m$$
$$\text{where } \tilde{N}_i := \begin{cases} N_i & \text{if } [\to N_i] \\ \texttt{Name}(F_i(N_i)) & \text{otherwise} \end{cases}$$

(a) Names

$$[\![t]\!]_\eta := \begin{cases} \eta(t) & t \in \text{Name} \\ t & t \in \text{Val} \cup \text{Num} \cup \{\textbf{NULL}\} \end{cases}$$
$$[\![(t_1, \cdots, t_m)]\!]_\eta := ([\![t_1]\!]_\eta, \cdots, [\![t_m]\!]_\eta)$$
$$[\![f(t_1, \cdots, t_m)]\!]_\eta := \begin{cases} \textbf{NULL} & \exists i : [\![t_i]\!]_\eta = \textbf{NULL} \\ f([\![t_1]\!]_\eta, \cdots, [\![t_m]\!]_\eta) & \text{otherwise} \end{cases}$$

(b) Terms

$$[\![P(t_1, \cdots, t_n)]\!]_{D,\eta} := \begin{cases} \mathbf{t} & \forall i : [\![t_i]\!]_\eta \neq \textbf{NULL}, [\![(t_1, \cdots, t_n)]\!]_\eta \in P \\ \mathbf{f} & \forall i : [\![t_i]\!]_\eta \neq \textbf{NULL}, [\![(t_1, \cdots, t_n)]\!]_\eta \notin P \\ \mathbf{u} & \exists i : [\![t_i]\!]_\eta = \textbf{NULL} \end{cases}$$
$$[\![t = t']\!]_{D,\eta} := \begin{cases} \mathbf{t} & [\![t]\!]_\eta, [\![t']\!]_\eta \neq \textbf{NULL}, [\![t]\!]_\eta = [\![t']\!]_\eta \\ \mathbf{f} & [\![t]\!]_\eta, [\![t']\!]_\eta \neq \textbf{NULL}, [\![t]\!]_\eta \neq [\![t']\!]_\eta \\ \mathbf{u} & [\![t]\!]_\eta = \textbf{NULL} \text{ or } [\![t']\!]_\eta = \textbf{NULL} \end{cases}$$

(c) Predicates

Basic conditions

$$[\![\mathbf{t}]\!]_{D,\eta} := \mathbf{t} \quad [\![\mathbf{f}]\!]_{D,\eta} := \mathbf{f}$$
$$[\![\texttt{isnull}(t)]\!]_{D,\eta} := \begin{cases} \mathbf{t} & [\![t]\!]_\eta = \textbf{NULL} \\ \mathbf{f} & \text{otherwise} \end{cases}$$
$$[\![\bar{t} \doteq \bar{t}']\!]_{D,\eta} := \bigwedge_{i=1}^{n} [\![t_i \doteq t_i']\!]_\eta$$
$$\text{where } \bar{t} := (t_1, \ldots, t_n)$$
$$\bar{t}' := (t_1', \ldots, t_n')$$
$$[\![\bar{t} \in E]\!]_{D,\eta} := \bigvee_{\bar{t}' \in [\![E]\!]_{D,\eta}} [\![\bar{t} \doteq \bar{t}']\!]_\eta$$
$$[\![t\, \omega\, \texttt{any}(E)]\!]_{D,\eta} := \bigvee_{t' \in [\![E]\!]_{D,\eta}} [\![t\, \omega\, t']\!]_\eta$$
$$[\![t\, \omega\, \texttt{all}(E)]\!]_{D,\eta} := \bigwedge_{t' \in [\![E]\!]_{D,\eta}} [\![t\, \omega\, t']\!]_\eta$$
$$[\![\texttt{empty}(E)]\!]_{D,\eta} := \begin{cases} \mathbf{t} & [\![E]\!]_{D,\eta} = \emptyset \\ \mathbf{f} & \text{otherwise} \end{cases}$$

Composite conditions

$$[\![\theta_1 \vee \theta_2]\!]_{D,\eta} := [\![\theta_1]\!]_{D,\eta} \vee [\![\theta_2]\!]_{D,\eta}$$
$$[\![\theta_1 \wedge \theta_2]\!]_{D,\eta} := [\![\theta_1]\!]_{D,\eta} \wedge [\![\theta_2]\!]_{D,\eta}$$
$$[\![\neg\theta]\!]_{D,\eta} := \neg[\![\theta]\!]_{D,\eta}.$$

Three-valued logic rules

$\wedge$	**t**	**f**	**u**
t	**t**	**f**	**u**
f	**f**	**f**	**f**
u	**u**	**f**	**u**

$\vee$	**t**	**f**	**u**
t	**t**	**t**	**t**
f	**t**	**f**	**u**
u	**t**	**u**	**u**

