SQL Nulls and Two-Valued Logic

SQL Nulls and Two-Valued Logic

Leonid Libkin
Liat Peterfreund
l@libk.in
liat.peterfreund@univ-eiffel.fr

ABSTRACT

The design of SQL is based on a three-valued logic (3VL), rather than the familiar two-valued Boolean logic (2VL). In addition to *true* and *false*, 3VL adds *unknown* to handle nulls. Viewed as indispensable for SQL expressiveness, it is often criticized for unintuitive behavior of queries and for being a source of programmer mistakes.

We show that, contrary to the widely held view, SQL could have been designed based on 2VL, without any loss of expressiveness. Similarly to SQL's WHERE clause, which only keeps true tuples, we conflate false and unknown for conditions involving nulls to obtain an equally expressive 2VL-based version of SQL. This applies to the core of the 1999 SQL Standard.

Queries written under the 2VL semantics can be efficiently translated into the 3VL SQL and thus executed on any existing RDBMS. We show that 2VL enables additional optimizations. To gauge its applicability, we establish criteria under which 2VL and 3VL semantics coincide, and analyze common benchmarks such as TPC-H and TPC-DS to show that most of their queries are such. For queries that behave differently under 2VL and 3VL, we undertake a user study to show a consistent preference for the 2VL semantics.

CCS CONCEPTS

• **Information systems** → **Relational database query languages**; *Relational database model*; • **Theory of computation** → *Logic*.

KEYWORDS

SQL; nulls; three-valued logic; Boolean logic; query equivalence; query optimization; user study

ACM Reference Format:
Leonid Libkin and Liat Peterfreund. 2023. SQL Nulls and Two-Valued Logic. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3584372.3588661

1 INTRODUCTION

To process data with nulls, SQL uses a three-valued logic (3VL), with an additional truth value *unknown*. This is one of the most often criticized aspects of the language, and one that is very confusing to programmers [7]. Database texts are full of damning statements about the treatment of nulls, such as the inability to explain them in a "comprehensible" manner [18], their tendency to "ruin everything" [9] and outright recommendations to "avoid nulls" [16]. The latter, however, is often not possible: in large volumes of data, incompleteness is hard to avoid.

Issues related to null handling stem not just from the use of 3VL, but from multiple and disparate ways of using it. To illustrate:

- Conditions, such as those in **WHERE**, are evaluated under 3VL, with any atomic condition involving a **NULL** resulting in *unknown*. In the end, however, only *true* tuples are kept; that is, *false* and *unknown* are conflated.
- Constraints, such as **UNIQUE** and foreign keys, are too evaluated under 3VL, but then a constraint holds if it does not evaluate to *false*; that is, *true* and *unknown* are conflated.
- SQL's **NULL** can also be viewed as a syntactic constant, making two **NULL**s equal; this is how grouping and set operations work.

Not only is the SQL programmer forced to use a logic different from other languages they are familiar with, even that logic is applied in different ways in different scenarios.

We now look at some examples where 3VL causes confusion even for very simple SQL queries. As a starter, consider the rewriting of **IN** subqueries into **EXISTS** ones. Queries

```
(Q1) SELECT R.A FROM R WHERE R.A NOT IN
       ( SELECT S.A FROM S )
```

and

```
(Q2) SELECT R.A FROM R WHERE NOT EXISTS
       ( SELECT S.A FROM S WHERE S.A=R.A )
```

would regularly be presented as equivalent (see, e.g., [45]). While equivalent if both **NOT**s are removed, these queries differ in SQL: if $R = \{1, \textbf{NULL}\}$ and $S = \{\textbf{NULL}\}$, then Q1 returns no tuples, while Q2 returns $\{1, \textbf{NULL}\}$. Such presumed, but incorrect, equivalence is a trap many SQL programmers are not aware of (see [7, 9]).

As another example, consider two queries given as an illustration of the HoTTSQL system for proving query equivalences [11]:

```
(Q3) SELECT DISTINCT X.A FROM R X, R Y
       WHERE X.A=Y.A

(Q4) SELECT DISTINCT R.A FROM R
```

Queries Q3 and Q4 are claimed to be equivalent in [11], but this is not the case: if $R = \{\textbf{NULL}\}$, then Q3 returns an empty table while Q4 returns **NULL**. In fairness, the reason why they are equivalent in [11] is that HoTTSQL considers only databases *without nulls*. Nonetheless, this is illustrative of the subtleties surrounding SQL nulls: what [11] chose as an "easy" example of equivalence involves two non-equivalent queries on a simple database containing **NULL**.

PODS '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0127-6/23/06…$15.00
https://doi.org/10.1145/3584372.3588661

Over the years, two main lines of research emerged for dealing with these problems. One is to provide a more complex logic for handling incompleteness [8, 12, 17, 24, 34, 46]. These proposals did not take off, because the underlying logic is even harder for programmers than 3VL. An alternative is to have a language with no nulls at all, and thus resort to the usual two-valued logic. This found more success, for example in the "3rd manifesto" [15] and the Tutorial D language, as well as in the LogicBlox system [3] and its successor [1], which use the sixth normal form to eliminate nulls. But nulls do occur in most SQL databases and thus must be handled; the world is not yet ready to dismiss them completely.

We thus pursue a different approach: a flavor of SQL with nulls, but based on Boolean logic. This goal is to have a flavor of SQL that can be offered as an alternative to coexist along with the 3VL standard. To achieve this, we need to fulfill the following criteria.

(1) *Do not make changes unless necessary*: On databases without nulls queries should be written exactly as before, and return the same results;
(2) *Do not lose any queries; do not invent new ones*: The new version of SQL should have exactly the same expressiveness as its version based on 3VL;
(3) *Do not make queries overly complicated*: For each SQL query using 3VL, the equivalent query in the two-valued should not add joins, and be roughly of the same size.

We pursue these goals along two different routes. First, we provide theoretical evidence that our desiderata are fulfilled for the core of SQL. Second, we go beyond theoretical results and supplement them by preliminary evidence of the utility of a version of SQL devoid of 3VL.

One may wonder why our goal is even achievable, considering almost 40 years of SQL practice firmly rooted in 3VL. The reason to pursue this line of work lies in 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 capturing 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 never evaluate to *unknown*. Following that, [14] considered many-valued first-order predicate calculi under set semantics, and showed that no many-valued logic provides additional expressive power over Boolean logic.

Results. We show that the elimination of *unknown* works for SQL, including its core features from the 1992 Standard (full relational algebra expanded with arithmetic functions and comparisons, aggregate functions and **GROUP BY**; comparisons of aggregates in **HAVING**, subqueries connected by **IN**, **EXISTS**, **ALL**, **ANY**, and set operations **UNION**, **INTERSECT**, **EXCEPT**, optionally with **ALL**), as well as **WITH RECURSIVE** added in the 1999 Standard [18, 20].

