SQL Nulls and Two-Valued Logic

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3584372

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '23)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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.

KEYWORDS

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:


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 NULLS 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 NOTs are removed, these queries differ in SQL: if R = {1, NULL} and S = {NULL}, then Q1 returns no tuples, while Q2 returns {1, 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 = {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.
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 3VL standard. To achieve this, we need to fulfill the following criteria.

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 I° and I°a11(E) with the semantics of SQL’s ANY and ALL (they check whether I° holds for some, respectively all, I 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 of values. To handle languages with aggregation, we need to distinguish between inconsistent data were also considered in the AI literature; see, e.g., [4, 22, 25]. Proposals for eliminating this idiosyncrasies of SQL’s syntax, it is not an ideal language

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 \( f(t_1, \ldots, t_k) \) where \( f : \text{Num}^k \rightarrow \text{Num} \) is a \( k \)-ary numerical function (e.g., addition or multiplication) and \( t_1, \ldots, t_k \) 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 \( \Omega \) of numerical and aggregate functions. We assume the standard comparison predicates \( \leq, \lt, \geq, \gt \) on numerical values are available \(^1\).

Given a schema \( S \) and such a collection \( \Omega \), the syntax of RSQL expressions and conditions over \( S \cup \Omega \) is given in Fig. 1, where \( R \) ranges over relation symbols in \( S \), each \( t_i \) is a term, each \( N_i, N'_i \) is a name, each \( N \) is a tuple of names, and each \( F_i \) is an aggregate function. In the generalized projection and in the grouping/aggregate, the parts in the squared brackets (i.e., \([-N_i]\) and \([-N'_i]\)) 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 \( x_1 < y_1 \vee (x_1 = y_1 \wedge x_2 < y_2) \).

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

\[
[ E ]_{D,P}
\]

\(^1\)We focus, without loss of generality, on these predicates; our results apply also for predicates of higher arities, e.g., BETWEEN in SQL.
defines the result of the evaluation of expression $E$ on database $D$ under the environment $\eta$. The environment $\eta$ is a partial mapping from the set Name of names to the union Val $\cup$ Num $\cup$ [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 RA$_{SQL}$ and a database $D$, the value of $E$ in $D$ is defined as $[E]_D$, where $\emptyset$ is the empty mapping (i.e., the top level expression has no parameters).

Similarly to SQL queries, each RA$_{SQL}$ 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 $\ast$ FROM $R$. Generalized projection captures SQL’s SELECT clause. Each term $t_j$ is evaluated, optionally renamed, and added as a column to the result. For example

- **SELECT** $A_1, B_1, A_2 B_2$ AS $C_1$, $A_3 B_3$ AS $D$ FROM $R$ is written as $\pi_{A, B, \text{add2}(A)}(\cdot, \text{mult}(A, B) \rightarrow D)(R)$, where add2$((x, y)) := x + 2$ and mult $(x, y) := x \cdot y$. Projection follows SQL’s bag semantics with terms evaluated along with their multiplicity. For instance if $(2, 3) \in \mathbb{E}$ and $(1, 6) \in \mathbb{E}$ then the result of $\pi_{\text{mult}(A, B)}(R)$ 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.

<table>
<thead>
<tr>
<th>$\land$</th>
<th>$\lor$</th>
<th>$\neg$</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>u</td>
<td>t</td>
<td>u</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>u</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>u</td>
</tr>
</tbody>
</table>

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

The condition $i \equiv i'$, where $i = (t_1, \ldots, t_m)$ and $i' = (t'_1, \ldots, t'_m)$ that compares tuples of terms is the abbreviation of the conjunction $\land_{j=1}^m t_j \equiv t'_j$, and the comparison $i \not\equiv i'$ abbreviates $\lor_{j=1}^m t_j \not\equiv t'_j$.

Comparisons $<, \leq, >, \geq$ of tuples are defined lexicographically.