	$\neg$
t	**f**
f	**t**
u	**u**

(d) Conditions

$$\bar{a} \in_k [\![R]\!]_{D,\eta} \text{ if } \bar{a} \in_k R^D$$
$$(a_1, \ldots, a_m) \in_k [\![\pi_{t_1,\cdots,t_m}(E)]\!]_{D,\eta} \text{ if } k = \sum_{j=1}^{n} k_j \text{ where } \bar{c}_j \in_{k_j} [\![E]\!]_{D,\eta} \text{ and for every } 1 \le i \le m\text{: } a_i = [\![t_i]\!]_{\eta;\eta_{\ell(E)}^{\bar{c}_j}}$$
$$[\![E_1 \,\mathsf{op}\, E_2]\!]_{D,\eta} := [\![E_1]\!]_{D,\eta} \,\mathsf{op}\, [\![E_2]\!]_{D,\eta} \text{ for } \mathsf{op} \in \{\cup, \cap, -, \times\}$$
$$\bar{a} \in_k [\![\sigma_\theta(E)]\!]_{D,\eta} \text{ if } \bar{a} \in_k [\![E]\!]_{D,\eta} \text{ and } [\![\theta]\!]_{D,\eta;\eta_{\ell(E)}^{\bar{a}}} = \mathbf{t}$$
$$\bar{a} \in_1 [\![\varepsilon(E)]\!]_{D,\eta} \text{ if } \bar{a} \in_k [\![E]\!]_{D,\eta} \text{ and } k > 0$$
$$(\bar{a}, \bar{a}') \in_1 [\![\texttt{Group}_{\bar{M}}\langle F_1(N_1), \cdots, F_m(N_m)\rangle(E)]\!]_{D,\eta} \text{ if } \bar{a} \in_k [\![\pi_{\bar{M}}(E)]\!]_{D,\eta}, \bar{a}' = ([\![\langle F_1(N_1)\rangle(E')]\!]_{D,\eta;\eta_{\ell(E)}^{\bar{a}}}, \cdots, [\![\langle F_m(N_m)\rangle(E')]\!]_{D,\eta;\eta_{\ell(E)}^{\bar{a}}})$$
$$\text{where } E' = [\![\pi_{N_1,\cdots,N_m}(\sigma_{\bar{M} \doteq \bar{a}}(E))]\!]_{D,\eta} \text{ and}$$
$$(\bar{a}, n) \in_k [\![\langle \texttt{Count}(\star)\rangle(E)]\!]_{D,\eta} \text{ if } \bar{a} \in_k [\![E]\!]_{D,\eta}, n = \mathsf{Card}([\![E]\!]_{D,\eta})$$
$$(\bar{a}, n) \in_k [\![\langle F(N)\rangle(E)]\!]_{D,\eta} \text{ if } \bar{a} \in_k [\![E]\!]_{D,\eta}, n = F\Big(([\![\pi_N(E)]\!]_{D,\eta})_{\neq \textbf{NULL}}\Big)$$

(e) Expressions

Figure 2: Names of Expressions and Semantics of Terms, Predicates, Conditions, and Expressions

For $\mu R.E_1 \sqcup E_2$, the semantics is defined by the following iterative process:

(1) $RES_0, R_0 := [\![\varepsilon(E_1)]\!]_{D,\eta}$
(2) $R_{i+1} := [\![\varepsilon(E_2)]\!]_{D \cup R_i,\eta} - RES_i, \quad RES_{i+1} := RES_i \cup R_{i+1}$

with the same stopping condition as before. Note that since R_{i+1} does not contain any tuples from RES_i, we have $RES_i \cup R_{i+1} = RES_i \sqcup R_i$ above. That is, either $\cup$ or $\sqcup$ could be used in rule (2) of this iterative process.

3 ELIMINATING UNKNOWN: CONFLATING UNKNOWN AND FALSE

To eliminate the use of SQL's three valued semantics, we need to replace the unknown truth value, and to do so, we need to see where this value arises. In SQL, *unknown* arises in the **WHERE** clause; in $\mathrm{RA}_{\mathrm{SQL}}$, in conditions. Specifically, it appears as the result of evaluation of predicates such as equality or $<$. It also arises in the evaluation of **IN** subqueries, but checking $\bar{t} \in E$ in $\mathrm{RA}_{\mathrm{SQL}}$ boils down to checking equalities, i.e., a disjunction of $\bar{t} \doteq \bar{t}'$ as $\bar{t}'$ ranges over tuples in E.

In applying basic predicates, *unknown* appears by following the rule that if one parameter is **NULL**, then the value of the predicate is **u**. Thus, we need to specify what we do in this case. SQL's existing features guide us in choosing our options. One is to *conflate* **u** *and* **f**, which is what is done after computing conditions in **WHERE**, as only those values for which the condition is **t** are kept. We shall now discuss this semantics, but our main result on the equivalence of query evaluation under 3VL and Boolean logic will extend not only to this but also to many other ways of eliminating unknown, see Section 4.

The new semantics, denoted by $[\![\,]\!]^{\mathbf{2VL}}$, replaces every case when **u** is produced by **f**. Since **u** only arises in evaluating predicates, it means that we have the following two rules:

$$[\![P(\bar{t})]\!]^{\mathbf{2VL}}_{D,\eta} := \begin{cases} \mathbf{t} & \forall i : [\![t_i]\!] \neq \textbf{NULL}, [\![(t_1, \cdots, t_n)]\!]_\eta \in \mathrm{P} \\ \mathbf{f} & \text{otherwise} \end{cases}$$

$$[\![t = t']\!]^{\mathbf{2VL}}_{D,\eta} := \begin{cases} \mathbf{t} & [\![t]\!]_\eta, [\![t']\!]_\eta \neq \textbf{NULL}, [\![t]\!]_\eta = [\![t']\!]_\eta \\ \mathbf{f} & \text{otherwise} \end{cases}$$

The rest of the semantics of expressions and conditions is exactly as in Fig. 2. Note that the truth value **u** never arises, and the rules of SQL's 3VL are exactly the rules of the Boolean two-valued logic when restricted to **t** and **f**.

Let us return to the example from the introduction of two queries Q_1 and Q_2 expressing difference of two relations R and S using **NOT IN** and **NOT EXISTS** subqueries. On database with $R = \{1, \textbf{NULL}\}$ and $S = \{\textbf{NULL}\}$ they produced different results: the empty table for Q_1 and $\{1, \textbf{NULL}\}$ for Q_2. Under the new $[\![\,]\!]^{\mathbf{2VL}}$ semantics, both Q_1 and Q_2 return the same answer $\{1, \textbf{NULL}\}$. The reason both elements 1 and **NULL** are returned is that all comparisons $1 \doteq \textbf{NULL}$ and $\textbf{NULL} \doteq \textbf{NULL}$ now evaluate to **f**, and then under negation **NOT** they become **t**, and hence both elements are returned. Previously, only the two-valued query with **NOT EXIST** (since **EXISTS** does not recourse to 3VL) did so.

3.1 Capturing SQL with $[\![\,]\!]^{\mathbf{2VL}}$

We now show that the two-valued semantics based on conflating **u** with **f** fulfills our desiderata for a two-valued version of SQL. Recall that it postulated three requirements: (1) that no expressiveness be gained or lost compared to the standard SQL; (2) that over databases without nulls no changes would be required; and (3) that when changes are required in the presence of nulls, they are small and do not significantly affect the size of the query.

