A Formal Semantics of SQL Queries, Its Validation, and Applications

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the VLDB Endowment (PVLDB)

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.
A Formal Semantics of SQL Queries, Its Validation, and Applications

Paolo Guagliardo  
School of Informatics  
University of Edinburgh  
pguaglia@inf.ed.ac.uk

Leonid Libkin  
School of Informatics  
University of Edinburgh  
libkin@inf.ed.ac.uk

ABSTRACT

While formal semantics of theoretical languages underlying SQL have been provided in the past, they all made simplifying assumptions ranging from changes in the syntax to omitting bag semantics and nulls. This situation is reminiscent of what happens in the field of programming languages, where semantics of formal calculi underlying main features of languages are abundant, but formal semantics of real languages that people use are few and far between.

We take the basic class of SQL queries – essentially SELECT-FROM-WHERE queries with subqueries, set/bag operations, and nulls – and define a formal semantics for it, without any departures from the real language. Already this fragment requires decisions related to the data model and handling variable names that are normally disregarded by simplified semantics. To justify our choice of the semantics, we validate it experimentally on a large number of randomly generated queries and databases.

We give two applications of the semantics. One is the first formal proof of the equivalence of basic SQL and relational algebra that extends to bag semantics and nulls. The other application looks at two simple examples. The first option is to use a NOT IN subquery: every SQL query in our fragment can be evaluated under the usual two-valued Boolean semantics.

1. INTRODUCTION

Providing a formal semantics of a language is a major task in programming language research [17, 19, 26]. It enables one to formally reason about languages, verify correctness of programs, and it becomes an important tool in designing language extensions as well as new languages. Given the complexities of real-life languages, it is very common to abstract the core of a language by means of a well-behaved theoretical calculus and study its semantics. Providing the semantics of a real language is typically a much harder task as it needs to account for all its idiosyncrasies. This has been done for several languages [1, 13, 18, 25, 28, 29]; the difference is that to describe such a formal semantics one needs a book, rather than a paper (or sometimes even a book to explain what the first book said [24]).

When it comes to the main query language used by relational DBMSs – SQL – we have the Standard [20], but it cannot serve as a formal semantics, as it is written in natural language. In fact, it is well known that different vendors of RDBMSs interpret various points of the Standard differently (see, e.g., [4, 21]). A natural language description does not lend itself to proper formal reasoning that is necessary to derive language equivalences and optimization rules.

Given the problems of using the Standard as the definition of formal semantics, there have been attempts to formalize SQL. Several of them go via translating SQL queries into relational algebra (RA), for which formal semantics has been properly defined. Database texts (e.g., [2, 30, 14]) of course provide examples of SQL-to-RA translations, but at an informal level. Formal translations did appear [6, 32] but they imposed rather strong restrictions: for example, queries interpreted under set semantics of queries, absence of nulls, disallowed subqueries in FROM, etc.

A different line of work attempted to provide a formal semantics of SQL directly, but all such attempts have fallen short of the real SQL. An early paper [27] looked only at set semantics, and the more recent and rigorous formalization [7, 8] – designed to prove equivalences of queries with the help of a proof assistant – did not include null values and used a reconstruction of the language, thus not accounting for some of the trickier aspects of variable binding. Other attempts were made in the programming languages community [23, 33] but they too restricted the language significantly: for example, [23] works essentially with RA, rather than SQL, and set semantics, while [33] disallows nested subqueries in both FROM and WHERE and uses list semantics.

To see why restrictions such as the absence of nulls or set semantics deviate significantly from the behavior of real language, and could even lead to wrong equivalences among queries, we look at two simple examples.

Example 1. In standard literature translations from SQL to RA, one converts $\text{IN}$ and $\text{NOT IN}$ subqueries into $\text{EXISTS}$ and $\text{NOT EXISTS}$ subqueries. But such conversions do not always work in real life. To see this, take two relations $R$ and $S$ with a single attribute $A$ and compute their difference $R - S$. The first option is to use a $\text{NOT IN}$ subquery:

$$Q_1: \text{SELECT } R.A \text{ FROM } R \text{ WHERE } R.A \not\in \{ \text{SELECT } S.A \text{ FROM } S \}$$
By using the conversion of \texttt{NOT IN} into \texttt{NOT EXISTS}, as for example \cite{6, 32} would suggest, we get a seemingly equivalent query:

\begin{verbatim}
Q2: SELECT R.A FROM R WHERE NOT EXISTS ( 
  SELECT * FROM S WHERE S.A = R.A )
\end{verbatim}

And of course SQL gives us a direct way of writing the difference query:

\begin{verbatim}
Q3: SELECT R.A FROM R EXCEPT SELECT S.A FROM S
\end{verbatim}

While these may look equivalent, they are not. Take a database \(D\) with \(R = \{1, \text{NULL}\}\) and \(S = \{\text{NULL}\}\), where \texttt{NULL} denotes SQL’s null value. Then \(Q_1(D) = \emptyset\), while \(Q_2(D) = \{1, \text{NULL}\}\) and \(Q_3(D) = \{1\}\).

Thus, queries that database theory would want us to view as equivalent are hardly equivalent in real life.

\textbf{Example 2.} While many of the existing SQL-to-RA translations and SQL semantics \cite{6, 27, 32, 23} assume that relations are sets of tuples, SQL tables are bags, or multisets. For instance, referring to relations from the previous example, \texttt{SELECT S.A FROM S} produces \{NULL\}, while \texttt{SELECT S.A FROM S, R} produces \{NULL, NULL\}, i.e., two occurrences of it. This of course needs to be taken into account in producing proper semantics.

Our main goal is to give the formal semantics of the core of real SQL and provide evidence that this semantics correctly captures the behavior of the language. Here we look at the basic language that includes \texttt{SELECT-FROM-WHERE} queries without aggregation but with subqueries in both \texttt{FROM} and \texttt{WHERE}. Boolean operations, null values, and duplicate elimination. This is already an expressive fragment (it captures all relational algebra queries, for instance) and it illustrates many issues that need to be addressed in defining a formal semantics, crucially the variable binding rules.

Once we define the semantics, we need to justify it. How do we know that this is the right semantics of SQL? Since this is the first fully formal attempt to specify the basic fragment of the Standard, there is nothing to rely on to formally prove correctness. Thus, we believe that the only way to convince oneself that the proposed semantics really models SQL is \textit{experimental}: one needs to implement it, and compare with the behavior of a very large number of queries on a real RDBMS.

But this step is not as easy as it seems: all major RDBMSs stay very close to the Standard, and yet have subtle differences. Let us illustrate this by an example.

\textbf{Example 3.} One of the standard textbook assumptions of the relational model is that attributes in tables do not repeat. However, SQL makes it very easy to create such tables, for instance, by writing \texttt{SELECT A, A FROM R} (again, referring to relation \(R\) with attribute \(A\)). But how about using such an expression in subqueries, e.g.,

\begin{verbatim}
SELECT * FROM (SELECT R.A, R.A FROM R) AS T
\end{verbatim}

This expression will be accepted by Postgres, but will result in compile-time error if one uses some of the commercial RDBMSs. On the other hand, if this query is used as a subquery in

\begin{verbatim}
SELECT * FROM R WHERE EXISTS
  (SELECT * FROM (SELECT R.A, R.A FROM R) AS T)
\end{verbatim}

then suddenly it is fine, even with RDBMSs where the subquery alone refused to compile. Thus, not only is * a tricky feature to model, it also shows that no single semantics will account for all the existing RDBMSs, even for the core language.

Fortunately, differences between real implementations are minor and well documented. It is easy to adjust the general semantics to account for little quirks of individual implementations. Having done so, we are able to provide experimental validation of the semantics, using two different RDBMSs, and two adjustments of the semantics.