The condition $i \in E$, 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 $t_1, \ldots, t_n$, then $i \in E$ stands for the disjunction $\lor_{j=1}^n t_j \equiv i$. Other predicates not typically included in RA presentation, though included here for direct correspondence with SQL, are ALL and ANY comparisons. The condition $i \in \text{any}(E)$ checks whether there exists a tuple $t'$ in $E$ so that $i \equiv t'$ holds, where $\omega$ is one of the allowed comparisons. Likewise, $i \in \text{all}(E)$ checks whether $i \equiv t'$ holds for every tuple $t'$ in $E$ (in particular, if $E$ returns no tuples, this condition is true). If $\omega$ is $\equiv$ or $\not\equiv$, conditions with any and all are applicable at either ordinary or numerical type; if $\omega$ is one of $<, \leq, >, \geq$, then all the components of $i$ and all attributes of $E$ are of numerical type.

The condition $\emptyset(E)$ 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 $\theta$ for each tuple, and keeps tuples for which $\theta$ 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 $\ell$ 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 in 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 $\theta$ for each tuple, and keeps tuples for which $\theta$ 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).
Finally, we describe the operator Group$_N.M(E)$. The tuple $N$ lists attributes in Group BY, and the $i$th coordinate of $M$ is of the form $F_i(N_i)$ where $F_i$ is an aggregate over the numerical columns $N_i$ optionally renamed $N_j'$ if $[\rightarrow N_j']$ is present. For example

\begin{verbatim}
SELECT A, COUNT(B) AS C, SUM(B) FROM R GROUP BY A
\end{verbatim}

will be expressed by Group$_A(F_{\text{COUNT}}(B) \rightarrow C)FS_{\text{SUM}}(B)(R)$ where $F_{\text{COUNT}}((a_1, \ldots, a_n)) := n$ and $F_{\text{SUM}}((a_1, \ldots, a_n)) := a_1 + \cdots + a_n$. Note that $N$ could be empty; this corresponds to computing aggregates over the entire table, without grouping, for example, as in

\begin{verbatim}
SELECT COUNT(B), SUM(B) FROM R.
\end{verbatim}

\textbf{Example 1.} We start by showing how queries $Q_1$–$Q_4$ from the introduction are expressible in RASQL:

\begin{align*}
Q_1 &= \sigma_{c_{\text{acctbal}}} (R) \\
Q_2 &= \sigma_{\exists A.S(E)} (R) \\
Q_3 &= \rho \left( \sigma_{\forall A.S(A)} (R) \right) \\
Q_4 &= \pi \left( \rho_{\forall A.S(A)} (R) \right)
\end{align*}

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]:

\begin{verbatim}
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 c_custkey FROM orders))
GROUP BY c_nationkey
\end{verbatim}

Below we use abbreviations $C$ for customer and $O$ for orders, and abbreviations for attributes like $c_n$ for $c_{\text{nationkey}}$ etc. The NOT IN condition in the subquery is then translated as $(\neg c_\in \pi_0.C(O))$, the whole condition is translated as $\theta := (c_a > 0) \land \neg(c_c \in \pi_0.c(O))$ and the aggregate subquery becomes

\begin{align*}
Q_{agg} &= \text{Group}\theta (F_{\text{avg}}(c_a) (\pi_{c_a}(\sigma_\theta (C))))
\end{align*}

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

\begin{align*}
\text{Group}_{c_n}(F_{\text{COUNT}}(c_a) (\sigma_{c_a > \text{any}(Q_{agg})}(C))).
\end{align*}

Putting everything together, we have the final RASQL expression:

\begin{align*}
\text{Group}_{c_n}(F_{\text{COUNT}}(c_a) (\sigma_{c_a > \text{any}(Q_{agg})}(C)) (C)).
\end{align*}

\textbf{ADDITION Recursion.} We now incorporate recursive queries, a feature added in the SQL 1999 standard with its \textit{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].