We now summarize these conditions that an alternative semantics satisfies our desiderata in the following definition.

Definition 1. Given a query language $\mathcal{L}$ over relational databases with nulls, and two semantics of it, $[\![\,]\!]$ and $[\![\,]\!]'$, we say that $[\![\,]\!]'$ *captures* the semantics $[\![\,]\!]$ of $\mathcal{L}$ if the following conditions are satisfied:

(1) for every expression E of $\mathcal{L}$ there exists another expression G such that
$$[\![E]\!]'_D = [\![G]\!]_D$$
for every database D;
(2) for every expression E of $\mathcal{L}$ there exists another expression F such that
$$[\![E]\!]_D = [\![F]\!]'_D$$
for every database D; and
(3) for every expression E of $\mathcal{L}$, $[\![E]\!]_D = [\![E]\!]'_D$ for every database D without nulls.

Furthermore, $[\![\,]\!]'$ captures $[\![\,]\!]$ *efficiently* if the size of expressions F and G in items 1 and 2 above is at most linear in the size of expression E. □

Our main result is that the two-valued semantics of SQL captures its standard semantics efficiently.

Theorem 2. *The $[\![\,]\!]^{\mathbf{2VL}}$ semantics of RA^{REC}_{SQL} expressions, and of RA_{SQL} expressions, captures their SQL semantics $[\![\,]\!]$ efficiently.*

Note that the capture statement for $\mathrm{RA}_{\mathrm{SQL}}$ is not a corollary of the statement of $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$, as the capture definition states that the equivalent query must come from the same language. Thus, applying the statement about $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$ to an $\mathrm{RA}_{\mathrm{SQL}}$ expression E would only yield expressions G and F of $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$.

In the absence of nulls the definitions of $[\![\,]\!]^{\mathbf{2VL}}$ and $[\![\,]\!]$ coincide. Thus, condition (3) in the definition is satisfied. In the rest of this section we discuss the proof outline partly – we present the translation schemes for (1) along with some examples. All the translations are defined by a mutual induction on expressions and conditions. The key element is mimicking the conditions of one semantics under the other. That is, for a condition θ is a truth value $\boldsymbol{\tau}$, we have a condition $\theta^{\boldsymbol{\tau}}$ so that $[\![\theta]\!] = \boldsymbol{\tau}$ if and only if $[\![\theta^{\boldsymbol{\tau}}]\!]^{\mathbf{2VL}} = \mathbf{t}$. We inductively propagate these changes through the query, as our conditions can involve subqueries, e.g,. $\mathtt{empty}(E)$.

3.2 Translations from $[\![\,]\!]^{\mathbf{2VL}}$ to $[\![\,]\!]$

Given an expression E of $\mathrm{RA}_{\mathrm{SQL}}$, we describe how to construct an expression G such that $[\![E]\!]^{\mathbf{2VL}}_D = [\![G]\!]_D$ for every database D. To do so, as explained earlier, we define three translations by a mutual induction:

- from conditions θ to $\theta^{\mathbf{t}}$ and $\theta^{\mathbf{f}}$ such that:

$$[\![\theta]\!]^{\text{2VL}}_{D,\eta} = \mathbf{t} \text{ if and only if } [\![\theta^{\mathbf{t}}]\!]_{D,\eta} = \mathbf{t}$$
$$[\![\theta]\!]^{\text{2VL}}_{D,\eta} = \mathbf{f} \text{ if and only if } [\![\theta^{\mathbf{f}}]\!]_{D,\eta} = \mathbf{t}$$

  (note that $[\![\theta]\!]^{\text{2VL}}_{D,\eta}$ can only produce values $\mathbf{t}$ and $\mathbf{f}$);
- from expression E to G by inductively replacing each condition θ with $\theta^{\mathbf{t}}$.

The full details of these construction are presented in Figure 3. We note that the size of the resulting G is indeed linear in E.

Example 2. We now look at queries Q_1–Q_5 of Example 1 and provide their translations. That is, we assume these queries have been written assuming the two-valued 2VL semantics, and we show how they would then look in conventional SQL. To start with, queries Q_2, Q_3, and Q_4 remain unchanged by the translation.

Query Q_1 is translated into Q_1' given by

$$Q_1' \quad = \quad \sigma_{\texttt{isnull}(R.A) \vee \neg(R.A \in \sigma_{\neg \texttt{isnull}(S.A)} S)}(R)$$

In SQL, this corresponds to

```
SELECT R.A FROM R WHERE R.A IS NULL
OR R.A NOT IN
(SELECT S.A FROM S WHERE S.A IS NOT NULL)
```

In the translation Q_5' of query Q_5, the condition $(c\_a > 0) \wedge \neg(c\_c \in \pi_{o\_c}(O))$ in the subquery is translated as

$$\theta^{\mathbf{t}} := (c\_a > 0) \wedge \Big(\texttt{isnull}(c\_c) \vee \neg(c\_c \in \sigma_{\neg \texttt{isnull}(o\_c)}(\pi_{o\_c} O))\Big)$$

which is then used in the aggregate subquery Q_{agg}; the rest of the query does not change. In terms of SQL, in general these would be translated into additional **IS NULL** and **IS NOT NULL** conditions in the aggregate query as follows:

```
SELECT AVG( c_acctbal )
FROM customer WHERE c_acctbal > 0.0 AND
(c_custkey IS NULL OR
c_custkey NOT IN ( SELECT o_custkey FROM orders
                   WHERE o_custkey IS NOT NULL ))
```

This is a correct translation of Q_5 that makes no extra assumptions about the schema. Having such additional information (e.g., the fact that c_custkey is the key of customer) can simplify translation even further (e.g., by removing the **IS NULL** condition).

3.3 Query equivalence under two-valued semantics

Recall queries Q_1 and Q_2 from the introduction. Intuitively, one expects them to be equivalent: indeed, if we remove the **NOT** from both of them, then they are actually equivalent. And it seems that if θ_1 and θ_2 are equivalent, then so must be $\neg\theta_1$ and $\neg\theta_2$. So what is going on there?