We then move to applications of the semantics which concern language equivalences and expressiveness. The first is to provide, for the first time, a formal proof that basic SQL can be captured by relational algebra. As mentioned earlier, existing translations \cite{6, 32} used a much simplified model, and procedural languages employed by RDBMSs go beyond RA capabilities. Now, with the formal semantics of SQL, we can formally prove this folklore (but so far unproven) result.

For our second application, we look at SQL’s logic of null values. For operations involving nulls, SQL operates with three truth values \cite{12}: true (\(t\)), false (\(f\)), and unknown (\(u\)). The particular three-valued logic (3VL) used by SQL is the well known Kleene logic \cite{5}. In fact, SQL mixes it with the usual Boolean logic: conditions in \texttt{WHERE} are evaluated under 3VL, but then \(f\) and \(u\) are conflated, and only tuples for which the condition is \(t\) are returned.

It is commonly believed that 3VL is really necessary to model SQL behavior. Using the formal semantics, we show that this is not so: basic SQL queries have the same power under 3VL and under the usual two-valued Boolean logic. That is, as far as expressiveness is concerned, 3VL is not needed for SQL, even to handle nulls.

To recap, our main contributions are as follows.

1. We give a precise semantics to the basic fragment of SQL (defined as \texttt{SELECT-FROM-WHERE} queries without aggregation) that accounts for its real-life behavior.
2. To justify it as the right semantics of SQL, we experimentally validate it with respect to a very large number of randomly generated queries to ensure that it always produces the same results as SQL implementations in commercial RDBMSs.
3. We provide a translation from basic SQL into RA and, using the formal semantics, prove its correctness.
4. We show that, contrary to the common belief, three-valued logic is not really necessary to model SQL behavior, even for handling nulls.

\textbf{Organization.} The data model and the syntax of the basic SQL fragment are defined in Section 2. The formal semantics is presented in Section 3. Experimental validation of the semantics is described in Section 4. Formal proof of the equivalence of basic SQL and RA is in Section 5. Section 6 shows how to eliminate three-valued logic from SQL. Concluding remarks are given in Section 7.

2. Basics SQL: Data Model and Syntax

In this section we describe the data model we use, and the syntax of the basic fragment of SQL’s query language.

2.1 Data model and syntax

As we all know, an SQL table is made up of rows, which may occur multiple times, and it is organized into columns, which have names attached to them. While this looks quite
straightforward, the way column names are handled in SQL is of paramount importance for providing the semantics of queries, which is our goal, and it requires a few clarifications.

- Can column names be repeated? For base tables stored in the database this is not allowed, but it is fairly easy to write SQL queries that produce tables with repeated column names. For example, if \( R \) is a base table with a column named \( A \), the query \( \text{SELECT} A, A \text{ FROM} R \) outputs a table with two columns, both named \( A \).

- What are column names exactly? If we only look at base tables or at the output of an SQL query, these are just attribute names. However, we also need to provide the semantics of subqueries, and each subquery appearing in the FROM clause is given a name. For example, in the query

\[
\text{SELECT} \quad R.A, \quad S.A \quad \text{FROM} \quad R, \quad (\text{SELECT} \quad A \quad \text{FROM} \quad R) \quad \text{AS} \quad S
\]

the base table \( R \) and the subquery in FROM must produce a table whose columns are named \( R.A \) and \( S.A \), which are pairs of names.

Thus, in general, column names in a table can repeat, and they can be either names or pairs of names. Towards capturing this, we assume the following two countable infinite sets:

- \( \mathbb{N} \) of names, which will serve as names of tables and their columns, and
- \( \mathbb{C} \) of data values that, along with \( \mathbb{N} \), will populate databases.

We refer to the elements of \( \mathbb{N} \) as \textit{names}, and to pairs of elements of \( \mathbb{N} \) (i.e., elements of \( \mathbb{N}^2 \)) as \textit{full names}, for which we will use the SQL-like notation \( N_1.N_2 \) rather than \( (N_1,N_2) \).

We can now define the data model. A \textit{record} is a tuple of elements of \( \mathbb{C} \cup \{\text{NULL}\} \), and a \textit{table} of arity \( k > 0 \) is a bag of records of length \( k \). A \textit{schema} is a set \( R \subseteq \mathbb{N} \) of (base) table names, where each \( R \) is a set associated with a non-empty tuple \( \ell(R) \) of distinct attribute names from \( \mathbb{N} \). A \textit{database} \( D \) maps each \( R \) to a (base) table \( R \ell(R) \) of arity \( |\ell(R)| \). We write \( R(A_1,\ldots,A_n) \) to indicate that \( \ell(R) = (A_1,\ldots,A_n) \).

### 2.2 Syntax of basic SQL

Our goal is to define the semantics of syntactically correct SQL queries, which have been successfully type-checked and compiled. Thus, w.l.o.g. we assume that queries are given in a form where all attribute names are fully annotated with the name of the table they come from. As an example, consider a schema with \( R(A) \) and \( T(A,B) \), and the query

\[
\text{SELECT} \quad A, \quad B \quad \text{AS} \quad C \quad \text{FROM} \quad R, \quad (\text{SELECT} \quad B \quad \text{FROM} \quad T) \quad \text{AS} \quad U \quad \text{WHERE} \quad A = B
\]

The fully annotated version of this query will be

\[
\text{SELECT} \quad R.A \quad \text{AS} \quad A, \quad U.B \quad \text{AS} \quad C \quad \text{FROM} \quad R \quad \text{AS} \quad R, \quad (\text{SELECT} \quad T.B \quad \text{AS} \quad B \quad \text{FROM} \quad T) \quad \text{AS} \quad U \quad \text{WHERE} \quad R.A = U.B
\]

In other words, each base table or subquery in FROM is given an explicit name, and its attributes are then qualified using that name; moreover, the names of the attributes that will appear in the output of the query are explicitly listed in the \textit{WHERE} clause. In fact, this closely resembles what happens when compiling SQL queries: RDBMSs add similar annotations to table and attribute names.

Another observation is that if a query compiled successfully, there are no type clashes, and thus we can assume that all comparisons and operations are applied to arguments of the right types. This explains why we assumed that there is just one set of data values that includes values of all types.

As already explained, in this paper we fully analyze the fragment that we call \textit{basic SQL}. This fragment includes:

- the usual \textit{SELECT-FROM-WHERE} queries;
- \textit{constants} and \textit{NULLs} in the \textit{SELECT} list, along with (fully qualified) attribute names;
- \textit{NULLs} handled according to SQL’s 3-valued logic;
- \textit{arbitrary user-specified conditions} on base tables;
- \textit{correlated subqueries} in \textit{WHERE} connected with \textit{EXISTS}, \textit{IN} and their negations;
- \textit{correlated subqueries} in \textit{FROM};
- \textit{set} and \textit{bag} semantics of queries;
- \textit{operations} of union, intersection, and difference (in both \textit{set} and \textit{bag} flavors); and
- \textit{arbitrary Boolean} combinations of conditions.

### Notations and conventions

A \textit{term} \( t \) is either a constant in \( \mathbb{C} \), or \textit{NULL}, or a full name in \( \mathbb{N}^2 \). We let \( \bar{t} \) stand for tuples of terms. We shall adopt the following conventions:

- \( \mathbb{N} \) ranges over names in \( \mathbb{N} \).
- \( A \) ranges over full names (elements of \( \mathbb{N}^2 \)).
- \( \alpha \) ranges over tuples of terms.
- \( \beta \) ranges over tuples of names.
- \( R \) ranges over names of base tables in a database.
- \( c \) ranges over constants (elements of \( \mathbb{C} \)).