\textbf{Syntax of RASQL.} Recall that $\cup$ stands for bag union, i.e., multiplicities of tuples are added up, as in SQL's \texttt{UNION ALL}. We also need the operation $\cup_1 \cup_2$ defined as $\epsilon(\cup_1 \cup_2)$, i.e., union in which a single copy of each tuple is kept. This corresponds to SQL's \texttt{UNION}.

An RASQL expression is defined with the grammar of RASQL in Fig. 1 with the addition of the constructor $\mu.RE$ where $R$ is a fresh relation symbol (i.e., not in the schema) and $E$ is an expression of the form $E_1 \cup_2 E_2$ or $E_1 \cup E_2$ where both $E_1$ and $E_2$ are RASQL 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.

\textbf{Semantics of RASQL.} Similarly to the syntactic definition, we distinguish between the two cases.

For $\mu.R.E_1 \cup_2 E_2$, the semantics $[\mu.R.E_1 \cup_2 E_2]_{D,E}$ is defined by the following iterative process:

\begin{align*}
(1) & \quad \text{RES}_0, R_0 := E_1_{D,E} \\
(2) & \quad R_{i+1} := [E_2]_{D,R_i,E} \cup \text{RES}_i \cup R_{i+1}
\end{align*}

with the condition that if $R_i = \emptyset$, then the iteration stops and $\text{RES}_i$ is returned.

For $\mu.R.E_1 \cup_2 E_2$, the semantics is defined by a different iteration:

\begin{align*}
(1) & \quad \text{RES}_0, R_0 := [\epsilon(E_1)]_{D,E} \\
(2) & \quad R_{i+1} := [\epsilon(E_2)]_{D,R_i,E} \cup \text{RES}_i \cup R_{i+1}
\end{align*}

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.

\section{Eliminating Unknown}

To replace 3VL with Boolean logic, we need to eliminate the unknown truth value. In SQL, $\texttt{u}$ arises in $\texttt{WHERE}$ which corresponds to conditions in RASQL. It appears as the result of evaluation of comparison predicates such as $=, \leq, \not\equiv$ etc. Consequently, it also arises in \texttt{IN}, \texttt{ANY} and \texttt{ALL} conditions for subqueries.

In comparisons, $\texttt{u}$ appears due to the rule that if one parameter is \texttt{NULL}, then the value of the predicate is $\texttt{u}$. Thus, we need to change this rule, and to say what to do when one of the parameters is \texttt{NULL}. In doing so, we are guided by SQL's existing semantics of conditions in $\texttt{WHERE}$. While those can evaluate to $\texttt{t, f, u}$ or $\texttt{u}$, in the end only the true values are kept: that is, $\texttt{u}$ and $\texttt{f}$ 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 $\texttt{NULL} \equiv \texttt{NULL}$ and $\texttt{NULL} \not\equiv \texttt{NULL}$ evaluate to $\texttt{f}$, and thus $\texttt{NULL} \not\equiv \texttt{NULL}$ cannot be equivalent to $\neg(\texttt{NULL} \equiv \texttt{NULL})$. This however can easily be resolved by treating conditions consistent with syntactic equality differently.
3.1 The two-valued semantics \([\text{2VL}]\) and \([\text{2V}]\)

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

\[
[t \circ t']_{\text{2VL}} := \begin{cases} 
  t & \text{if } \langle t, t' \rangle_D \not= \text{NULL}, \text{ and } \langle t \rangle_D \not= \langle t' \rangle_D \\
  f & \text{otherwise}
\end{cases}
\]

The rest of the semantics is exactly the same as before. Note that in conditions like \(t \equiv t'\), or \(t \in E\), or \(t \circ 1\) any \(E\), the conjunctions and disjunctions will be interpreted as the standard Boolean ones, since \(u\) no longer arises.