The *unknown* appears when one evaluates a condition such as `R.A=S.A` in which one or more arguments are **NULL**. Once a condition in **WHERE** is evaluated, only *true* tuples are kept, while *unknown* or *false* ones are dismissed. A minimal change that ensures elimination of 3VL then brings this conflating *unknown* with *false* forward, before the exit from the **WHERE** clause. One small variation lies in treating conditions like **NULL** = **NULL**. One may still evaluate them to *false*, or assume syntactic equality as in **GROUP BY** and evaluate them to *true*. Either way, we obtain two-valued versions of SQL satisfying our desiderata.

Replacing 3VL with 2VL does not necessitate any changes to the underlying implementation of RDBMSs. A user can write a query under a two-valued Boolean semantics; the query is then translated into an equivalent one under the standard SQL semantics, which any of the existing engines can evaluate.

We investigate the impact of this result from two different angles. The first concerns optimizations. By changing the semantics we change equivalences among queries. We show that the two-valued version of SQL recovers certain optimizations, in particular those often incorrectly assumed by programmers under 3VL.

Our second question concerns real-life usability of two-valued SQL. To investigate this, we ask two questions:

(1) Does it happen often that the choice of logic, 3VL or 2VL, has no impact on the query output?
(2) If there is a difference between 3VL and 2VL, which one would users prefer?

Regarding (1), we observe that for many queries, there is actually no difference between outputs while using 2VL or 3VL. We provide sufficient conditions for this to happen, and then analyze queries in commonly used benchmarks, TPC-H [44] and TPC-DS [43], to show that a huge majority of queries fall in that category, giving us the two-valued SQL essentially for free. This is not very surprising since these benchmarks were written by experienced programmers who know how to avoid semantic pitfalls.

When there is a difference, the only way to know what users prefer is to ask the users. We thus designed an introductory user survey asking about preference for 2VL or 3VL in both query outputs and query equivalence. As with every user survey, there is a tradeoff between the costs of the running a survey and reliability of its results. This being the first survey of the kind, we wanted to get an initial indication of what users think; starting the project we had no idea whether they would love the idea of 2VL, or reject it outright, or fall somewhere in between. The survey of roughly 80% practitioners and 20% academics showed that by – on average – the margin of 2-to-1 users prefer 2VL. This should not be viewed as the final word but rather as an initial confirmation of the feasibility of the approach, and an invitation for a more detailed user study before potential proposals for language changes.

Finally, we show an extension of our results: no other reasonable (essentially, avoiding paradoxical behaviour) many-valued logic in place of 3VL could give a more expressive language than SQL.

The choice of language. To prove results formally, we need a language closely resembling SQL and yet having a formal semantics one can reason about. Our choice is an extended relational algebra (RA) similar to an algebra into which RDBMS implementations translate SQL. It expands the standard textbook RA with bag semantics, duplicate elimination, and several new features. Selection conditions, in addition to the standard comparisons such as = and <, include tests for nulls (as SQL's **IS NULL**) and both **IN** or **EXISTS** *subqueries*. We also add conditions $\bar{t}\omega\mathsf{any}(E)$ and $\bar{t}\omega\mathsf{all}(E)$ with the semantics of SQL's **ANY** and **ALL** (they check whether $\bar{t}\omega\bar{t}'$ holds for some, respectively all, $\bar{t}'$ in the result of E, where ω is one of the standard comparisons). Selection conditions are evaluated according to SQL's 3VL. The algebra has aggregate functions and

a grouping operation. It allows function application to attributes, to mimic expressions in the **SELECT** clause. It also has an iterator operation whose semantics captures SQL recursion.

Related work. The idea of using Boolean logic for nulls predates SQL; it actually appeared in QUEL (the language of Ingres that appeared in 1976 [41]; 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 [12, 17, 24, 34, 46] or producing more complex classifications of nulls, e.g., [8, 47]. Elaborate many-valued logics for handling incomplete and inconsistent data were also considered in the AI literature; see, e.g., [4, 22, 25]. Proposals for eliminating nulls have appeared in [1, 3, 15].

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. Our work is orthogonal to that: we are concerned with finding a logic that makes it more natural for programmers to write queries; once this is achieved, one will need to modify the evaluation schemes to produce subsets of certain answers if one so desires. For connections between real SQL nulls and theoretical models, such as marked or Codd nulls, see [29].

Some papers looked into handling nulls and incomplete data in bag-based data models as employed by SQL [13, 30, 38], but none focused on the underlying logic of nulls.

Finally, our user survey can be seen as complementary to the extensive survey on the use of nulls [42]; that survey asked multiple questions but not on replacing SQL's logic of nulls.

Organization. Section 2 presents the syntax and the semantics of the language. Section 3 shows how to eliminate *unknown* to achieve our desiderata. Section 4 studies optimizations under 2VL. Section 5 discusses conditions under which 2VL and 3VL semantics are equivalent. Section 6 studies applicability of our results. Extensions are given in Section 7.

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

Given the idiosyncrasies of SQL's syntax, it is not an ideal language – syntactically – to reason about. We know, however, that its queries are all translatable into an extended RA; indeed, this is what is done inside every RDBMS, and multiple such translations are described in the literature [5, 10, 28, 37, 45]. Our language $\mathrm{RA}_{\mathrm{SQL}}$ is close to what real-life SQL queries are translated into.

2.1 Data Model

The usual presentation of RA assumes a countably infinite domain of values. To handle languages with aggregation, 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 *ordinary*). This is without any loss of generality since the treatment of nulls as values of all types is the same, except numerical nulls that behave differently with respect to aggregation.

Assume the following pairwise disjoint countable infinite sets:

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

Each name has a *type*: either o (ordinary) or n (numerical). If $N \in \mathbf{Name}$, then $\mathrm{type}(N) \in \{\mathsf{o}, \mathsf{n}\}$ defines the type of elements in column N. Furthermore, $\mathrm{type}(e) = \mathsf{n}$ if $e \in \mathbf{Num}$ and $\mathrm{type}(e) = \mathsf{o}$ if $e \in \mathbf{Val}$. We use a 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 $a_i \in \mathbf{Num} \cup \{\mathbf{NULL}\}$ whenever $\tau_i = \mathsf{n}$, and $a_i \in \mathbf{Val} \cup \{\mathbf{NULL}\}$ whenever $\tau_i = \mathsf{o}$.

For an n-ary relation symbol R in the schema we write $\ell(R) = N_1 \cdots N_n \in \mathbf{Name}^n$ for the sequence of its attribute names. The *type* of R is the sequence $\mathrm{type}(R) = \mathrm{type}(N_1) \cdots \mathrm{type}(N_n)$. A *relation* $\mathbf{R}$ over R is a *bag* of $\mathrm{type}(R)$-records (a record may appear more than once). We write $\bar{a} \in_k \mathbf{R}$ if $\bar{a}$ occurs exactly $k > 0$ times in $\mathbf{R}$.

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

2.2 Syntax

A *term* is defined recursively 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 $f : \mathbf{Num}^k \to \mathbf{Num}$ is a *k-ary numerical function* (e.g., addition or multiplication) and $t_1, \cdots, t_k$ are terms that evaluate to values of numerical type.

An *aggregate function* is a function F that maps bags of numerical values into a numerical value. For example, SQL's aggregates **COUNT**, **AVG**, **SUM**, **MIN**, **MAX** are such.