References to tables are denoted by \( T \), which indicates either a query \( Q \) (whose output is indeed a table), or the name of a base table \( R \). We let \( \tau \) range over tuples of (references to) tables.

The syntax of basic SQL is given in Figure 1, where both \textit{queries} \( Q \) and \textit{conditions} \( \theta \) are defined by mutual recursion: queries have conditions in the \textit{WHERE} clause, and a condition may involve a query within \textit{EXISTS} or \textit{IN}.
Our goal now is to provide a formal semantics of queries from the SQL fragment defined in the previous section. Following the standard convention, we denote the semantics of a query \( Q \) by \([Q]\). This is a function that takes a database \( D \) as input and produces the output \([Q]_D\), which is the table obtained by executing \( Q \) on \( D \). The tuple of names assigned to the columns of \([Q]_D\) is denoted by \( \ell(Q) \), which is defined inductively on the structure of \( Q \) as shown in Figure 2 (concatenation of tuples is denoted by juxtaposition). For example, for \( Q = \text{SELECT } * \text{ FROM } R, S \) on a schema with \( R(A, B) \) and \( S(A, C) \), we have \( \ell(Q) = \ell(R) \ell(S) = (A, B, A, C) \).

As for \([Q]\), in general it is not enough to assume that the only input is the database \( D \), since we also need to provide the semantics of subqueries, which may take parameters. In conditions of the form \( \ell \in Q \), for example, the query \( Q \) can refer to full names in \( \ell \), whose values come from elsewhere. The standard way to account for this in programming semantics [17, 26] is to define an environment \( \eta \) that provides values for such parameters. In our case, the parameters are full names, so \( \eta \) is a partial map from \( N^2 \) to values, that provides the binding for each pair of table name and attribute name (e.g., \( S, B \)) on which it is defined.

This suggests that the function we need to define is \([Q]_{D, \eta}\) that takes a database \( D \) and the bindings of the environment \( \eta \) and produces the output of \( Q \). Then, for a query without parameters, we are looking at \([Q]_D = [Q]_{D, \emptyset}\).

This is almost true, but there is one more Boolean input that needs to be added. The problem with the definitions of the SQL Standard is that the semantics of queries is not compositional: that is, semantically a query can behave differently depending on the context in which it occurs. This is true of queries of the form \( \text{SELECT } * \). Normally \(*\) means that all attributes have to be returned, but if such a query occurs under \( \text{EXISTS} \), then \(*\) is equivalent to having any constant \( c \) in its place. This could lead to different behaviors. For example, given a base table \( R \) with attribute \( A \), the query \( Q = \text{SELECT } * \text{ FROM } (\text{SELECT } R.A, R.A \text{ FROM } R) \) as \( \tau \) will fail due to the ambiguity of the reference to \( R.A \), but the query \( \text{SELECT } * \text{ FROM } R \text{ WHERE EXISTS } (Q) \) will work and output \( R \) whenever it is nonempty. Thus, the same query \( Q \) has different semantics depending on the context.

To take into account the two meanings of \(*\) in the \( \text{SELECT} \) clause of queries, we introduce an additional Boolean input to \([Q]\). If \( Q \) is the outermost query nested inside an \( \text{EXISTS} \) condition, this switch is set to 1, otherwise to 0. Then, when \( Q \) is of the form \( \text{SELECT } * \), value 1 indicates that \(*\) is to be replaced with an arbitrary constant, and 0 that it must be expanded into a list of full names (provided by the \( \text{FROM} \) clause, as we shall see shortly). Thus, our semantic function becomes \([Q]_{D, \eta, x}\) where \( x \) is the value of the Boolean switch; for the top-level query \( Q \), we then take \([Q]_D = [Q]_{D, 0, 0}\).

Before providing the formal semantics of SQL queries, we need to introduce a few notions related to names and their bindings, and define operations on relations.

### Scopes and bindings

Each full name \( MN \) mentioned in the \( \text{SELECT} \) or \( \text{WHERE} \) clause of queries is a reference to some attribute \( N \) in some table \( M \). How are references resolved? Each \( \text{SELECT-FROM-WHERE} \) block defines a scope, and scopes are nested according to the structure of the query. Then, for each reference \( MN \), we first look for a match (i.e., a table \( M \) with an attribute \( N \)) in the \( \text{FROM} \) clause of the local scope where the reference occurs; if a match is not found (which is the case of parameters), we look at the \( \text{FROM} \) clause of the innermost scope in which the current one is nested, and so on until a match is found (or the query does not compile).

To model the notion of scope, we first define the operation \( N(A_1, \ldots, A_n) \) that prefixes each name \( N_i \) with \( N \), yielding the tuple of full names \( (N, N_1, \ldots, N, N_m) \). For \( \tau = (T_1, \ldots, T_k) \) and \( \beta = (N_1, \ldots, N_k) \), we then let

\[
\ell(\tau : \beta) = N_1 \ell(T_1) \cdots N_k \ell(T_k)
\]

where again juxtaposition means concatenation of tuples.

We now formalize how the full names in a scope are bound to the values of a record in order to provide an environment. Given a tuple of full names \( \tilde{A} = (A_1, \ldots, A_m) \) and a record \( \tilde{r} = (a_1, \ldots, a_m) \) of the same length, we define the environment \( \eta_{\tilde{A}, \tilde{r}} \) that maps each non-repeated element \( A_i \) of \( \tilde{A} \) to the corresponding value \( a_i \) of \( \tilde{r} \); if \( A_i \) occurs more than once in \( \tilde{A} \), then \( \eta_{\tilde{A}, \tilde{r}} \) is not defined on it (a reference to a repeated full name is ambiguous).

The following definitions formalize how an environment is updated w.r.t. a scope and revised with new bindings. Given an environment \( \eta \) and a tuple of full names \( A \), we denote by \( \eta \uparrow A \) the environment obtained by removing from \( \eta \) the bindings for all elements of \( A \). That is, \( \eta \uparrow A \) is undefined on every \( A \in \tilde{A} \), and it is otherwise identical to \( \eta \). We define the environment \( \eta_{\tilde{A}, \tilde{r}} \) that maps each non-repeated element \( A_i \) of \( \tilde{A} \) to the corresponding value \( a_i \) of \( \tilde{r} \): if \( A_i \) occurs more than once in \( \tilde{A} \), then \( \eta_{\tilde{A}, \tilde{r}} \) is not defined on it (a reference to a repeated full name is ambiguous).

Operations on tables

To describe the semantics of SQL queries, we will use some of the standard operations on bags

---

1. This is the behavior prescribed by the Standard; not all RDBMSs follow it, see Section 4.
In addition, we use \([3, 15, 22]\). We denote by Figure 3: Semantics of SQL terms and truth values.

\[
[\mathcal{T}]_\eta = \begin{cases} 
\eta(A) & \text{if } t = A \\
\eta(c) & \text{if } t = c \in C \\
\text{NULL} & \text{if } t = \text{NULL}
\end{cases}
\]

\([t_1, \ldots, t_n]_\eta = ([t_1]_\eta, \ldots, [t_n]_\eta)\)

**Explanation of the semantics**

We now explain the key elements of the semantics, presented in Figures 3–6. The semantic function \([\cdot]_\eta\) takes different inputs depending on the syntactic construct under consideration: for queries \(Q\) the inputs are a database \(D\), an environment \(\eta\) and a Boolean variable \(x\) whose value is either 0 or 1; for conditions \(\theta\), the inputs are just the database and the environment; for terms \(t\), the only input is the environment.

**Terms** (see Figure 3) The semantics of a term is given by the environment \(\eta\): if a term \(t\) is a constant or null, it denotes itself; if it is a full name \(A\), then it denotes \(\eta(A)\). The semantics of a tuple \(t\) of terms is simply the tuple of values obtained by interpreting each term in \(t\).