Recall that the effect of the **WHERE** clause is to keep tuples for which the condition evaluated to $\mathbf{t}$. So equivalence of conditions θ_1 and θ_2, from SQL's point of view, means

$$[\![\theta_1]\!] = \mathbf{t} \;\; \Leftrightarrow \;\; [\![\theta_2]\!] = \mathbf{t}\,.$$

Thus, to state that the equivalence of θ_1 and θ_2 implies the equivalence of $\neg\theta_1$ and $\neg\theta_2$ we need the following condition:

$$\Big([\![\theta_1]\!] = \mathbf{t} \Leftrightarrow [\![\theta_2]\!] = \mathbf{t}\Big) \;\Rightarrow\; \Big([\![\neg\theta_1]\!] = \mathbf{t} \Leftrightarrow [\![\neg\theta_2]\!] = \mathbf{t}\Big) \qquad (1)$$

If (1) we true, we could conclude from the equivalence of queries

SELECT	...		**SELECT**	...
FROM	...	and	**FROM**	...
WHERE	θ_1		**WHERE**	θ_2

that queries

SELECT	...		**SELECT**	...
FROM	...	and	**FROM**	...
WHERE	$\neg\theta_1$		**WHERE**	$\neg\theta_2$

are equivalent. And this brings us to the reason why the two-valued semantics restores the expected equivalences of queries Q_1 and Q_2.

Proposition 1. *Implication (1) does not hold for SQL's semantics but holds for* $[\![\,]\!]^{\text{2VL}}$.

The reason behind it is that in 3VL, the *law of excluded middle*, $\theta \vee \neg\theta \leftrightarrow \mathbf{t}$, does not hold, and that is why (1) is invalidated. In two-valued logic, on the other hand, this law does hold, which gives us (1).

With the law of excluded middle we restore many more equivalences, often assumed for granted as one thinks in terms of Boolean logic, and yet programs in 3VL (perhaps accounting for some of the typical programmer mistakes in SQL [7, 9]).

Proposition 2. *The following equivalences hold:*

(1) $[\![\sigma_\theta(E)]\!]^{\text{2VL}}_{D,\eta} = [\![E - \sigma_{\neg\theta} E]\!]^{\text{2VL}}_{D,\eta}$

(2) $\bar{t} \in [\![E]\!]^{\text{2VL}}_{D,\eta}$ *if and only if* $[\![\texttt{empty}(\sigma_{\bar{t}=\ell(E)}(E))]\!]^{\text{2VL}}_{D,\eta} = \mathbf{f}$

(3) $[\![t\;\omega\;\texttt{any}(E)]\!]^{\text{2VL}}_{D,\eta}$ *if and only if* $[\![\texttt{empty}(\sigma_{\bar{t}\;\omega\;\ell(E)}(E))]\!]^{\text{2VL}}_{D,\eta} = \mathbf{f}$

(4) $[\![t\;\omega\;\texttt{all}(E)]\!]^{\text{2VL}}_{D,\eta}$ *if and only if* $[\![\texttt{empty}(\sigma_{\neg(\bar{t}\;\omega\;\ell(E))}(E))]\!]^{\text{2VL}}_{D,\eta} = \mathbf{t}$

for any RA^{REC}_{SQL} *expression* E*, tuple* $\bar{t}$*, and condition* θ*.*

Neither of those is true in general for SQL's three-valued semantics $[\![\,]\!]$.

4 OTHER TWO-VALUED SEMANTICS

We now show that the result on the equivalence of two-valued semantics with the usual SQL semantics is very robust. That is, many other two-valued semantics could be used in place of $[\![\,]\!]^{\text{2VL}}$, and with each of them we recover the equivalence with SQL's 3VL semantics.

What other two-valued semantics could be there? For a starter, there is the *syntactic equality* semantics, already used in SQL in connection with the **GROUP BY** operation as well as set operations, which treat **NULL** syntactically. In other words, for those operations, **NULL** equals itself. Formally, the semantic equality semantics $[\![\,]\!]^{=}$ is defined by changing the semantics of equality to

$$[\![t = t']\!]^{=}_{D,\eta} := \begin{cases} \mathbf{t} & [\![t]\!]_\eta = [\![t']\!]_\eta \\ \mathbf{f} & \text{otherwise} \end{cases}$$

and keeping the rest as in the definition of $[\![\,]\!]^{\text{2VL}}$. The only difference is that under this semantics, **NULL** $\doteq$ **NULL** evaluates to $\mathbf{t}$.

Basic conditions

$$(\mathbf{t})^{\mathbf{t}} := \mathbf{t} \qquad (\mathbf{t})^{\mathbf{f}} := \mathbf{f}$$

$$(\mathbf{f})^{\mathbf{t}} := \mathbf{f} \qquad (\mathbf{f})^{\mathbf{f}} := \mathbf{t}$$

$$\big(\mathtt{isnull}(t)\big)^{\mathbf{t}} := \mathtt{isnull}(t) \qquad \big(\mathtt{isnull}(t)\big)^{\mathbf{f}} := \neg\mathtt{isnull}(t)$$

$$\big(\bar{t} \doteq \bar{t}'\big)^{\mathbf{t}} := \bar{t} \doteq \bar{t}' \qquad \big(\bar{t} \doteq \bar{t}'\big)^{\mathbf{f}} := \bigvee_{i=1}^{n} \Big(\neg(t_i \doteq t_i') \vee \mathtt{isnull}(t_i) \vee \mathtt{isnull}(t_i')\Big)$$

$$\big(P(\bar{t})\big)^{\mathbf{t}} := P(\bar{t}) \qquad \big(P(\bar{t})\big)^{\mathbf{f}} := \neg P(\bar{t}) \vee \bigvee_{i=1}^{n} \mathtt{isnull}(t_i)$$