The algebra is parameterized by a collection Ω of numerical and aggregate functions. We assume the standard comparison predicates $\doteq, \neq, <, >, \leq, \geq$ on numerical values are available [1].

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 R ranges over relation symbols in $\mathcal{S}$, 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 the size of its parse tree. We assume that comparisons between tuples are spelled out as Boolean combinations of atomic comparisons: e.g., $(x_1, x_2) = (y_1, y_2)$ is $x_1 = y_1 \wedge x_2 = y_2$ and $(x_1, x_2) < (y_1, y_2)$ is $x_1 < y_1 \vee (x_1 = y_2 \wedge x_2 < y_2)$.

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).

2.3 Semantics

To make reading easier, we present the semantics by recourse to SQL, but full and formal definitions exist too and are found in the full version [36]. The semantics function

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

[1] We focus, without loss of generality, on these predicates; our results apply also for predicates of higher arities, e.g., **BETWEEN** in SQL.

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

Expressions:

$E :=$	R	(base relation)
	$\pi_{t_1[\to N'_1], \cdots, t_m[\to N'_m]}(E)$	(generalized projection w/ 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)
	$\texttt{Group}_{\bar{N}, \langle F_1(N_1)[\to N'_1], \cdots, F_m(N_m)[\to N'_m]\rangle}(E)$	(grouping/aggregation w/ optional renaming)

Atomic conditions: $ac := \mathbf{t} \mid \mathbf{f} \mid \texttt{isnull}(t) \mid \bar{t}\ \omega\ \bar{t}' \mid \bar{t} \in E \mid \texttt{empty}(E) \mid \bar{t}\ \omega\ \texttt{any}(E) \mid \bar{t}\ \omega\ \texttt{all}(E) \qquad \omega \in \{\doteq, \neq, <, >, \leq, \geq\}$

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

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

defines the result of the evaluation of expression E on database D under the *environment* η. The environment η is a partial mapping from the set **Name** of names to the union $\mathbf{Val} \cup \mathbf{Num} \cup \{\texttt{NULL}\}$. It provides values of parameters of the query. This is necessary to give semantics of subqueries that can refer to attributes from the outer query.

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 parameters).

Similarly to SQL queries, each $\mathrm{RA}_{\mathrm{SQL}}$ expression E produces tables over a list of attributes; this list will be denoted by $\ell(E)$.

2.3.1 Base relations and generalized projections. A base relation R is SQL's **SELECT * FROM R**. Generalized projection captures SQL's **SELECT** clause. Each term t_i is evaluated, optionally renamed, and added as a column to the result. For example

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

is written as $\pi_{A,B,\mathrm{add2}(A)\to C,\mathrm{mult}(A,B)\to D}(R)$, where $\mathrm{add2}(x) := x + 2$ and $\mathrm{mult}(x, y) := x \cdot y$. Projection follows SQL's bag semantics with terms evaluated along with their multiplicity. For instance if $(2, 3) \in_2 R$ and $(1, 6) \in_3 R$ then the result of $\pi_{\mathrm{mult}(A,B)}(R)$ contains the tuple (6) with multiplicity 5.

2.3.2 Conditions and selections. SQL uses three-valued logic and its conditions are evaluated to either true (**t**), or false (**f**), or unknown (**u**). Logical connectives are used to compose conditions, and truth values are propagated according to Kleene logic below.

$\wedge$	**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**

Atomic condition $\texttt{isnull}(t)$ is evaluated to **t** if the term t is **NULL**, and to **f** otherwise. Comparisons $t\ \omega\ t'$ are defined naturally when the arguments are not **NULL**. If at least one argument is **NULL**, then the value is unknown (**u**).

The condition $\bar{t} \doteq \bar{t}'$, where $\bar{t} = (t_1, \ldots, t_m)$ and $\bar{t}' = (t'_1, \ldots, t'_m)$ that compares tuples of terms is the abbreviation of the conjunction $\bigwedge_{i=1}^m t_i \doteq t'_i$, and the comparison $\bar{t} \neq \bar{t}'$ abbreviates $\bigvee_{i=1}^m t_i \neq t'_i$. Comparisons $<, \leq, \geq, >$ of tuples are defined lexicographically.

The condition $\bar{t} \in E$, not typically included in RA, tests whether a tuple belongs to the result of a query, and corresponds to SQL's **IN** subqueries. If E evaluates to the bag containing $\bar{t}_1, \cdots, \bar{t}_n$ then $\bar{t} \in E$ stands for the disjunction $\bigvee_{i=1}^n \bar{t} \doteq \bar{t}_i$. Other predicates not typically included in RA presentation, though included here for direct correspondence with SQL, are **ALL** and **ANY** comparisons. The condition $\bar{t}\ \omega\ \texttt{any}(E)$ checks whether there exists a tuple $\bar{t}'$ in E so that $\bar{t}\ \omega\ \bar{t}'$ holds, where ω is one of the allowed comparisons. Likewise, $\bar{t}\ \omega\ \texttt{all}(E)$ checks whether $\bar{t}\ \omega\ \bar{t}'$ holds for every tuple $\bar{t}'$ in E (in particular, if E returns no tuples, this condition is true). If ω is $\doteq$ or $\neq$, conditions with any and all are applicable at either ordinary or numerical type; if ω is one of $<, \leq, >, \geq$, then all the components of $\bar{t}$ and all attributes of E are of numerical type.

The condition $\texttt{empty}(E)$ checks if the result of E is empty, and corresponds to SQL's **EXISTS** subqueries. Note that **EXISTS** subqueries can be evaluated to **t** or **f**, whereas **IN** subqueries can also be evaluated to **u**. The semantics of composite conditions is defined by the 3VL truth-tables.

Selection evaluates the condition θ for each tuple, and keeps tuples for which θ is **t** (i.e., not **f** nor **u**). Operations of generalized projection and selection correspond to sequential scans in query plans (with filtering in the case of selection).

2.3.3 Bag operations and grouping/aggregations. The operation ε is SQL's **DISTINCT**: it eliminates duplicates and keeps one copy of each record. Operations union, intersection, and difference, have the standard meaning under the bag semantics, and correspond, respectively, to SQL's **UNION ALL**, **INTERSECT ALL**, and **EXCEPT ALL**. Dropping the keyword **ALL** amounts to using set semantics for both arguments and operations.

Cartesian product has the standard meaning and corresponds to listing relations in the **FROM** clause.

We use SQL's semantics of functions: if one of its arguments is **NULL**, then the result is null (e.g., 3 + 2 is 5, but **NULL** + 2 is **NULL**).