**Queries** (see Figure 4) A base table \(R\) obviously denotes its interpretation in the database, i.e., \(R^\eta\). The evaluation of a \texttt{SELECT-FROM-WHERE} block starts by computing the Cartesian product of the tables produced by the elements of \(\tau\), each of which is either a base table or the output of a subquery. When the \texttt{WHERE} \(\theta\) clause is added, the tuples satisfying \(\theta\) are selected from the product. Observe that in this case the environment changes: when the condition \(\theta\) is evaluated for a record in the Cartesian product, the environment must be revised with the bindings for that record, because the scope of the local \texttt{FROM} clause has precedence over the outer scopes. For each record in the product that satisfies \(\theta\), the revised environment is then applied to the \texttt{SELECT} list \(\alpha\), which may also contain parameters, to produce the final output.

As discussed before, if the \texttt{SELECT} list is “\*”, the behavior depends on the context in which the query block occurs; this is determined by the value of the Boolean switch \(x\), which is set to 1 only for queries nested in an \texttt{EXISTS} condition.

**Conditions** (see Figure 5) As already mentioned, SQL operates with three truth values: true \(t\), false \(f\), and unknown \(u\). The semantics of a condition is one of these truth values. The expressions \texttt{TRUE} and \texttt{FALSE} denote \(t\) and \(f\) respectively. For a \(k\)-ary predicate \(P\), defined on non-null values, the semantics is \(u\) if one of the arguments is \texttt{NULL}. For equality, which is always assumed to be among the available predicates, we have that \([t_1 = t_2]_\eta\) is \(u\) if one of \([t_1]_\eta\) or \([t_2]_\eta\) is \texttt{NULL}; if both are elements \(c_1, c_2 \in C\), then the semantics is simply the result of the comparison \(c_1 = c_2\).

The condition \(\bar{i} \in Q\) is the disjunction of all the equalities \(\bar{i} = \bar{s}\) for every \(\bar{s}\) in the output of \(Q\), while \texttt{EXISTS} \(Q\) tests for non-emptiness. Note that, among conditions, only the basic predicates \(P \in \Pi\) and \(i \in Q\) can produce the truth value \(u\); this is then propagated through the connectives \&, \lor and \neg following the truth tables of SQL’s 3VL, which corresponds to what is known as the Kleene logic (see [5]).

**Operations** (see Figure 6) \texttt{UNION ALL}, \texttt{INTERSECT ALL}, and \texttt{EXCEPT ALL} are the bag operations \(\cup, \cap, \neg\) and \(-\) we described before. Without the keyword \texttt{ALL}, their set-theoretic version is used (for difference, duplicate elimination is applied first).

**Examples** It is easy to follow the rules of the semantics to see that queries \(Q_1\rightarrow Q_2\) from the introduction produce exactly the same results as they should, namely \(\emptyset\), \(\{1\}\) and \(\{1\}\) on a database with \(R = \{1\}\) and \(S = \{\texttt{NULL}\}\) as the condition \(\texttt{EXISTS}\) resolves. The second one, as it occurs under \texttt{EXISTS}, will be allowed, because * will be replaced by an arbitrary constant and no such ambiguity will occur.

These observations confirm the correctness of the semantics on the small number of examples from the introduction; in the next section we shall use many more examples of queries for validating the semantics.

4. EXPERIMENTAL VALIDATION OF SQL SEMANTICS

Now that we have given a formal semantics of basic SQL queries, how can we be sure that it is correct? The Standard is written in natural language; this was the motivation to provide a proper formal specification for the language in the first place. But what does it even mean that the semantics is correct? Intuitively, the correctness of the semantics should entail that it produces the same results as real RDBMSs do. Of course, proving such a statement formally is infeasible, which leaves open one route: experimental validation.

Thus, our plan is to experimentally confirm, with a sufficiently high degree of confidence, that the formal semantics from Section 2 is the right one, i.e., agrees with a very large number of randomly generated SQL queries, on random relational databases. There is one obstacle though, already discussed in the introduction. We formalized the description of the Standard, but all RDBMSs deviate from the Standard, typically in small but nonetheless significant ways [4, 21]. These necessitate adjusting the semantics we presented to account for the small differences real systems have with the Standard.

To give some concrete example, PostgreSQL has chosen to use compositional semantics of queries: that is, \texttt{SELECT} * behaves in the same way regardless of the context in which the query is used. This means that the extra Boolean switch is
\[ \lfloor R \rfloor_{D,\eta,x} = R^D \]
\[ \lfloor \tau: \beta \rfloor_{D,\eta,x} = \lfloor T_1 \rfloor_{D,\eta,0} \times \cdots \times \lfloor T_k \rfloor_{D,\eta,0} \quad \text{for } \tau = (T_1, \ldots, T_k) \]
\[
\begin{aligned}
\text{FROM } & \tau: \beta \\
\text{WHERE } & \theta \\
\end{aligned}
\]
\[
\begin{aligned}
\text{SELECT DISTINCT } & \alpha: \beta' \\
\text{FROM } & \tau: \beta \\
\text{WHERE } & \theta \\
\end{aligned}
\]
\[
\begin{aligned}
\text{SELECT } & * \\
\text{FROM } & \tau: \beta \\
\text{WHERE } & \theta \\
\end{aligned}
\]
\[
\begin{aligned}
\text{SELECT } & * \\
\text{FROM } & \tau: \beta \\
\text{WHERE } & \theta \\
\end{aligned}
\]
\[
\begin{aligned}
\text{SELECT DISTINCT } & \alpha: \beta' | * \\
\text{FROM } & \tau: \beta \\
\text{WHERE } & \theta \\
\end{aligned}
\]

**Figure 4:** Semantics of basic SQL: Queries.

\[
\lfloor P(t_1, \ldots, t_k) \rfloor_{D,\eta} = \begin{cases} 
  t & \text{if } P([t_1]_\eta, \ldots, [t_k]_\eta) \text{ holds and } [t_i]_\eta \neq \text{NULL} \text{ for all } i \in \{1, \ldots, k\} \\
  f & \text{if } P([t_1]_\eta, \ldots, [t_k]_\eta) \text{ does not hold and } [t_i]_\eta \neq \text{NULL} \text{ for all } i \in \{1, \ldots, k\} \\
  u & \text{if } [t_i]_\eta = \text{NULL} \text{ for some } i \in \{1, \ldots, k\} 
\end{cases}
\]
\[
\lfloor t \text{ IS NULL} \rfloor_{D,\eta} = \begin{cases} 
  t & \text{if } [t]_\eta = \text{NULL} \\
  f & \text{if } [t]_\eta \neq \text{NULL} 
\end{cases}
\]
\[
\lfloor t \text{ IS NOT NULL} \rfloor_{D,\eta} = \neg \lfloor t \text{ IS NULL} \rfloor_{D,\eta}
\]
\[
\lfloor (t_1, \ldots, t_n) = (t'_1, \ldots, t'_n) \rfloor_{D,\eta} = \bigwedge_{i=1}^n [t_i = t'_i]_{D,\eta} 
\]
\[
\lfloor \exists i \in Q \rfloor_{D,\eta} = \begin{cases} 
  t & \text{if } \exists r \in [Q]_{D,\eta,0} \text{ s.t. } [r]_{D,\eta} = t \\
  f & \text{if } \forall r \in [Q]_{D,\eta,0} \text{ s.t. } [r]_{D,\eta} = f \\
  u & \text{if } \exists r \in [Q]_{D,\eta,0} \text{ s.t. } [r]_{D,\eta} = t \text{ and } \exists r \in [Q]_{D,\eta,0} \text{ s.t. } [r]_{D,\eta} = f 
\end{cases}
\]
\[
\lfloor \neg \exists i \in Q \rfloor_{D,\eta} = \neg \lfloor \exists i \in Q \rfloor_{D,\eta}
\]
\[
\lfloor \text{EXISTS } Q \rfloor_{D,\eta} = \begin{cases} 
  t & \text{if } [Q]_{D,\eta,1} \neq \emptyset \\
  f & \text{if } [Q]_{D,\eta,1} = \emptyset 
\end{cases}
\]
\[
\lfloor \text{TRUE} \rfloor_{D,\eta} = t 
\]
\[
\lfloor \text{FALSE} \rfloor_{D,\eta} = f 
\]