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

\[
[t \circ t']_{\text{2V}} := \begin{cases} 
  t & \text{if } \langle t, t' \rangle_D = t \text{ or } \langle t \rangle_D = \langle t' \rangle_D = \text{NULL} \\
  f & \text{otherwise}
\end{cases}
\]

and keeping the rest as in the definition of \([\text{2VL}]\). The only difference is that now conditions \text{NULL} \leq \text{NULL}, \text{NULL} \leq \text{NULL},\text{ and } \text{NULL} \geq \text{NULL} evaluate to true.

3.2 Capturing SQL with \([\text{2VL}]\) and \([\text{2V}]\)

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 \([\ast]\) of queries captures the semantics \([\cdot]\) 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 each database \(D\),
   \[ [F]_D = [E]_D \; . \]
3. For every expression \(E\) of RA SQL, and every database \(D\) without nulls, \([E]_D = [E]_D^\prime\).

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 \([\text{2VL}]\) and \([\text{2V}]\) semantics of RA REC SQL expressions, and of RA SQL expressions, capture their SQL semantics \([\cdot]\) efficiently.

Note that the capture statement for RA SQL is not a corollary of the statement of RA REC SQL.

---

### Running 2VL on existing RDBMSs

We sketch one direction of the proof of Theorem 1, namely from \([\text{2VL}]\) and \([\text{2V}]\) to \([\cdot]\).

From \([\text{2VL}]\) to \([\cdot]\), we define the translation toSQL that specifies how to take a query \(Q\) written under the 2VL semantics that conflates \(u\) with \(f\) and translate it into a query toSQL(E) that gives the same result when evaluated under the usual SQL semantics: \([E]_{\text{2VL}} = [\text{toSQL}_{\text{2VL}}(E)]_{\text{D}}\) for every database \(D\). Thus, toSQL(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 \(\text{tr}_{\text{2VL}}(\cdot), \text{tr}_{\text{2V}}(\cdot)\) on conditions \(\theta\) ensure

\[
[\theta]_{\text{2VL}} = t \text{ if and only if } [\text{tr}_{\text{2VL}}(\theta)]_{\text{D}} = t \; .
\]

Making \([\theta]_{\text{2VL}}\) produces only \(t\) and \(f\). Then we go from \(E\) to toSQL(E) by inductively replacing each condition \(\theta\) with \(\text{tr}_{\text{2VL}}(\theta)\). The full details of the translations are in Figure 2.

---

**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 toSQL(E) is \(\sigma_{\text{isnull}(R.A)} \land (R.A = \text{isnull}(S.A)) S(R)\). 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 toSQL₂VL(Q₅), the condition \((c \cdot a > 0) \land \neg(c \cdot c \in \pi_{\alpha \cdot c}(O))\) in the subquery is translated by \(\text{tr}_{SQL}^{t}()\) as
\[
(c \cdot a > 0) \land \left(\text{isnull}(c \cdot c) \lor \neg(c \cdot c \in \sigma_{\text{isnull}(c \cdot c)}(\pi_{\alpha \cdot c}(O)))\right)
\]
which is then used in the aggregate subquery \(Q_{\text{agg}}\) (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 \(c\cdot\text{acctb} > 0.0\) AND \((c\cdot\text{custkey} \text{ IS NULL OR c\cdot\text{custkey NOT IN (SELECT o\cdot\text{custkey FROM orders WHERE o\cdot\text{custkey IS NOT NULL})}}})\)

This translation of \(Q_5\) makes no extra assumptions about the schema. Having additional information (e.g., that \(c \cdot \text{custkey}\) is the key of \(c\cdot\text{customer}\)) simplifies translation even further; see Section 5.