Finally, we describe the operator $\mathtt{Group}_{\bar{N},\langle\bar{M}\rangle}(E)$. The tuple $\bar{N}$ lists attributes in **GROUP BY**, and the i'th coordinate of $\bar{M}$ is of the form $F_i(N_i)$ where F_i is an aggregate over the numerical columns N_i optionally renamed N_i' if $[\to N_i']$ is present. For example

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

will be expressed by $\mathtt{Group}_{A,\langle F_{\mathsf{count}}(B)[\to C],F_{\mathsf{sum}}(B)\rangle}(R)$ 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 $\mathrm{RA}_{\mathrm{SQL}}$:

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

A more complex example is a query Q_5 below; it is a slightly simplified (to fit in one column) query 22 from TPC-H [44]:

```
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
```

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 $\theta' := c\_a > \mathsf{any}(Q_{agg})$ which is then applied to C, i.e., $\sigma_{c\_a>\mathsf{any}(Q_{agg})}(C)$, and finally grouping by c_n and counting of c_a are performed over it, giving us

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

Putting everything together, we have the final $\mathrm{RA}_{\mathrm{SQL}}$ expression:

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

ADDING RECURSION. We now incorporate recursive queries, a feature added in the SQL 1999 standard with its **WITH RECURSIVE** construct. While extensions of relational algebra with various kinds of recursion exist (e.g., transitive closure [2] or fixed-point operator [33]), we stay closer to SQL as it is. Specifically, SQL uses a special type of iteration – in fact two kinds depending on the syntactic shape of the query [39].

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 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.

In SQL, various restrictions are imposed on query E_2, such as the linearity of recursion (at most one reference to R within E_2), restrictions on the use of recursively defined relations in subqueries, on the use of aggregation, etc. These eliminate many of the common cases of non-terminating queries. Here we shall not impose these restrictions, as our result is *more general*: passing from 3VL to two-valued logic is possible even if such restrictions were not in place.

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) $R_{i+1} := [\![E_2]\!]_{D\cup R_i,\eta}$, $RES_{i+1} := RES_i \cup R_{i+1}$

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

For $\mu R.E_1 \sqcup E_2$, the semantics is defined by a different iteration

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

with the same stopping condition as before.

Note that while for queries not involving recursion only the environment changes during the computation, for recursion the relation that is iterated over (R above) changes as well, and each new iteration is evaluated on a modified database.

3 ELIMINATING UNKNOWN

To replace 3VL with Boolean logic, we need to eliminate the unknown truth value. In SQL, **u** arises in **WHERE** which corresponds to conditions in $\mathrm{RA}_{\mathrm{SQL}}$. It appears as the result of evaluation of comparison predicates such as $=, \leq, \neq$ etc. Consequently, it also arises in **IN**, **ANY** and **ALL** conditions for subqueries.

In comparisons, **u** appears due to the rule that *if one parameter is* **NULL**, *then the value of the predicate is* **u**. Thus, we need to change this rule, and to say what to do when one of the parameters is **NULL**. In doing so, we are guided by SQL's existing semantics of conditions in **WHERE**. While those can evaluate to **t**, **f**, or **u**, in the end only the *true* values are kept: that is, **u** *and* **f** *are conflated*. SQL does it at the end of evaluating a condition; thus a natural approach to a two-valued version of SQL is to use the same rule *throughout* the evaluation.

This is a natural proposal, and in fact we shall that this results in a version of SQL satisfying our desiderata. It may have a potential drawback with respect to optimizations. Namely, both **NULL** $\doteq$ **NULL** and **NULL** $\neq$ **NULL** evaluate to **f**, and thus **NULL** $\neq$ **NULL** cannot be equivalent to $\neg$(**NULL** $\doteq$ **NULL**). This however can easily be resolved by treating conditions consistent with syntactic equality differently.

3.1 The two-valued semantics $[\![\,]\!]^{\mathbf{2VL}}$ and $[\![\,]\!]^{=}$

As explained above, in the new semantics $[\![\,]\!]^{\mathbf{2VL}}$ we only need to modify the rule for comparisons of terms, $t\,\omega\,t'$. In the simplest case (**f** instead of **u**) this is done by

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

The rest of the semantics is exactly the same as before. Note that in conditions like $\bar{t} \doteq \bar{t}'$, or $\bar{t} \in E$, or $\bar{t}\,\omega\,\mathtt{all}(E)$, and $\bar{t}\,\omega\,\mathsf{any}(E)$, the conjunctions and disjunctions will be interpreted as the standard Boolean ones, since **u** no longer arises.

A more elaborate version $[\![\,]\!]^{=}$ takes into account syntactic equality. It is the same as the $[\![\,]\!]^{\mathbf{2VL}}$ semantics except for three comparisons compatible with equality: $\doteq$, $\leq$ and $\geq$. For them, it is as follows

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

and keeping the rest as in the definition of $[\![\,]\!]^{\mathbf{2VL}}$. The only difference is that now conditions $\mathtt{NULL} \doteq \mathtt{NULL}$, $\mathtt{NULL} \leq \mathtt{NULL}$, and $\mathtt{NULL} \geq \mathtt{NULL}$ evaluate to true.

3.2 Capturing SQL with $[\![\,]\!]^{\mathbf{2VL}}$ and $[\![\,]\!]^{=}$

We now show that the two semantics presented above fulfill 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 be required; and (3) that when changes are required in the presence of nulls, they ought to be small and not affect significantly the size of the query. These conditions are formalized in the definition below.

Definition 1. *A semantics $[\![\,]\!]'$ of queries* captures *the semantics $[\![\,]\!]$ of RA_{SQL} if the following are satisfied:*

(1) for every expression E of RA_{SQL} there exists an expression G of RA_{SQL} such that, for each database D,

$$[\![E]\!]'_D = [\![G]\!]_D\,;$$

(2) for every expression E of RA_{SQL} there exists an expression F of RA_{SQL} such that, for every database D,

$$[\![E]\!]_D = [\![F]\!]'_D\,;$$

(3) for every expression E of RA_{SQL}, and every database D without nulls, $[\![E]\!]_D = [\![E]\!]'_D$.

When in place of RA_{SQL} above we use RA^{REC}_{SQL}, then we speak of capturing the semantics of RA^{REC}_{SQL}.

If the size of expressions F and G in items (1) and (2) is at most linear in the size of E, we say that the semantics is captured efficiently. □

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

Theorem 1. *The $[\![\,]\!]^{\mathbf{2VL}}$ and $[\![\,]\!]^{=}$ semantics of RA^{REC}_{SQL} expressions, and of RA_{SQL} expressions, capture their SQL semantics $[\![\,]\!]$ efficiently.*

Note that the capture statement for $\mathrm{RA}_{\mathrm{SQL}}$ is *not* a corollary of the statement of $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$.

Running 2VL on existing RDBMSs

We sketch one direction of the proof of Theorem 1, namely from $[\![\,]\!]^{\mathbf{2VL}}$ and $[\![\,]\!]^{=}$ to $[\![\,]\!]$.

From $[\![\cdot]\!]^{\mathbf{2VL}}$ to $[\![\,]\!]$, we define the translation $\mathrm{toSQL}_{\mathrm{2VL}}(\,)$ that specifies how to take a query E written under the 2VL semantics that conflates **u** with **f** and translate it into a query $\mathrm{toSQL}_{\mathrm{2VL}}(E)$ that gives the same result when evaluated under the usual SQL semantics: $[\![E]\!]^{\mathbf{2VL}}_D = [\![\mathrm{toSQL}_{\mathrm{2VL}}(E)]\!]_D$ for every database D. Thus, $\mathrm{toSQL}_{\mathrm{2VL}}(E)$ can execute a 2VL query in any existing implementation of SQL.