$$(\bar{t} \in E)^{\mathbf{t}} := \bar{t} \in G \qquad (\bar{t} \in E)^{\mathbf{f}} := \bigvee_{i=1}^{n} \mathtt{isnull}(t_i) \vee \neg(\bar{t} \in \sigma_{\neg\mathtt{isnull}(N_1)\wedge\cdots\wedge\neg\mathtt{isnull}(N_n)}(G))$$

where $\bar{t} := (t_1, \cdots, t_n)$, $\bar{t}' := (t_1', \cdots, t_n')$, and $\ell(G) := (N_1, \cdots, N_n)$

$$\big(\mathtt{empty}(E)\big)^{\mathbf{t}} := \mathtt{empty}(G) \qquad \big(\mathtt{empty}(E)\big)^{\mathbf{f}} := \neg\mathtt{empty}(G)$$

$$\big(t\,\omega\,\mathtt{any}(E)\big)^{\mathbf{t}} := t\,\omega\,\mathtt{any}(G) \qquad \big(t\,\omega\,\mathtt{any}(E)\big)^{\mathbf{f}} := \mathtt{empty}(\sigma_{\neg\theta}(G))$$

$$\big(t\,\omega\,\mathtt{all}(E)\big)^{\mathbf{t}} := t\,\omega\,\mathtt{all}(G) \qquad \big(t\,\omega\,\mathtt{all}(E)\big)^{\mathbf{f}} := \neg\mathtt{empty}(\sigma_{\theta}(G))$$

where $\ell(G) := N$ and $\theta := \mathtt{isnull}(t) \vee \mathtt{isnull}(N) \vee (\neg\mathtt{isnull}(t) \wedge \neg\mathtt{isnull}(N) \wedge \neg t\,\omega\,N)$

Composite conditions

$$(\theta_1 \vee \theta_2)^{\mathbf{t}} := (\theta_1)^{\mathbf{t}} \vee (\theta_2)^{\mathbf{t}} \quad (\theta_1 \vee \theta_2)^{\mathbf{f}} := (\theta_1)^{\mathbf{f}} \wedge (\theta_2)^{\mathbf{f}} \quad (\theta_1 \wedge \theta_2)^{\mathbf{t}} := (\theta_1)^{\mathbf{t}} \wedge (\theta_2)^{\mathbf{t}} \quad (\theta_1 \wedge \theta_2)^{\mathbf{f}} := (\theta_1)^{\mathbf{f}} \vee (\theta_2)^{\mathbf{f}} \quad (\neg\theta)^{\mathbf{t}} := \theta^{\mathbf{f}} \quad (\neg\theta)^{\mathbf{f}} := \theta^{\mathbf{t}}$$

Figure 3: 2VL semantics to SQL semantics

Coming back to the example of queries Q_1 and Q_2 from the introduction, under this semantics on $R = \{1, \mathbf{NULL}\}$ and $S = \{\mathbf{NULL}\}$ they produce $\{1\}$ as the answer: in this case, it is the same as we would have by applying R **EXCEPT** S based on syntactic equality. The semantics $[\![\,]\!]^{\mathbf{2VL}}$ and $[\![\,]\!]^{=}$ are, as expected, different, but note that both of them make queries Q_1 and Q_2 produce the same result, as they both satisfy condition (1) from Section 3.3 eliminating potential confusion of writing (supposedly) the difference query in ways that produce different results.

Is this the only possible other two-valued semantics? Of course not. Consider for example the predicate $\leq$. In both $[\![\,]\!]^{=}$ and $[\![\,]\!]^{\mathbf{2VL}}$, $\mathbf{NULL} \leq \mathbf{NULL}$ is $\mathbf{f}$, but under the syntactic equality interpretation it is not unreasonable to say that $\mathbf{NULL} \leq \mathbf{NULL}$ is true, as $\leq$ subsumes equality. This gives a general idea of how different semantics can be obtained: when some arguments of a predicate are **NULL**, we can decide what the truth value based on other values. Just for the sake of example, we could say that $n \leq \mathbf{NULL}$ is $\mathbf{f}$ for $n < 0$ and $\mathbf{t}$ for $n \geq 0$.

To define such alternative semantics in a general way, we simply state, for each predicate, what happens when some of the arguments are **NULL**. Formally, to define such semantics, we introduce the notion of *grounding* of predicates in Ω, as well as equality and comparison. It is a function $\mathbf{gr}$ that takes an an n-ary predicate P of type τ and a non-empty sequence $\mathrm{I} = \langle i_1, \cdots, i_k\rangle$ of indices $1 \leq i_1 < \cdots < i_k \leq n$, and produces a relation $\mathbf{gr}(P, \mathrm{I})$ that contains τ-records $(t_1, \cdots, t_n)$ where $t_i = \mathbf{NULL}$ for every $i \in \mathrm{I}$, and $t_i \neq \mathbf{NULL}$ for every $i \notin \mathrm{I}$. In the above example, if P is $\leq$, then $\mathbf{gr}(P, \langle 1\rangle) = \{(\mathbf{NULL}, n) \mid n \geq 0\}$ while $\mathbf{gr}(P, \langle 2\rangle) = \{(\mathbf{NULL}, n) \mid n < 0\}$ and $\mathbf{gr}(P, \langle 1, 2\rangle) = \{(\mathbf{NULL}, \mathbf{NULL})\}$.

The semantics $[\![\,]\!]^{\mathbf{gr}}$ based on such grounding is given by redefining the truth value of $[\![P(t_1, \ldots, t_n)]\!]^{\mathbf{gr}}$ as that of $\bar{t} \in \mathbf{gr}(P, \mathrm{I})$, where I is the list on indices i such that $[\![t_i]\!]^{\mathbf{gr}} = \mathbf{NULL}$.

Such semantics generalize $[\![\,]\!]^{\mathbf{2VL}}$ and $[\![\,]\!]^{=}$. In the former, by setting $\mathbf{gr}(P, \mathrm{I}) = \emptyset$ for each nonempty I; in the latter, it is the same except that $\mathbf{gr}(=, \langle 1, 2\rangle)$ would contain the tuple $(\mathbf{NULL}, \mathbf{NULL})$.