From \([\cdot]^{\text{sql}}\) to \([\cdot]^{\text{2vl}}\), we define the translation to\(\text{SQL}^{\cdot\text{}}()\) that specifies how to take a query \(E\) written under the syntactic equality semantics and translate it into a query to\(\text{SQL}^{\cdot\text{}}(E)\) where \([E]^{D}_{\eta} = [\text{toSQL}^{\cdot\text{}}(E)]^{D}_{\eta}\) for every \(D\). Similarly to before, we define translations of conditions and queries by mutual induction such that translations \(\text{tr}_{SQL}^{\cdot\text{}}(), \text{tr}_{\text{2vl}}^{\cdot\text{}}()\) on conditions \(\theta\) ensure
\[
[\theta]^{D}_{\eta} = t\text{ if and only if } [\text{tr}_{\text{2vl}}^{\cdot\text{}}(\theta)]^{D}_{\eta} = t
\]
and we go from \(E\) to \(\text{toSQL}^{\cdot\text{}}(E)\) by inductively replacing each condition \(\theta\) with \(\text{tr}_{\text{2vl}}^{\cdot\text{}}(\theta)\). The full details of the translations are in Figure 3.

Example 3. Following the previous example, we look at the translation to\(\text{SQL}^{\cdot\text{}}()\) of queries \(Q_1 \setminus Q_5\). In these translations we often see the condition of the form
\[
\theta[t, t'] = (\text{isnull}(t) \land \text{isnull}(t')) \lor (-\text{isnull}(t) \land \neg\text{isnull}(t') \land t \equiv t').
\]

Query \(Q_1\), which is equivalent to \(\sigma_{\neg(A \land \text{any}(S))}(R)\), is translated as \(\text{empty}(\sigma_{\text{RASAss}(S)}(R))\). Query \(Q_5\) is translated into the same expression. Query \(Q_1\) is translated as
\[
[\theta[t, t']] = f(\pi_{X \cdot A}(\sigma_{\theta}(\pi_{X \cdot A}(R) \times R \times \pi_{R \cdot Y}(A)))).
\]

While \(Q_1\) remains unchanged, in the subquery of \(Q_5\), the condition \((c \cdot a > 0) \land \neg(c \cdot c \in \pi_{\alpha \cdot c}(O))\) is translated to
\[
(-\text{isnull}(c \cdot a) \land c \cdot a > 0) \lor \left(\text{empty}(\sigma_{\theta}(c \cdot c, c, c))(O)\right).
\]

Notice that the size of expressions to\(\text{SQL}^{\cdot\text{}}(E)\) and to\(\text{SQL}^{\cdot\text{}}(E)\) 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 \(\text{NOT}\) from both of them, then they are equivalent. And it seems that if conditions \(\theta_1\) and \(\theta_2\) 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 \(\text{WHERE}\) clause is to keep tuples for which the condition is evaluated to \(t\). So equivalence of conditions \(\theta_1\) and \(\theta_2\), from SQL’s point of view, means \([\theta_1]^{D}_{\eta} = t \iff [\theta_2]^{D}_{\eta} = t\).

Figure 3: \(\text{tr}_{\text{2vl}}^{\cdot\text{}}()\) and \(\text{tr}_{\text{sql}}^{\cdot\text{}}()\) of basic and composite conditions

\([\theta_1]^{D}_{\eta} = t\) for all \(D\) and \(\eta\). Of course in 2VL this is the same as stating \([\theta_1]^{D}_{\eta} = [\theta_2]^{D}_{\eta}\) 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 \(t\) at the same time.

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 RA₅SQL expressions, these equivalences are as follows.