To do so, we define translations of conditions and queries by mutual induction. Translations $\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\cdot)$, $\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\cdot)$ on conditions θ ensure

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

(note that $[\![\theta]\!]^{\mathbf{2VL}}_{D,\eta}$ produces only **t** and **f**). Then we go from E to $\mathrm{toSQL}_{\mathrm{2VL}}(E)$ by inductively replacing each condition θ with $\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta)$.

The full details of the translations are in Figure 2.

Basic:	$\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta) := \theta$ for $\theta := \mathbf{t} \mid \mathbf{f} \mid \mathtt{isnull}(t) \mid t\,\omega\,t'$
	$\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\mathsf{empty}(E)) := \mathsf{empty}(\mathrm{toSQL}_{\mathrm{2VL}}(E))$
	$\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\bar{t}\,\omega\,\mathsf{any}(E)) := \bar{t}\,\omega\,\mathsf{any}(\mathrm{toSQL}_{\mathrm{2VL}}(E))$
	$\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\bar{t}\,\omega\,\mathtt{all}(E)) := \bar{t}\,\omega\,\mathtt{all}(\mathrm{toSQL}_{\mathrm{2VL}}(E))$
Comp.:	$\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta_1 \vee \theta_2) := \mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta_1) \vee \mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta_2)$
	$\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta_1 \wedge \theta_2) := \mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta_1) \wedge \mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta_2)$
	$\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\neg\theta) := \mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta)$
Basic:	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta) := \neg\theta$ for $\theta := \mathbf{t} \mid \mathbf{f} \mid \mathtt{isnull}(t)$
	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(t\,\omega\,t') := \mathtt{isnull}(t) \vee \mathtt{isnull}(t') \vee \neg t\,\omega\,t'$
	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\mathsf{empty}(E)) := \neg\mathsf{empty}(\mathrm{toSQL}_{\mathrm{2VL}}(E))$
	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\bar{t}\,\omega\,\mathsf{any}(E)) := \mathsf{empty}(\sigma_{\neg\theta}(\mathrm{toSQL}_{\mathrm{2VL}}(E)))$
	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\bar{t}\,\omega\,\mathtt{all}(E)) := \neg\mathsf{empty}(\sigma_{\theta}(\mathrm{toSQL}_{\mathrm{2VL}}(E)))$ where $\theta := \mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\bar{t}\,\omega\,\ell(E))$
Comp.:	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta_1 \vee \theta_2) := \mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta_1) \wedge \mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta_2)$
	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta_1 \wedge \theta_2) := \mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta_1) \vee \mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\theta_2)$
	$\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\neg\theta) := \mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\theta)$

Figure 2: $\mathrm{tr}^{\mathbf{t}}_{\mathrm{2VL}}(\cdot)$ and $\mathrm{tr}^{\mathbf{f}}_{\mathrm{2VL}}(\cdot)$ of basic and composite conditions

Example 2. We now look at translations of queries Q_1–Q_5 of Example 1. That is, suppose these queries have been written assuming the two-valued 2VL semantics; 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.

The query $\mathrm{toSQL}_{\mathrm{2VL}}(Q_1)$ is $\sigma_{\mathtt{isnull}(R.A) \vee \neg(R.A \in \sigma_{\neg\mathtt{isnull}(S.A)}S)}(R)$. In SQL, this is equivalent 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 $\text{toSQL}_{2VL}(Q_5)$, the condition $(c\_a > 0) \wedge \neg(c\_c \in \pi_{o\_c}(O))$ in the subquery is translated by $\text{tr}^{\mathbf{t}}_{2VL}(\cdot)$ as

$$(c\_a > 0) \wedge \Big(\texttt{isnull}(c\_c) \vee \neg(c\_c \in \sigma_{\neg\texttt{isnull}(o\_c)}(\pi_{o\_c}O))\Big)$$