We say that a grounding is *expressible* if for each $P \in \Omega$ and each I there is a condition $\theta_{P,\mathrm{I}}$ such that $\bar{t} \in \mathbf{gr}(P, \mathrm{I})$ iff $\pi_{\bar{\mathrm{I}}}(\bar{t})$ satisfies $\theta_{P,\mathrm{I}}$. Note that the projection on the complement $\bar{\mathrm{I}}$ of I would only contain non-null elements. The semantics seen previously satisfy this condition.

Theorem 3. *For every expressible grounding* $\mathbf{gr}$*, the* $[\![\,]\!]^{\mathbf{gr}}$ *semantics of* RA^{REC}_{SQL} *or* RA_{SQL} *expressions captures their SQL semantics.*

5 OTHER MANY-VALUED LOGICS

To further support the claim that two-valued logic is a very natural alternative to 3VL in SQL, we now show that no other many-valued logic could have been used in its place in a way that would have altered the expressiveness of the language. Thus, of all the logics that give us *equal expressive power* it makes sense to choose the simplest and the most familiar one.

We build upon a result of [15] which proved such an equivalence between many-valued first-order logics under set semantics, and also under restrictions on the behavior of the many-valued connectives. We strengthen this by going from first-order logic to full SQL, and by eliminating the restrictions the result of [15] required.

We first define extensions of $\mathrm{RA}_{\mathrm{SQL}}$ based on different many-valued logics and state the equivalence result, and then give a hint of the proof.

5.1 Many-valued Interpretations

SQL's 3VL is an example of a many-valued logic, known well before SQL as *Kleene's logic* [6]. It is not the only logic proposed to deal with null values; there were others with 3,4,5, and even 6 values [13, 24, 34, 44]; see the discussion in the introduction. Could using one of them give us a more expressive language? We now give the negative answer.

Recall that a many-valued (propositional) logic MVL is given by a finite collection $\mathbf{T}$ of *truth values* with $\mathbf{t}, \mathbf{f} \in \mathbf{T}$, and a finite set Γ of *logical connectors* $\gamma : \mathbf{T}^{\mathsf{ar}(\gamma)} \to \mathbf{T}$. We refer to $\mathsf{ar}(\gamma)$ as the *arity* of γ, and assume that any many-valued logic includes at least the unary connective $\neg$ for **NOT**, and the binary connectives $\vee$ and $\wedge$ for **OR** and **AND**, whose restriction to the basic truth-values $\{\mathbf{t}, \mathbf{f}\}$ follows the rules of Boolean logic.

The only condition we impose on MVL is that $\vee$ and $\wedge$ be associative and commutative; without those we cannot write conditions like θ_1 **OR** $\cdots$ **OR** θ_k (and likewise for **AND**) in **WHERE** without worrying about the order of conditions and parentheses around them. Not having commutativity and associativity would also break many optimizations. The ability to write conditions like that is taken for granted by SQL programmers and is therefore a requirement one cannot waive.

A semantics $[\![\,]\!]^{\mathsf{MVL}}$ of $\mathrm{RA}_{\mathrm{SQL}}$ or $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$ expressions is given by defining the semantics of atomic conditions $P(\bar{t})$ and equality, and then following the connectives of MVL for complex conditions. Such a semantics of atomic conditions is *expressible* if it does not deviate from the standard semantics in the absence of nulls (i.e., $=$ is equality, $\leq$ is less-than-or-equal, etc), and if for each atomic condition P, the fact that $P(\bar{t})$ evaluates to a truth value $\boldsymbol{\tau}$ can be expressed by a condition under SQL's semantics $[\![\,]\!]$. This excludes pathological situations when conditions like $1 \leq 2$ evaluate to truth values other than $\mathbf{t}$, $\mathbf{f}$, or when conditions like "$\mathbf{NULL} \doteq n$ evaluates to $\boldsymbol{\tau}$" are not expressible in SQL (say, having a truth value $\boldsymbol{\tau}$ so that $\mathbf{NULL} \doteq n$ is $\boldsymbol{\tau}$ if n is an encoding of a halting Turing machine). Anything reasonable is permitted by being expressible. Formally, a semantics $[\![\,]\!]^{\mathsf{MVL}}$ is *SQL-expressible for atomic predicates* if:

(1) in the absence of nulls, the semantics of atomic conditions is the same as SQL's semantics $[\![\,]\!]$;
(2) for each truth value $\boldsymbol{\tau} \in \mathbf{T}$ and each atomic predicate P there is a condition $\theta_{P,\boldsymbol{\tau}}$ such that $[\![P(\bar{t})]\!]^{\mathsf{MVL}}_{D,\eta} = \boldsymbol{\tau}$ if and only if $[\![\theta_{P,\boldsymbol{\tau}}]\!]_{D,\eta} = \mathbf{t}$, for all D and η.

Example 3. We consider a 4-valued logic from [13]. It has truth values $\mathbf{t}$, $\mathbf{f}$, $\mathbf{u}$ just as SQL's 3VL, and also a new value $\mathbf{s}$. This value means "sometimes": under some interpretation of nulls the condition is true, but under some it is false. The semantics is then defined in the same way as SQL's semantics except $\mathbf{u}$ is replaced by $\mathbf{s}$.

In this logic the unknown $\mathbf{u}$ appears when one uses complex conditions. Suppose conditions θ_1 and θ_2 both evaluate to $\mathbf{s}$. Then there are interpretations of nulls where each one of them is true, and when each one of them is false, but we cannot conclude that there is an interpretation where both are true. Hence $\theta_1 \wedge \theta_2$ evaluates to $\mathbf{u}$ rather than $\mathbf{s}$. For full truth tables of this 4VL, see [13].

The previously known equivalence result [15] considered first-order logic under set semantics, and many-valued logics that are idempotent ($\boldsymbol{\tau} \vee \boldsymbol{\tau} = \boldsymbol{\tau}$) or weakly idempotent ($\boldsymbol{\tau} \vee \boldsymbol{\tau} \vee \boldsymbol{\tau} = \boldsymbol{\tau} \vee \boldsymbol{\tau}$). For example that the 4-valued logic from Example 3 is not idempotent but weakly idempotent.