**Truth Tables:**

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>t</th>
<th>f</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>t</td>
<td>t</td>
<td>u</td>
<td>u</td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td>t</td>
<td>f</td>
<td>u</td>
<td>u</td>
</tr>
</tbody>
</table>

**Figure 5:** Semantics of basic SQL: Conditions.
no longer needed and we just need to provide the semantics \([Q]_{D,\eta}\). The rule for \texttt{SELECT} * then simply changes to
\[
\begin{bmatrix}
\text{SELECT} \\
\text{FROM} \\
\text{WHERE}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{SELECT} \\
\text{FROM} \\
\text{WHERE}
\end{bmatrix}_{D,\eta}
\]

Other systems slightly change the syntax; for example Oracle uses \texttt{MINUS} instead of \texttt{EXCEPT}, while MySQL does not have it altogether. Such syntactic modifications are easy to account for.

Thus, to experimentally validate the semantics, we need to provide minor adjustments so that it would capture precisely what a concrete system implements. Under this understanding, we need to describe the following three components:

1. the correctness criterion;
2. the query generator for experiments;
3. implementation of the formal semantics; and
4. results of the experiments.

Correctness criterion Once we implement the semantics, we shall validate it w.r.t. a large number of randomly generated SQL queries, on random relational databases. By validating we mean that the semantics coincides with the result of executing the same query on an RDBMS. By “coincide” we mean that the table obtained from our implementation of the semantics and the table obtained as output from the DBMS have precisely the same number of columns, with the same names and in the same order, and that they have precisely the same rows (with the same multiplicities) although their order is arbitrary.

Query generator There are well-defined database benchmarks, like TPC-H [31], but they are designed for analyzing database performance. Benchmarks use commonly occurring queries (e.g., business support queries in TPC-H), but they have relatively few of them (22 for TPC-H). In order to validate the semantics, we need to compare it with the output of DBMSs on a significantly larger number of queries. While this precludes the use of standard benchmarks, we can still look at them to analyze the structure and features of their queries, and use those to generate a large number of queries that look somewhat like those found in benchmarks.

Towards that goal, we look at characteristics of the TPC-H benchmark. There are a total of eight base tables, but on average each benchmark query uses only 3.2 and all queries but one use 6 or fewer. Each query uses relatively few \texttt{WHERE} conditions per block, in fact only three queries use more than 8 conditions, and no query exceeds three levels of nesting.

We implemented a random query generator, which takes as input a schema, a set of names that can be used as aliases for attributes and tables, and the following parameters:

- \texttt{tables} = max number of tables (counting repetitions) mentioned in a well-defined \texttt{SELECT-FROM-WHERE} block, including nested subqueries;
- \texttt{nest} = max level of nested queries in \texttt{FROM} and \texttt{WHERE};
- \texttt{attr} = max number of attributes in a \texttt{SELECT} clause;
- \texttt{cond} = max number of atomic conditions in \texttt{WHERE}.

Based on the above observations from TPC-H, we chose the values \texttt{table} = 6, \texttt{nest} = 3, \texttt{attr} = 3, \texttt{cond} = 8.

Implementation of the semantics We implemented the semantics of Figures 3–6 in Python. Note that we only need this implementation to verify correctness against RDBMSs, and not for its performance. In fact, we have two slightly different implementations: one that accounts for PostgreSQL’s compositional semantics, and one for Oracle’s syntax.

Experimental results We used a fixed schema with base tables \(R_1, \ldots, R_8\), where each \(R_i\) consists of \(i+1\) attributes. Since the data type of values is immaterial to our semantics, to avoid type checking and therefore simplify query generation, all attributes in the schema are of type \texttt{int}. Using the query generator described earlier, we generated 100,000 random queries over this schema, and for each of them we generated a corresponding database instance using the random data generator Datafiller [11]. As we are not assessing performance here, the size of database instances is of secondary importance; hence, to speed up our implementation of the semantics (which computes Cartesian products) we capped the size of each generated base table to 50 rows.

For each query and associated database, we compared the output of PostgreSQL and Oracle with the output produced by our implementation of each variant of the semantics. The results were always the same. In particular, for some queries involving \texttt{SELECT} - Oracle raised an error due to presence of ambiguous references; in each of these cases, our implementation (the variant adjusted for Oracle) also raised an error, due to the environment being undefined on such ambiguous references, as expected. Of course, these situations did not arise for PostgreSQL.

This gives us good evidence to state that the semantics of Figures 3–6 is correct.

To sum up, our experiments validate the semantics of Section 2, and allow us to proceed to use this semantics in two applications that formally prove results about real-life SQL.

5. APPLICATION: EQUIVALENCE OF SQL AND ALGEBRA

It is a fundamental result of relational database theory that the expressiveness of the basic declarative query language, relational calculus, is the same as that of the basic

\[
\begin{align*}
[Q_1, \text{UNION ALL } Q_2]_{D,\eta,x} &= [Q_1]_{D,\eta,0} \cup [Q_2]_{D,\eta,0} \\
[Q_1, \text{INTERSECT ALL } Q_2]_{D,\eta,x} &= [Q_1]_{D,\eta,0} \cap [Q_2]_{D,\eta,0} \\
[Q_1, \text{EXCEPT ALL } Q_2]_{D,\eta,x} &= [Q_1]_{D,\eta,0} - [Q_2]_{D,\eta,0} \\
[Q_1, \text{UNION } Q_2]_{D,\eta,x} &= \varepsilon([Q_1, \text{UNION ALL } Q_2]_{D,\eta,x}) \\
[Q_1, \text{INTERSECT } Q_2]_{D,\eta,x} &= \varepsilon([Q_1, \text{INTERSECT ALL } Q_2]_{D,\eta,x}) \\
[Q_1, \text{EXCEPT } Q_2]_{D,\eta,x} &= \varepsilon([Q_1]_{D,\eta,0}) - [Q_2]_{D,\eta,0}
\end{align*}
\]
procedural language, relational algebra (RA). First shown by Codd in 1971 [10], it now belongs to all standard database texts. Relational DBMSs do not use relational calculus though; rather, they speak SQL. Of course SQL has many features that go beyond RA. Indeed, their data models differ slightly (for RA, it is assumed that attributes cannot repeat [2, 30], and RA queries simply manipulate data that already exists in the database, without the ability to create new data elements. But we shall show that once restricted to the same data model without repeating attributes, the power of data manipulating queries in basic SQL is the same as RA under bag semantics.

Bag Relational Algebra: Syntax The data model for RA is very similar to the one we used for basic SQL: tables are multisets of records, and have attribute names which are just elements of N. That is, lists of attributes are tuples we referred to as β in Section 2. Crucially, column names cannot repeat within a table, which is the standard assumption for RA, and we follow it here.