which is then used in the aggregate subquery Q_{agg} (see details in Example 1 at the end of Section 2.3); the rest of the query does not change. In SQL, these are translated into additional **`IS NULL`** and **`IS NOT NULL`** conditions in the **`WHERE`** of the aggregate query:

```
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 translation of Q_5 makes no extra assumptions about the schema. Having additional information (e.g., that `c_custkey` is the key of `customer`) simplifies translation even further; see Section 5.

From $[\![\,]\!]^{=}$ to $[\![\,]\!]$, we define the translation $\text{toSQL}_{=}(\,)$ that specifies how to take a query E written under the syntactic equality semantics and translate it into a query $\text{toSQL}_{=}(E)$ where $[\![E]\!]^{=}_{D} = [\![\text{toSQL}_{=}(E)]\!]_D$ for every D. Similarly to before, we define translations of conditions and queries by mutual induction such that translations $\text{tr}^{\mathbf{t}}_{=}(\cdot)$, $\text{tr}^{\mathbf{f}}_{=}(\cdot)$ on conditions θ ensure

$$[\![\theta]\!]^{=}_{D,\eta} = \mathbf{t} \text{ if and only if } [\![\text{tr}^{\mathbf{t}}_{=}(\theta)]\!]_{D,\eta} = \mathbf{t}$$

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

and we go from E to $\text{toSQL}_{=}(E)$ by inductively replacing each condition θ with $\text{tr}^{\mathbf{t}}_{=}(\theta)$. The full details of the translations are in Figure 3.

Example 3. Following the previous example, we look at the translation $\text{toSQL}_{=}(\cdot)$ of queries Q_1–Q_5. In these translations we often see the condition of the form

$$\theta[t,t'] = \big(\texttt{isnull}(t) \wedge \texttt{isnull}(t')\big) \vee \big(\neg\texttt{isnull}(t) \wedge \neg\texttt{isnull}(t') \wedge t \doteq t'\big).$$

Query Q_1, which is equivalent to $\sigma_{\neg(R.A \doteq \text{any}(S))}(R)$, is translated as $\sigma_{\texttt{empty}(\sigma_{\theta[R.A,S.A]}(S))}(R)$. Query Q_2 is translated into the same expression. Query Q_3 is translated as

$$\varepsilon\Big(\pi_{X.A}(\sigma_{\theta[X.A,Y.A]}(\rho_{R.A\to X.A}(R) \times \rho_{R.A\to Y.A}(R)))\Big).$$

While Q_4 remains unchanged, in the subquery of Q_5, the condition $(c\_a > 0) \wedge \neg(c\_c \in \pi_{o\_c}(O))$ is translated to

$$(\neg\texttt{isnull}(c\_a) \wedge c\_a > 0) \wedge \Big(\texttt{empty}(\sigma_{\theta[o\_c,c\_c]}(O))\Big).$$

Notice that the size of expressions $\text{toSQL}_{2VL}(E)$ and $\text{toSQL}_{=}(E)$ is indeed linear in E.

4 RESTORING EXPECTED OPTIMIZATIONS

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 equivalent. And it seems that if conditions 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 is evaluated to $\mathbf{t}$. So equivalence of conditions θ_1 and θ_2, from SQL's point of view, means $[\![\theta_1]\!]_{D,\eta} = \mathbf{t} \Leftrightarrow [\![\theta_2]\!]_{D,\eta} = \mathbf{t}$ for all D and η. Of course in 2VL this is the same as stating that $[\![\theta_1]\!]_{D,\eta} = [\![\theta_2]\!]_{D,\eta}$ due to the fact that there are only two mutually exclusive truth values. In 3VL this is not the case however: we can have non-equivalent conditions that evaluate to $\mathbf{t}$ at the same time.

Basic:

$\text{tr}^{\mathbf{t}}_{=}(\theta) := \theta$ for $\theta := \mathbf{t} \mid \mathbf{f} \mid \texttt{isnull}(t)$

$\text{tr}^{\mathbf{t}}_{=}(t\ \omega\ t') := \neg\texttt{isnull}(t) \wedge \neg\texttt{isnull}(t') \wedge t\ \omega\ t'$ for $\omega \in \{<, >, \neq\}$

$\text{tr}^{\mathbf{t}}_{=}(t\ \omega\ t') := (\texttt{isnull}(t) \wedge \texttt{isnull}(t')) \vee (\neg\texttt{isnull}(t) \wedge \neg\texttt{isnull}(t') \wedge t\ \omega\ t')$ for $\omega \in \{\leq, \geq, \doteq\}$

$\text{tr}^{\mathbf{t}}_{=}(\texttt{empty}(E)) := \texttt{empty}(\text{toSQL}_{=}(E))$

$\text{tr}^{\mathbf{t}}_{=}(\bar{t}\ \omega\ \text{any}(E)) := \neg\texttt{empty}\,(\sigma_{\theta}\,(\text{toSQL}_{=}(E)))$

$\text{tr}^{\mathbf{t}}_{=}(\bar{t}\ \omega\ \texttt{all}(E)) := \texttt{empty}\,(\sigma_{\neg\theta}\,(\text{toSQL}_{=}(E)))$ where $\theta := \text{tr}^{\mathbf{t}}_{=}(\bar{t}\ \omega\ \ell(E))$

Comp.:

$\text{tr}^{\mathbf{t}}_{=}(\theta_1 \vee \theta_2) := \text{tr}^{\mathbf{t}}_{=}(\theta_1) \vee \text{tr}^{\mathbf{t}}_{=}(\theta_2)$

$\text{tr}^{\mathbf{t}}_{=}(\theta_1 \wedge \theta_2) := \text{tr}^{\mathbf{t}}_{=}(\theta_1) \wedge \text{tr}^{\mathbf{t}}_{=}(\theta_2)$

$\text{tr}^{\mathbf{t}}_{=}(\neg\theta) := \text{tr}^{\mathbf{f}}_{=}(\theta)$

Basic:

$\text{tr}^{\mathbf{f}}_{=}(\theta) := \neg\theta$ for $\theta := \mathbf{t} \mid \mathbf{f} \mid \texttt{isnull}(t)$

$\text{tr}^{\mathbf{f}}_{=}(t\ \omega\ t') := \texttt{isnull}(t) \vee \texttt{isnull}(t') \vee (\neg\texttt{isnull}(t) \wedge \neg\texttt{isnull}(t') \wedge \neg t\ \omega\ t')$ for $\omega \in \{<, >, \neq\}$

$\text{tr}^{\mathbf{f}}_{=}(t\ \omega\ t') := (\texttt{isnull}(t) \wedge \neg\texttt{isnull}(t')) \vee (\neg\texttt{isnull}(t) \wedge \neg\texttt{isnull}(t') \wedge \neg t\ \omega\ t') \vee (\texttt{isnull}(t') \wedge \neg\texttt{isnull}(t))$ for $\omega \in \{\leq, \geq, \doteq\}$

$\text{tr}^{\mathbf{f}}_{=}(\texttt{empty}(E)) := \neg\texttt{empty}(\text{toSQL}_{=}(E))$

$\text{tr}^{\mathbf{f}}_{=}(\bar{t}\ \omega\ \text{any}(E)) := \texttt{empty}(\sigma_{\theta}(\text{toSQL}_{=}(E)))$

$\text{tr}^{\mathbf{f}}_{=}(\bar{t}\ \omega\ \texttt{all}(E)) := \neg\texttt{empty}(\sigma_{\neg\theta}(\text{toSQL}_{=}(E)))$ where $\theta := \text{tr}^{\mathbf{t}}_{=}(\bar{t}\ \omega\ \ell(E))$

Comp.:

$\text{tr}^{\mathbf{f}}_{=}(\theta_1 \vee \theta_2) := \text{tr}^{\mathbf{f}}_{=}(\theta_1) \wedge \text{tr}^{\mathbf{f}}_{=}(\theta_2)$

$\text{tr}^{\mathbf{f}}_{=}(\theta_1 \wedge \theta_2) := \text{tr}^{\mathbf{f}}_{=}(\theta_1) \vee \text{tr}^{\mathbf{f}}_{=}(\theta_2)$

$\text{tr}^{\mathbf{f}}_{=}(\neg\theta) := \text{tr}^{\mathbf{t}}_{=}(\theta)$

Figure 3: $\text{tr}^{\mathbf{t}}_{=}(\cdot)$ and $\text{tr}^{\mathbf{f}}_{=}(\cdot)$ of basic and composite conditions

With the two-valued semantics eliminating this problem, we restore many query equivalences. It is natural to assume them for granted even though they are not true under 3VL, perhaps accounting for some typical programmer mistakes in SQL [7, 9]. In terms of RA_{SQL} expressions, these equivalences are as follows.

Proposition 1. *The following equivalences hold,* $\star \in \{2VL, =\}$*:*

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

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

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

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

for every RA^{REC}_{SQL} *expression* E*, tuple* $\bar{t}$ *of terms, condition* θ*, database* D*, and environment* η*.*

Neither of those is true in general under SQL's 3VL semantics.

5 TWO-VALUED SEMANTICS FOR FREE

Theorem 1 shows that every query written under 2VL semantics can be translated into a query that runs on existing RDBMSs and produces the same result. But ideally we want *the same* query to produce the right result, without any modifications. We now show that this happens very often, for a very large class of queries, including majority of benchmark queries used to evaluate RDBMSs. A key to this is the fact some attributes cannot have **NULL** in them, in particular those in primary keys and those declared as **NOT NULL**.