We now show a much stronger result. As the query language we take either $\mathrm{RA}_{\mathrm{SQL}}$ or $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$. A many-valued logic does not have idempotency restrictions. Even in this general setting, the resulting semantics does not give any extra expressiveness compared to the standard SQL semantics.

Theorem 4. *For a many-valued logic* MVL *in which* $\wedge$ *and* $\vee$ *are associative and commutative, let* $[\![\,]\!]^{\mathsf{MVL}}$ *be a semantics of* RA_{SQL} *or* RA^{REC}_{SQL} *expressions based on* MVL. *Assume that this semantics is SQL-expressible for atomic predicates. Then it captures the SQL semantics.*

The translation from MVL semantics into SQL semantics may add an application of the **COUNT** aggregate and basic arithmetic $(+, *)$. The idea of the translation is discussed below.

5.2 $[\![\,]\!]^{\mathsf{MVL}}$ to SQL semantics

The main difficulty that needs to be addressed is that we assume practically nothing about the many-valued logic which makes the evaluation of conditions such as $\bar{t} \in E$ complicated. This is the disjunction of truth values of conditions $\bar{t}' \doteq \bar{t}$ as $\bar{t}'$ ranges over tuples in the evaluation of E, and if we assume nothing about the truth tables, how do we compute this? With idempotency, this is easy: no matter how many times a truth value $\boldsymbol{\tau}$ occurs in this disjunction, it collapses to one occurrence. With weak idempotency, it collapses to one or two occurrences. But without these conditions, we need to look for other mechanisms.

Continuing with this example, our goal is to construct a new expression F, for a truth value $\boldsymbol{\tau}$, such that $[\![\bar{t} \in E]\!]^{\mathsf{MVL}} = \boldsymbol{\tau}$ if and only if $[\![F]\!] = \mathbf{t}$. The most straightforward way to do this consists of two steps. First, we compute for each tuple $\bar{t}' \in E$ the truth value $[\![\bar{t}' \doteq \bar{t}]\!]^{\mathsf{MVL}}$. Second, we aggregate $\vee$ over these truth values. As our many-valued logic is expressible, we can assume that the first step is expressible, by means of a condition denoted by $\theta_{\doteq \bar{t}}$. Assume that we have an aggregate $\bigvee$ that takes a bag of truth values $\{\boldsymbol{\tau}_1, \cdots, \boldsymbol{\tau}_m\}$ to $\boldsymbol{\tau}_1 \vee \cdots \vee \boldsymbol{\tau}_m$. With that, F could be defined as follows, assuming $\bar{N} = \ell(E)$:

$$\mathtt{Group}_{\emptyset}\langle \bigvee(A)\rangle\Big(\pi_{\theta_{\doteq \bar{t}}(\bar{N}) \to A}(E)\Big)$$

The problem is that we do not have such an aggregate function in general; it would be unrealistic to expect it to exist for every MVL. For the usual Boolean logic, it can be implemented as **MAX** by associating $\mathbf{t}$ with 1 and $\mathbf{f}$ with 0, but in general we have no such recourse to numerical aggregates. Thus, we need a different approach, explained below.

Commutativity and associativity of disjunction allow us to change the order of the disjuncts as wanted without affecting the result. In fact, it implies that the result of the disjunction is only determined by the number of disjuncts for each truth value, hence

implying that we can view disjunction as a function $f_\vee$ that maps a vectors of integers of arity equals the number of truth values of our many-valued logic into a single truth value. More precisely, for a logic with truth values $\boldsymbol{\tau}_1, \cdots, \boldsymbol{\tau}_m$, we have $f_\vee(k_1, \cdots, k_m) = \boldsymbol{\tau}$ if

$$\boldsymbol{\tau} \;=\; \underbrace{(\boldsymbol{\tau}_1 \vee \cdots \vee \boldsymbol{\tau}_1)}_{k_1 \text{ times}} \vee \cdots \vee \underbrace{(\boldsymbol{\tau}_m \vee \cdots \vee \boldsymbol{\tau}_m)}_{k_m \text{ times}} .$$

Note that by expressibility of conditions in SQL's semantics, the number k_i of how many times $\bar{t}' \doteq \bar{t}$ evaluates to $\boldsymbol{\tau}_i$ can be expressed in $\mathrm{RA}_{\mathrm{SQL}}$ with the help of the **COUNT** aggregate.

The last missing bit is to calculate the disjunction of k $\boldsymbol{\tau}$s. Since $\mathbf{T}$ is of fixed size, we know that the sequence of truth values $\boldsymbol{\tau}, \boldsymbol{\tau} \vee \boldsymbol{\tau}, \boldsymbol{\tau} \vee \boldsymbol{\tau} \vee \boldsymbol{\tau}, \cdots$ eventually exhibits a periodic behavior. One can calculate the period and explicitly list truth values of such disjunctions up to the point from which the periodic behavior starts (that would be at most $|\mathbf{T}|+1$). With this, and simple arithmetic operations, one can calculate the value of the disjunction of k $\boldsymbol{\tau}$s, for each given k that was computed by a counting aggregate. This explains the main idea behind the proof; full details are in the appendix.

6 CONCLUSIONS

We have demonstrated that one of the most criticized aspects of SQL, and one that is the source of confusion for numerous SQL programmers – the use of the three-valued logic – was not really necessary, and perfectly reasonable two-valued semantics exist that achieve exactly the same expressiveness as the original three-valued design. Of course there is so much existing SQL code that operates under the three-valued semantics that changing that aspect of the language as if it were never there is not realistic. Thus, the questions are: what can we do with this now, and what can we do in the future.