Relational algebra expressions are given by the grammar:

\[ E ::= R \mid \pi_{\beta}(E) \mid \sigma_{\theta}(E) \mid E \times E \mid E \cup E \mid E \cap E \mid E - E \mid \rho_{\beta \to \beta'}(E) \mid \varepsilon(E) \]

In relational algebra, terms are given by \( t ::= N \mid c \mid \text{NULL} \) where \( N \) now ranges over \( N \), and \( c \) ranges over \( C \) as before. Note that \( \beta \) and \( \beta' \) in the grammar above are tuples of terms, in particular tuples of names.

For a given collection of predicates \( P \in \mathcal{P} \), the conditions \( \theta \) in selections are given by

\[ \theta ::= \text{TRUE} \mid \text{FALSE} \mid P(t) \mid \text{const}(t) \mid \text{null}(t) \mid \theta \land \theta \mid \theta \lor \theta \mid \neg \theta \]

As in the case of SQL, we assume that \( \mathcal{P} \) contains at least the equality predicate, and predicates are interpreted under three-valued logic. The predicate \( \text{null}(t) \) tests if the value of a term is null, and \( \text{const}(t) \) is the negation of \( \text{null}(t) \).
We define the signature $\ell(E)$ of an expression, i.e., the list of attribute names of the table that $E$ generates, as follows:

$$\ell(R) = \text{list of attributes of base relation } R$$

$$\ell(E_1 \times E_2) = \ell(E_1) \times \ell(E_2)$$

$$\ell(E_1 \text{ op } E_2) = \ell(E_1)$$

$$\ell(\pi_{\beta}(E)) = \beta$$

$$\ell(\alpha(E)) = \ell(E)$$

$$\ell(\rho_{\beta \rightarrow \beta'}(E)) = \beta'$$

The expression $E_1 \times E_2$ is well-defined only if $\ell(E_1)$ and $\ell(E_2)$ are disjoint; the expression $E_1 \text{ op } E_2$ for $\text{op} \in \{\cup, \cap, -\}$ is well-defined only if $\ell(E_1) = \ell(E_2)$; the expression $\pi_{\beta}(E)$ is well-defined only if $\beta$ consists of elements of $\ell(E)$ and does not have repetitions; the expression $\rho_{\beta \rightarrow \beta'}(E)$ is well-defined only if $\beta = \ell(E)$, and $\beta'$ has the same length as $\beta$ and does not have any repetitions.

**Bag Relational Algebra: Semantics**

The semantics of well-defined relational algebra expressions is given in Figure 7. An expression $E$, evaluated on a database $D$, produces the table $[E]_D$, whose column names are given by $\ell(E)$. The operations of union, intersection, difference, Cartesian product, and duplicate elimination have their bag interpretation given in Section 3.

The environment $\eta$ is a partial mapping from $N$ to values (constants or null). It is only needed for evaluating selection conditions, and projections. For $\beta = (N_1, \ldots, N_m)$ and $\bar{a} = (a_1, \ldots, a_m)$, the environment $\eta^{\beta}_{\bar{a}}$ is defined so that $\eta^{\beta}_{\bar{a}}(N_i) = a_i$ for $i \in \{1, \ldots, m\}$. Since no repetitions in $\beta$ are allowed, $\eta^{\beta}_{\bar{a}}$ is always well defined.

Note that the semantics of projection is the standard one under 3VL; for example, for a base table $J_{E}$, the expression $\pi_{\beta}(E)$ is well-defined only if $\beta = \ell(E)$, and $\beta'$ has the same length as $\beta$ and does not have any repetitions.

**SQL and RA: equivalence**

As already explained, equivalence cannot be guaranteed without imposing any restrictions on SQL queries, simply because RA queries just manipulate data that is available in the database and do not invent new values, nor repeat attributes. But this is the only restriction we need to impose.

**Definition 1.** A basic SQL query is a data manipulation query if the query itself and every subquery in it is of the form $\text{SELECT [DISTINCT] } \alpha : \beta'$ FROM $\beta$ WHERE $\theta$ so that the names in $\beta'$ do not repeat, and for each $A = N_1, N_2 \in \alpha$, the name $N_1$ occurs in $\beta$.

Thus, we simply disallow repetition of column names in query/subquery results, force attributes in * to be listed explicitly, and only allow data from relations/subqueries in the FROM clause to appear in SELECT. Note that we do not forbid duplication of columns per se: for example, one can write $\text{SELECT } \text{R.A AS } A_1, \text{R.A AS } A_2 \text{ FROM } \text{R}$; we only require that in the output the columns be named differently.

**Theorem 1.** Data manipulation queries of basic SQL and relational algebra under bag semantics have the same expressive power.

We now explain the proof. It is more direct, and covers more cases than translations of [6, 32]. The translation of algebra into SQL is completely standard. For the converse, we first introduce a new version of relational algebra called SQL-RA. It adds selection conditions that mimic nested IN and EXISTS subqueries, which makes the translation from SQL easy. We then show that these extra conditions are just syntactic sugar, and SQL-RA is equivalent to RA.

The extension to SQL-RA makes the definition of expressions and conditions mutually recursive by extending conditions as follows:

$$\theta := \text{true} | \text{false} | P(t) | \text{const}(t) | \text{null}(t)$$

$$| \theta \land \theta | \theta \lor \theta | \neg \theta | t \in E | \text{empty}(E)$$

These additions are direct analogs of SQL’s IN and EXISTS subqueries.

To extend the semantics, now every expression, not just conditions, must carry an environment $\eta$, which only changes in one case:

$$[\sigma_{\theta}(E)]_{D,\eta} = \left\{ \bar{a} \in \left\{ \bar{a}_k \right\}_D \text{ and } \left[ \theta \right]_{D,\eta;\bar{a}_k(E)} = t \right\}$$

The semantics of the additional conditions is as follows:

$$[\text{empty}(Q)]_{D,\eta} = \begin{cases} t & \text{if } Q \in [E]_D: \left[ \text{false} \right]_{D,\eta} = t \\ f & \text{if } \forall Q \in [E]_D: \left[ \text{false} \right]_{D,\eta} = f \\ u & \text{otherwise} \end{cases}$$

where, for $\bar{r} = (r_1, \ldots, r_m)$ and $\bar{s} = (s_1, \ldots, s_m)$, we define $\left[ \text{false} \right]_{D,\eta} = \left\{ \left[ r_i \right]_{D,\eta} = \left[ s_i \right]_{D,\eta} \mid 1 \leq i \leq m \right\}$.

Some expressions of SQL-RA have parameters: for instance, expressions $E$ in $t \in E$ can refer to values in $t$. When we say that an SQL-RA expression is equivalent to an SQL query or an RA expression, we mean expressions with no parameters, evaluated under the empty environment. The set of parameters of an expression $E$, denoted by $\param(E)$, and the set of parameters of a condition $\theta$ with respect to a set of attribute names $A$, denoted by $\param(\theta, A)$, are defined by mutual recursion as follows. Below $\op$ stands for binary operations $\times, \cup, \cap, -\}$; $\conn$ for connectives $\land, \lor$, and $\name$ applied to a set or tuple of terms returns those terms that are names (as opposed to constants and nulls).

$$\param(R) = \emptyset$$

$$\param(E_1 \op E_2) = \param(E_1) \cup \param(E_2)$$

$$\param(E) = \param(\pi_{\alpha}(E)) = \param(E)$$

$$\param(\pi_{\alpha}(E)) = \param(\pi_{\alpha}(\theta, \{A \mid A \in \ell(E)\}))$$

$$\param(P(t_1, \ldots, t_k, A) = \name \{t_1, \ldots, t_k\}) = A$$

$$\param(\theta; \conn \theta_2, A) = \param(\theta_1, A) \cup \param(\theta_2, A)$$

$$\param(\neg \theta, A) = \param(\theta, A) \cup \param(\text{false}) = A$$

$$\param(\text{empty}(E), A) = \param(E) = A$$

Then SQL-RA queries are defined as SQL-RA expressions $E$ with $\param(E) = \emptyset$. The semantics of an SQL-RA query $E$ with respect to a database $D$ is given by $[E]_{D,\emptyset}$.