We provide a sufficient condition that a query produces the same result under the 2VL and 3VL semantics, for a given list of attributes that cannot be **NULL**. It is an easy observation that this equivalence in general is undecidable; hence we look for a sufficient condition. It is defined in two steps. The first tracks attributes in outputs of $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$ queries that are *nullable*, i.e., can have **NULL** in them. The second step restricts nullable attributes in queries.

Tracking nullable attributes. We define recursively the sequence $\mathtt{nullable}(E)$ of attributes of an expression E; those may have a **NULL** in them; others are guaranteed not to have any. We assume that $\mathtt{nullable}(R)$ for a base relation R is defined in the schema: it is the subsequence of attributes of R that are not part of R's primary key nor are declared with **NOT NULL**. Others are as follows:

- $\mathtt{nullable}(\varepsilon(E)) = \mathtt{nullable}(\sigma_\theta E) := \mathtt{nullable}(E)$;
- $\mathtt{nullable}(E_1 \times E_2) := \mathtt{nullable}(E_1) \cdot \mathtt{nullable}(E_2)$;
- For $\cup$ and $\cap$, assume that $\ell(E_1) = A_1 \cdots A_n$ and $\ell(E_2) = B_1 \cdots B_n$. Then $\mathtt{nullable}(E_1 \,\mathrm{op}\, E_2) = A_{i_1} \cdots A_{i_k}$ where i_j is on the list if: for $\cup$, either $A_{i_j} \in \mathtt{nullable}(E_1)$ or $B_{i_j} \in \mathtt{nullable}(E_2)$, and for $\cap$ both $A_{i_j} \in \mathtt{nullable}(E_1)$ and $B_{i_j} \in \mathtt{nullable}(E_2)$;
- $\mathtt{nullable}(E_1 - E_2) := \mathtt{nullable}(E_1)$;
- $\mathtt{nullable}(\mu R.E_1 \,\mathrm{op}\, E_2) = \mathtt{nullable}(E_1 \cup E_2)$;
- $\mathtt{nullable}(\pi_{t_1,\cdots,t_m}(E)) := t_{i_1} \cdots t_{i_k}$ where t_{i_j}'s are those terms that mention names in $\mathtt{nullable}(E)$.
- $\mathtt{nullable}(\mathtt{Group}_{\bar{M},\langle F_1(N_1),\cdots,F_m(N_m)\rangle}(E)) := \bar{M}'\bar{F}'$ where $\bar{M}'$ is the sequence obtained from $\bar{M}$ by keeping names that are in $\mathtt{nullable}(E)$, and $\bar{F}'$ is obtained from $F_1(N_1), \cdots, F_m(N_m)$ by keeping $F_i(N_i)$ whenever N_i is in $\mathtt{nullable}(E)$. In the last two rules, if renamings are specified, names are changed accordingly.

Restricting the nullable attributes. Their use is restricted under negation in selection conditions. We say that $\sigma_\theta(E)$ is *null-free* if for every sub-condition of θ of the form $\neg\theta'$ the following hold:

- the constant **NULL** does not appear in θ';
- for every atomic condition $\bar{t}\,\omega\,\bar{t}'$ in θ', no name in $\bar{t}, \bar{t}'$ is in $\mathtt{nullable}(E)$;
- for every atomic condition $\bar{t} \in F$, $\bar{t}\,\omega\,\mathtt{any}(F)$ or $\bar{t}\,\omega\,\mathtt{all}(F)$ in θ', the set $\mathtt{nullable}(F)$ is empty and no name in $\bar{t}$ is in $\mathtt{nullable}(E)$.

Theorem 2. *Let E be an RA^{REC}_{SQL} expression. If every subexpression of E of the form $\sigma_\theta(F)$ is null-free, then*

$$[\![E]\!]^{\mathbf{2VL}}_{D,\eta} \;=\; [\![E]\!]^{\mathbf{=}}_{D,\eta} \;=\; [\![E]\!]_{D,\eta}\,.$$

We clarify here that by a subexpression we mean expressions that are given by subtrees of the parse-tree of an expression.

Example 4. Consider the queries from our running example. Theorem 2 applies to Q_1, Q_4 if $R.A$ is a key, and to Q_2, Q_3 if both $R.A$ and $S.A$ are keys in R and S respectively. In query Q_5 from our running example we have the condition $(c\_a > 0) \wedge \neg(c\_c \in \pi_{o\_c}(O))$. Note that if c_c is a key it is not in $\mathtt{nullable}(Q_5)$. If, in addition, o_c is specified as **NOT NULL** in table O, then Theorem 2 says that for Q_5 its SQL and each of the two-valued semantics $[\![\cdot]\!]^{\mathbf{2VL}}$ and $[\![\cdot]\!]^{\mathbf{=}}$ coincide.

6 APPLICABILITY OF 2VL SEMANTICS

To gauge the level of applicability of our results, we answer two questions here: (a) how often do the 3VL and 2VL semantics cooincide, so the user can safely forget the *unknown*? and (b) when 2VL and 3VL semantics differ, which one is preferred by users?

How often do the semantics coincide?

To answer this, we look at popular relational performance benchmarks: TPC-H [44] containing 22 queries, and TCP-DS [43] containing 99 queries. A meticulous analysis of queries in those benchmarks shows that the following satisfy conditions of Theorem 2:

- *All* of TPC-DS queries;
- 21 out of 22 (i.e., 95%) TPC-H queries.

Thus, out of 121 benchmark queries, only one (Q16 of TPC-H) failed the conditions. It means that 120 of those 121 queries produce the same results under 2VL and 3VL semantics. These benchmarks were constructed to represent typical workloads of RDBMSs, meaning that many queries will not be affected by a switch to 2VL.

Which one is preferred by users?

For some queries, as we have seen, 3VL and 2VL do differ. The lack of those in benchmarks might be partly explained by the fact that those queries are written by experienced programmers who tend to avoid **NULL** pitfalls. When such queries do occur, is it more natural to expect SQL programmers to follow 3VL or 2VL?

To provide a preliminary answer to this question, we designed a short 10-question user survey. It should be noted that this approach is very common in social sciences, but in our field sociotechnological aspects perhaps do not get the attention they deserve, at least for forming research agenda (with a few exceptions though such as [40, 42]). This survey is intended to be a preliminary one, to gauge the level of potential applicability.

The survey started with queries where 3VL vs 2VL makes no difference and asked if users agree with SQL's output. It then showed three queries with different 3VL and 2VL results and asked users which one they preferred. It then showed three pairs of queries equivalent under 2VL but not 3VL and asked users whether they want these queries to be equivalent. Finally, it showed a foreign key constraint involving nulls, and asked whether it should hold.

Of 57 received responses, 81% came from database practitioners and 19% from academics. The results are shown in Figure 4. The first column is for queries where results coincide (i.e., the 2VL column here is the same as the 3VL column). The next three columns

are about outputs of queries, the following three are about query equivalences, and the last one about a foreign key constraint.

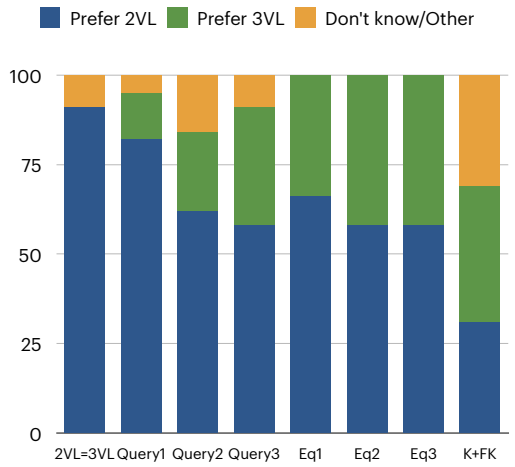

Figure 4: Results of the user survey

To summarize:

- When 2VL and 3VL coincide, by a 10-to-1 margin users agree with SQL's behavior.
- When 2VL and 3VL do not coincide, by a 3-to-1 margin (on average) users prefer query outputs under 2VL.
- For query equivalence, users still prefer 2VL but slighly less convincingly, by 60% to 40% on average.
- Foreign keys, when SQL switches from *true* to not *false*, are truly confusing to users who are almost evenly split between 2VL, 3VL, and "do not know".