Regarding now, there are two directions. First, since we provided equivalence results, it is entirely feasible to let programmers use two-valued SQL without changing the underlying DBMS implementation. One simply translates a query written under the two-valued semantics into standard SQL and runs it. Implementing such a translation is one of our immediate goals. Thus, no new implementation of query evaluation is necessary; we can reuse all the existing technology while at the same time getting rid of one of its most problematic aspects.

Second, as we explained in the introduction, this new point of view may well be of interest to designers of new languages. This activity is especially visible in the area of graph databases [43] and an alternative to SQL's confusing 3VL might well be considered.

For the future, the key direction to pursue is to make the translation more practical by taking into account additional semantic information. Such information is most likely to come in the form of constraints such as keys, foreign keys, and **NOT NULL** constraints. For now, translations we presented do not take them into account but we already saw in one of the examples that they could be very useful. We also plan to adapt works like [21, 27] to produce evaluation schemes that return results with certainty guarantees, under the two-valued approach.

Acknowledgments Part of this work was done when the second author was at the University of Edinburgh and with FSMP (Foundation Sciences Mathématiques de Paris) hosted by IRIF and ENS. We acknowledge support of EPSRC grants N023056 and S003800, and grants from FSMP and Neo4j.

REFERENCES

[1] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Software Eng.*, 14(7):879–885, 1988.
[2] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutiérrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-CORE A Core for Future Graph Query Languages. In *ACM SIGMOD*, 2018.
[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.
[4] O. Arieli, A. Avron, and A. Zamansky. What is an ideal logic for reasoning with inconsistency? In *IJCAI*, pages 706–711, 2011.
[5] V. Benzaken and E. Contejean. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 249–261. ACM, 2019.
[6] L. Bolc and P. Borowik. *Many-Valued Logics: Theoretical Foundations.* Springer, 1992.
[7] S. Brass and C. Goldberg. Semantic errors in SQL queries: A quite complete list. *J. Syst. Softw.*, 79(5):630–644, 2006.
[8] K. S. Candan, J. Grant, and V. S. Subrahmanian. A unified treatment of null values using constraints. *Inf. Sci.*, 98(1-4):99–156, 1997.
[9] J. Celko. *SQL for Smarties: Advanced SQL Programming.* Morgan Kaufmann, 2005.
[10] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. Software Eng.*, 11(4):324–345, 1985.
[11] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endow.*, 11(11):1482–1495, 2018.
[12] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017.
[13] M. Console, P. Guagliardo, and L. Libkin. Approximations and refinements of certain answers via many-valued logics. In *KR*, pages 349–358. AAAI Press, 2016.
[14] M. Console, P. Guagliardo, and L. Libkin. On querying incomplete information in databases under bag semantics. In *IJCAI*, pages 993–999, 2017.
[15] M. Console, P. Guagliardo, and L. Libkin. Propositional and predicate logics of incomplete information. In *KR*, pages 592–601. AAAI Press, 2018.
[16] H. Darwen and C. J. Date. The third manifesto. *SIGMOD Record*, 24(1):39–49, 1995.
[17] C. J. Date. *Database in Depth - Relational Theory for Practitioners.* O'Reilly, 2005.
[18] C. J. Date. A critique of Claude Rubinson's paper nulls, three-valued logic, and ambiguity in SQL: critiquing Date's critique. *SIGMOD Record*, 37(3):20–22, 2008.
[19] C. J. Date and H. Darwen. *A Guide to the SQL Standard.* Addison-Wesley, 1996.
[20] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. Aggregation support for modern graph analytics in TigerGraph. In *SIGMOD*, pages 377–392. ACM, 2020.
[21] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases - A lightweight approach for approximating certain answers. In *SIGMOD*, pages 1313–1330. ACM, 2019.
[22] M. Fitting. Kleene's logic, generalized. *J. Log. Comput.*, 1(6):797–810, 1991.
[23] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445. ACM, 2018.
[24] G. H. Gessert. Four valued logic for relational database systems. *SIGMOD Record*, 19(1):29–35, 1990.
[25] M. L. Ginsberg. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.
[26] S. Greco, C. Molinaro, and I. Trubitsyna. Approximation algorithms for querying incomplete databases. *Inf. Syst.*, 86:28–45, 2019.
[27] P. Guagliardo and L. Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *PODS*, pages 211–223. ACM, 2016.
[28] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, 2017.
[29] P. Guagliardo and L. Libkin. On the Codd semantics of SQL nulls. *Information Systems*, 86:46–60, 2019.
[30] A. Hernich and P. G. Kolaitis. Foundations of information integration under bag semantics. In *LICS*, pages 1–12. IEEE Computer Society, 2017.
[31] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.
[32] Ingres 9.3. *QUEL Reference Guide*, 2009.
[33] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaïda. On the optimization of recursive relational queries: Application to graph queries. In *SIGMOD*, pages 681–697. ACM, 2020.

[34] Y. Jia, Z. Feng, and M. Miller. A multivalued approach to handle nulls in RDB. In *Future Databases*, volume 3 of *Advanced Database Research and Development Series*, pages 71–76. World Scientific, Singapore, 1992.
[35] L. Libkin. SQL's three-valued logic and certain answers. *ACM Trans. Database Syst.*, 41(1):1:1–1:28, 2016.
[36] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3):513–534, 1991.
[37] C. Nikolaou, E. V. Kostylev, G. Konstantinidis, M. Kaminski, B. C. Grau, and I. Horrocks. The bag semantics of ontology-based data access. In *IJCAI*, pages 1224–1230, 2017.
[38] PostgreSQL Documentation, Version 9.6.1. `www.postgresql.org/docs/manuals`, 2016.
[39] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.
[40] Transaction Processing Performance Council. *TPC Benchmark™ H Standard Specification*, 2014. Revision 2.17.1.
[41] J. Van den Bussche and S. Vansummeren. Translating SQL into the relational algebra. Course notes, Hasselt University and Université Libre de Bruxelles, 2009.
[42] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *GRADES*, page 7. ACM, 2016.
[43] Wikipedia contributors. GQL graph query language, 2020.
[44] 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.
[45] C. Zaniolo. Database relations with null values. *JCSS*, 28(1):142–166, 1984.