**Proposition 1.** For every data manipulation query in basic SQL, there is an equivalent SQL-RA query.
The translation follows the structure of queries closely once we have resolved two mismatches. The first is about the use of full names (i.e., elements of $\mathbb{N}^2$) in basic SQL vs the use of names from $\mathbb{N}$ in SQL-RA. The second is about SQL projection vs SQL-RA projection.

To address the former, since there are infinitely many names, we can simulate all full names used in a query with extra names from $\mathbb{N}$. For a given SQL query $Q$, we define an injective mapping $\chi: \mathbb{N}^2 \to \mathbb{N} - (\mathbb{NQ} \cup \mathbb{N}_{\text{base}})$, where $\mathbb{NQ}$ is the set of all names occurring in the rename list of each SELECT clause in $Q$, and $\mathbb{N}_{\text{base}}$ is the set of all column names of each base table in the schema. Given this correspondence, one can then simulate prefixing by renaming: for a tuple of distinct names $N_1, \ldots, N_m$ in $\mathbb{N}$ and a name $N \in \mathbb{N}$, we let:

$$\rho^\chi_N(N_1, \ldots, N_m) = (\chi(N, N_1), \ldots, \chi(N, N_m))$$

which is extended to RA expressions $E$ as follows: $\rho^\chi_N(E) = \rho_{\beta \to \beta'}(E)$ with $\beta = \ell(E)$ and $\beta' = \rho^\chi_N(\beta)$.

Given a name mapping $\chi$ and an SQL environment $\eta: \mathbb{N}^2 \to \mathbb{C} \cup \{\text{\sc null}\}$, the SQL-RA environment corresponding to $\eta$ w.r.t. $\chi$, denoted by $\eta^\chi$, is the partial mapping $\eta \circ \chi: \mathbb{N} \to \mathbb{C} \cup \{\text{\sc null}\}$, where $\chi$ is the left inverse of $\eta$.

When it comes to projection, SQL’s SELECT $\alpha$ may have repetitions of attributes, in which case we cannot use it directly in SQL-RA. But it always comes together with a renaming $\beta$ which, for data manipulation queries, contains no repetitions. This makes it possible to achieve duplication of columns in RA as well, as we show next. For this, we need to introduce a definition that will also be important in the next section.

**Definition 2.** The *syntactic equality* of terms, denoted by $[t_1 \equiv t_2]_{D, \eta}$, is the comparison with the following semantics:

$$[t_1 \equiv t_2]_{D, \eta} = \begin{cases} t & \text{if } [t_1]_{\eta} = [t_2]_{\eta} \\ f & \text{if } [t_1]_{\eta} \neq [t_2]_{\eta} \end{cases}$$

In other words, two terms are syntactically equal if they refer to the same constant or NULL. Syntactic equality does not add expressive power, because $t_1 \equiv t_2$ is equivalent to $(t_1 = t_2 \land \text{const}(t_1) \land \text{const}(t_2)) \lor (\text{null}(t_1) \land \text{null}(t_2))$.

If $E$ is an RA expression whose signature $\ell(E)$ consists of distinct names, $\alpha = (\alpha_1, \ldots, \alpha_n)$ is a tuple of names from $\ell(E)$, and $\beta = (\beta_1, \ldots, \beta_n)$ is a list of distinct names that do not appear in $\ell(E)$, then we define $\pi^\beta_\alpha(E)$ as

$$\left\{ \begin{array}{ll} \rho^\alpha_{\to \beta}(\pi^\alpha(\ell(E))), & \text{if } \alpha \text{ has no repetitions}, \\ \pi^\beta_\alpha(\pi^\alpha(\ell(E)) \cup^{\mathbb{N}^*} (\text{null}^{\mathbb{N}^*}(\rho^\alpha_{\to \beta}(\ell(E))))), & \text{otherwise}. \end{array} \right.$$ 

where $\cup^{\mathbb{N}^*}$ is the syntactic natural join, i.e., natural join where the comparison condition on common attributes is syntactic equality. Note that the projection operation is straightforward if there are no repetitions of attributes; if repetitions exist, one can only simulate them in RA using additional joins, which is captured by the above definition.

Using these, translations from basic SQL to SQL-RA, under renaming $\chi$, are defined in Figure 8. The proof of correctness proceeds by induction on the structure of queries and conditions.

We then need the second component of the proof of the equivalence, namely that the new conditions are syntactic sugar and can be eliminated.

**Proposition 2.** For every SQL-RA query, there is an equivalent RA query.

This can be shown in three steps. First, one can eliminate $\ell \in E$ conditions, replacing them with emptiness conditions instead. Then one translates the resulting expression into a special normal form where each condition is either a predicate $P(\ell)$, or empty($E$), or their negations. Finally one translates $\sigma_{\text{empty}}(E')$ and $\sigma_{\text{empty}}(E')$ into left (anti) semijoins of $E$ and $E'$.

This completes the proof of the equivalence. \hfill \square

**Example** We again return to the queries $Q_1$–$Q_3$ from the introduction, which provide three non-equivalent ways of expressing difference of relations $R$ and $S$ with one attribute $a$. The translations that account for the different behavior of these queries are as follows, where $R' = \rho_{\text{A} \to \text{B}}(R)$ and $S' = \rho_{\text{A} \to \text{C}}(S)$:

$Q_1 = \rho_{\text{B} \to \text{A}}(R' \semijoin_{\text{B} \to \text{C}}(R' \times S'))$

$Q_2 = \rho_{\text{B} \to \text{A}}(R' \semijoin_{\text{B} \to \text{C} \land \text{null}(\beta)}(R' \times S'))$

$Q_3 = R - S$.

The operation $\semijoin$ in the first two expressions above is the antijoin which is based on syntactic equality, i.e., $E_1 \semijoin_{\eta} E_2 = E_1 \setminus \pi_{\eta(E)}(E_1 \bowtie^\eta_{\eta (E)} E_2)$. Note that this is the standard interpretation of equality in implementations of antijoin plans in RDBMSs.

6. APPLICATION: DO WE NEED THREE-VALUED LOGIC?

It is commonly believed that to evaluate SQL queries it is necessary to use three-valued logic. So far the description of the semantics, which follows the Standard, and the equivalence to three-valued RA seem to confirm this. But if 3VL is really necessary, then what are the queries that we miss if we use the standard Boolean valued logic with only $t$ and $f$ truth values? The answer is, somewhat surprisingly: none. Despite what all the SQL books and database texts tell us about the need for three-valued logic to handle nulls, it turns out that the familiar two-valued logic suffices. The presence of a formal semantics of SQL allows us to provide a rigorous proof of this fact.

**Two-valued semantics of SQL** To define SQL semantics under two-valued logic, we need to analyze when the third truth value $\text{\sc unknown}$ ($\text{\sc u}$) appears. There is only base case, of predicates $P \in P$. This also includes equality $\equiv$, which we always assume to be present. Then $\text{\sc u}$ propagates further through Boolean connectives and conditions $\ell$ in $Q$, which amount to disjunctions of $\ell = \overline{\ell}$ for $\overline{\ell}$ in the result of $Q$.

To give the two-valued semantics to predicates $P \in \mathcal{P}$, other than equality, we can confute $t$ and $u$. Note that this is a rather natural decision in the context of SQL: after all, when conditions in WHERE are evaluated under 3VL, only tuples for which the conditions evaluate to $t$ are kept, and tuples for which it is $f$ or $u$ are discarded.