We reiterate that these results should not be interpreted as the definitive answer on the right choice of the semantics, but rather as a strong indication of 2VL's feasibility and the need of more extensive user surveys to justify alternatives to (rather than outright replacement of) SQL's 3VL semantics.

7 ROBUSTNESS: OTHER MANY-VALUED LOGICS

We now show the robustness of the equivalence result, by proving that no other many-valued logic could have been used in place of SQL's 3VL in a way that would have altered the expressiveness of the language. In fact, SQL's 3VL, known well before SQL as *Kleene's logic* [6], is not the only many-valued proposed to handle nulls; there were others with 3,4,5, and even 6 values [12, 24, 34, 46]. It is thus natural to ask if using one of those would give us a more expressive language? We now give the negative answer, extending a partial result from [14] that was proved for first-order logic under several restrictions on the connectives of the logic. We extend it to the full language $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$, and eliminate previously imposed restrictions.

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 connectives* $\gamma : \mathbf{T}^{\mathrm{arity}(\gamma)} \to \mathbf{T}$. We assume that MVL includes at least the usual connectives $\neg, \wedge, \vee$ whose restriction to $\{\mathbf{t}, \mathbf{f}\}$ follows the rules of Boolean logic (so that queries on databases without nulls would not produce results that differ from their normal behavior).

The only condition we impose on MVL is that $\vee$ and $\wedge$ be associative and commutative; otherwise we cannot write conditions θ_1 **OR** $\cdots$ **OR** θ_k and θ_1 **AND** $\cdots$ **AND** θ_k without worrying about the order of conditions. Not having commutativity and associativity is also problematic for optimizing conditions in **WHERE**, as such optimizations assume Boolean algebra identities.

A semantics $[\![\,]\!]^{\mathrm{MVL}}$ of $\mathrm{RA}_{\mathrm{SQL}}$ conditions is determined by the semantics of comparisons $t\omega t'$; it then follows the connectives of MVL to express the semantics of complex condition, and the expressions of $\mathrm{RA}_{\mathrm{SQL}}$ and $\mathrm{RA}^{\mathrm{REC}}_{\mathrm{SQL}}$ follow the semantics of SQL. Such a semantics $[\![\,]\!]^{\mathrm{MVL}}$ is *SQL-expressible for atomic predicates* if:

(1) without nulls, it coincides with SQL's semantics $[\![\,]\!]$;
(2) for each truth value $\boldsymbol{\tau} \in \mathbf{T}$ and each comparison ω there is a condition $\theta_{\omega,\boldsymbol{\tau}}(t, t')$ that evaluates to $\mathbf{t}$ in SQL if and only if $t\ \omega\ t'$ evaluates to $\boldsymbol{\tau}$ in $[\![\,]\!]^{\mathrm{MVL}}$.

These conditions simply exclude pathological situations when conditions like $1 \leq 2$ evaluate to truth values other than $\mathbf{t}$, $\mathbf{f}$, or when conditions like "**NULL** $\doteq n$ evaluates to $\mathbf{t}$" are not expressible in SQL (say, **NULL** $\doteq n$ is $\mathbf{t}$ iff the nth Turing machine in some enumeration halts on the empty input). Anything reasonable is permitted by being expressible.

Theorem 3. *For a many-valued logic* MVL *in which* $\wedge$ *and* $\vee$ *are associative and commutative, let* $[\![\,]\!]^{\mathrm{MVL}}$ *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.*

Different many-valued semantics are not pure theoretical inventions; for example, in MS SQL Server one can switch off the `ansi_nulls` option to obtain a different MVL of nulls that will be covered by Theorem 3.

8 CONCLUSIONS

We showed 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 with all the legacy SQL code based on 3VL, the ultimate goal is not to replace it but rather propose alternatives. Such alternatives can apply not only to SQL but also to newly designed query languages such as GQL for graph data [19, 23].

As for future lines of research, one is to sharpen the definition of the language to get even closer to everyday SQL. Another direction is to adapt works like [21, 27] to return results with certainty guarantees, but under 2VL as opposed to SQL's semantics. And most importantly we shall explore avenues of having some of these proposals adapted in relational DBMSs.

ACKNOWLEDGMENTS

This work was supported by a Leverhulme Trust Research Fellowship; EPSRC grants N023056 and S003800; and Agence Nationale de la Recherche project ANR-21-CE48-0015 (Verigraph). We are grateful to Molham Aref and Paolo Guagliardo for helpful discussions,

and to survey respondents for their willingness to participate. Part of this work was done while the second author was affiliated with ENS, PSL University.

REFERENCES

[1] RelationalAI Documentation. https://docs.relational.ai. Accessed: April 2023.
[2] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Software Eng.*, 14(7):879–885, 1988.
[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, K. Weitz, A. Cheung, and D. Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017.
[12] 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.
[13] M. Console, P. Guagliardo, and L. Libkin. On querying incomplete information in databases under bag semantics. In *IJCAI*, pages 993–999, 2017.
[14] M. Console, P. Guagliardo, and L. Libkin. Propositional and predicate logics of incomplete information. *Artif. Intell.*, 302:103603, 2022.
[15] H. Darwen and C. J. Date. The third manifesto. *SIGMOD Record*, 24(1):39–49, 1995.
[16] C. J. Date. *Database in Depth - Relational Theory for Practitioners*. O'Reilly, 2005.
[17] 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.
[18] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.
[19] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoc, M. Wu, and F. Zemke. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD*, pages 2246–2258. ACM, 2022.
[20] A. Eisenberg and J. Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Rec.*, 28(1):131–138, 1999.
[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. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. Rogova, and D. Vrgoč. A researcher's digest of GQL. In *ICDT*, volume 255 of *LIPIcs*, pages 1:1–1:22, 2023.
[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] L. Libkin and L. Peterfreund. Handling SQL nulls with two-valued logic. *CoRR*, abs/2012.13198, 2022.
[37] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3):513–534, 1991.
[38] 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.
[39] PostgreSQL Documentation, Version 9.6.1. `www.postgresql.org/docs/manuals`, 2016.
[40] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, 2017.
[41] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.
[42] E. Toussaint, P. Guagliardo, L. Libkin, and J. Sequeda. Troubles with nulls, views from the users. *Proc. VLDB Endow.*, 15(11):2613–2625, 2022.
[43] Transaction Processing Performance Council. *TPC Benchmark™ DS Standard Specification*, 2017. Revision 3.2.0.
[44] Transaction Processing Performance Council. *TPC Benchmark™ H Standard Specification*, 2018. Revision 2.18.0.
[45] J. Van den Bussche and S. Vansummeren. Translating SQL into the relational algebra. Course notes, Hasselt University and Université Libre de Bruxelles, 2009.
[46] 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.
[47] C. Zaniolo. Database relations with null values. *JCSS*, 28(1):142–166, 1984.