**Proposition 1.** The following equivalences hold, \(\star \in \{2\text{VL}, \star\}\):

1. \([\sigma_{\theta}(E)]^{D}_{\eta} = [E \setminus \sigma_{\neg\theta}(E)]^{D}_{\eta}\)
2. \([i \in E]^{D}_{\eta} = \text{f} \iff \text{f and only if } [\sigma_{\text{null}(t)}(E)]^{D}_{\eta} = \emptyset\)
3. \([i \cdot \text{any}(E)]^{D}_{\eta} = \text{f} \iff \text{f and only if } [\sigma_{\text{null}(t)}(E)]^{D}_{\eta} = \emptyset\)
4. \([i \cdot \text{all}(E)]^{D}_{\eta} = \text{t} \iff \text{t and only if } [\sigma_{\text{null}(t)}(E)]^{D}_{\eta} = \emptyset\)

for every RA₅SQL expression \(E\), tuple \(t\) of terms, condition \(\theta\), database \(D\), and environment \(\eta\).
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 RA_REC 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 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 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:

- nullable(r(E)) = nullable(a)(E) := nullable(E);
- nullable(E1 × E2) := nullable(E1) × nullable(E2);
- nullable(∪ E = null) and for ∩ both Ej ∈ nullable(E1) and Bi j ∈ nullable(E2), and for (E1 op E2) = A1 ∪ A2 where i j is on the list: for ∩, either Ai j ∈ nullable(E1) or Bi j ∈ nullable(E2), and for ∪ both Aij ∈ nullable(E1) and Bi j ∈ nullable(E2);
- nullable(E1 − E2) := nullable(E1);
- nullable(μF.E1 op E2) = nullable(E1 ∪ E2);
- nullable(σt1,···,tk.E) := t1,···,tk where t1,···,tk are those terms that mention names in nullable(E).
- nullable(σF = null E) := M1 × M2 where M1 is the sequence obtained from M by keeping names that are in nullable(E), and M2 is obtained from F1, F2,···,Fm by keeping F1 whenever N1 is in 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 a(E) is null-free if for every sub-condition of φ of the form ¬φ’ the following hold:

- the constant NULL does not appear in φ’;
- for every atomic condition I φ I’ in φ’, no name in I, I’ is in nullable(E);
- for every atomic condition I ∈ F, I φ any (F) or I φ all(F) in φ’, the set nullable(F) is empty and no name in I is in nullable(E).

Theorem 2. Let E be an RA_REC expression. If every subexpression of E of the form a(F) is null-free, then

\[ [E]^{2VL}_{D,H} = [E]^{3VL}_{D,H} = [E]^{t}_{D,H} \]

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 Q1, Q4 if R.A is a key, and to Q2, Q5 if both R.A and S.A are keys in R and S respectively. In query Q5 from our running example we have the condition (c_a > 0) ∧ ¬(c_e ∈ πo_c(O)). Note that if c_e is a key it is not in nullable(Q5). If, in addition, a_e is specified as NOT NULL in table O, then Theorem 2 says that for Q5 its SQL and each of the two-valued semantics \([·]^{2VL}\) and \([·]^{t}\) 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 coincide, 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 socio-technological 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
We now show the robustness of the equivalence result, by proving the usual connectives $\land$ and $\lor$ are associative and commutative; otherwise we cannot write conditions $\theta_1 \lor \cdots \lor \theta_k$ and $\theta_1 \land \cdots \land \theta_k$ 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 $[[\_]]^{\text{MVL}}$ of $\text{RA}_{\text{SQL}}$ conditions is determined by the semantics of comparisons $t = t'$; it then follows the connectives of MVL to express the semantics of complex condition, and the expressions of $\text{RA}_{\text{SQL}}$ and $\text{RA}_{\text{REC}}$ follow the semantics of SQL. Such a semantics $[[\_]]^{\text{MVL}}$ is SQL-expressible for atomic predicates if:

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

These conditions simply exclude pathological situations when conditions like $1 \leq 2$ evaluate to truth values other than $t, f,$ or when conditions like "null $\neq n$ evaluates to $t$" are not expressible in SQL (say, NULL $\neq n$ is $t$ 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 $\land$ and $\lor$ are associative and commutative, let $[[\_]]^{\text{MVL}}$ be a semantics of $\text{RA}_{\text{SQL}}$ or $\text{RA}_{\text{REC}}$ 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 ANS_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,