We thus modify the semantics of Figures 3–6 to obtain a two-valued semantics $\boxed{\equiv}$ by changing rules for predicates $P \in \mathcal{P}$, and for equality, which can be re-interpreted in two different ways.

For predicates $P$, we use

$$[P(t_1, \ldots, t_k)]^\equiv_{\eta} = \begin{cases} t & \text{if } P([t_1]_{\eta}^\equiv, \ldots, [t_k]_{\eta}^\equiv) \text{ holds and } [t_i]_{\eta}^\equiv \neq \text{\sc null}, 1 \leq i \leq k \\ f & \text{otherwise} \end{cases}$$
\( R \xrightarrow{\lambda} R \) if \( R \) is the name of a base relation

\[(T_1, \ldots, T_k) : (N_1, \ldots, N_k) \xrightarrow{\lambda} \rho^D_{\lambda_1}(E_1) \times \cdots \times \rho^D_{\lambda_k}(E_k) \) if \( T_i \xrightarrow{\lambda} E_i \), for \( 1 \leq i \leq k \)

\begin{align*}
\text{SELECT } \ [\text{DISTINCT}] & \; \alpha : \beta' \\
\text{FROM} & \; \theta \\
\text{WHERE} & \; \tau : \beta \xrightarrow{\lambda} \left\lfloor \varepsilon \right\rfloor \pi^\alpha_{\lambda'(\varepsilon)}(\sigma^\beta_{\theta}(E)) \quad \text{if } \tau : \beta \xrightarrow{\lambda} E \text{ and } \theta \xrightarrow{\lambda} \theta'
\end{align*}

\[
i(t) \xrightarrow{\lambda} \begin{cases} t & \text{ if } t \in C \cup \{ \text{NULL} \} \\ \chi(t) & \text{ otherwise} \end{cases}
\]

\[
b \xrightarrow{\lambda} b \quad \text{ if } b = \text{TRUE or FALSE}
\]

\[
t \text{ IS NOT NULL } \xrightarrow{\lambda} \lceil \neg \rceil \text{null}(t)
\]

\[
P(t_1, \ldots, t_n) \xrightarrow{\lambda} P(\hat{t}_1, \ldots, \hat{t}_n)
\]

\[
\text{EXISTS } Q \xrightarrow{\lambda} \neg \text{empty}(E)
\]

\[
\hat{t} \text{ NOT IN } Q \xrightarrow{\lambda} \lceil \neg \rceil (\hat{t}_1, \ldots, \hat{t}_n) \in E)
\]

For \( \theta_1 \xrightarrow{\lambda} \theta_1 ' \) and \( \theta_2 \xrightarrow{\lambda} \theta_2 ' \):
\[
\theta_1 \land \theta_2 \xrightarrow{\lambda} \theta_1 ' \land \theta_2 ' \quad , \quad \theta_1 \lor \theta_2 \xrightarrow{\lambda} \theta_1 ' \lor \theta_2 ' \quad , \quad \neg \theta_1 \xrightarrow{\lambda} \neg \theta_1 '
\]

For \( Q_1 \xrightarrow{\lambda} E_1 \) and \( Q_2 \xrightarrow{\lambda} E_2 \):
\[
Q_1 \text{ UNION ALL } Q_2 \xrightarrow{\lambda} E_1 \cup \rho_1(Q_2) \rightarrow \ell(Q_1)(E_2)
\]
\[
Q_1 \text{ INTERSECT ALL } Q_2 \xrightarrow{\lambda} E_1 \cap \rho_1(Q_2) \rightarrow \ell(Q_1)(E_2)
\]
\[
Q_1 \text{ EXCEPT ALL } Q_2 \xrightarrow{\lambda} E_1 \setminus \rho_1(Q_2) \rightarrow \ell(Q_1)(E_2)
\]

\begin{align*}
Q_1 \xrightarrow{\lambda} & \epsilon(E_1 \cup \rho_1(Q_2) \rightarrow \ell(Q_1)(E_2)) \\
Q_2 \xrightarrow{\lambda} & \epsilon(E_1 \cap \rho_1(Q_2) \rightarrow \ell(Q_1)(E_2)) \\
Q_2 \xrightarrow{\lambda} & \epsilon(E_1 \setminus \rho_1(Q_2) \rightarrow \ell(Q_1)(E_2))
\end{align*}

\textbf{Figure 8: Translation } \xrightarrow{\lambda} \text{ from core SQL to SQL-RA, under renaming } \chi

For \( t_1 = t_2 \), we have two options. One is to use syntactic equality \( \equiv \) of Definition 2 whose semantics is already guaranteed to produce only \( t/\hat{t} \) truth values. The other is to use the above definition. To spell it out, if \( \lfloor t_1 \rfloor^n_\theta = c_1 \) and \( \lfloor t_2 \rfloor^n_\theta = c_2 \), then \( \lfloor t_1 = t_2 \rfloor^n_{D, \eta} \) is \( t \) if \( c_1 = c_2 \) and \( \hat{f} \) if \( c_1 \neq c_2 \). If at least one of \( \lfloor t_i \rfloor^n_\theta \) is NULL, then \( \lfloor t_1 = t_2 \rfloor^n_{D, \eta} = \hat{f} \).

It turns out that it does not matter which definition we use: in both cases, the two-valued semantics captures the behavior of SQL.

\textbf{Theorem 2.} Basic SQL queries have the same expressiveness under the three-valued and the two-valued semantics. That is, for every query \( Q \) there is a query \( Q' \) such that \( \lfloor Q \rfloor_D = \lfloor Q' \rfloor^n_D \), and conversely, for every query \( Q \) there is a query \( Q'' \) such that \( \lfloor Q'' \rfloor_D = \lfloor Q \rfloor^n_D \), for all databases \( D \). This is true for either interpretation of equality under two-valued semantics.

\textbf{Proof outline.} Since the two-valued semantics of any predicate \( P \in \mathcal{P} \) can be expressed in the three-valued semantics simply by taking conjunctions with conditions that its arguments are not nulls, and since syntactic equality can be expressed as well, translations from two-valued semantics to three-valued semantics are immediate.

We thus concentrate on producing, from a query \( Q \) (potentially with parameters), another query \( Q' \) so that \( \lfloor Q \rfloor_D, \eta = \lfloor Q' \rfloor^n_D, \eta \). This is done by defining three translations by mutual induction:

- from conditions \( \theta \) to \( \theta' \) and \( \theta'' \) such that
  \[
  \lfloor \theta \rfloor_D, \eta = t \Leftrightarrow \lfloor \theta' \rfloor^n_D, \eta = t \\
  \lfloor \theta \rfloor_D, \eta = f \Leftrightarrow \lfloor \theta'' \rfloor^n_D, \eta = t
  \]
- from queries \( Q \) to \( Q' \) by inductively replacing each condition \( \theta \) by \( \theta'' \).
  That is, conditions \( \theta' \) and \( \theta'' \) describe, under two-valued semantics, the behavior of \( \theta \) under three-valued semantics (note that checking whether \( \lfloor \theta \rfloor = u \) is captured by \( \neg \theta \land \neg \theta' \)), and \( Q' \) simply replaces \( \theta \) with \( \theta'' \), since we only select tuples for which the condition is true.

Translations of conditions are shown in Figure 9 for the case when the two-valued interpretation of equality is the same as for other predicates \( P \in \mathcal{P} \). We use the abbreviation \( t \text{ IS NOT NULL} \) for the conjunction of \( t \text{ IS NOT NULL} \) for all \( t \) in \( \hat{t} \). In the translation of \( t \text{ IN } Q \), the name \( N \) is assumed to be fresh, and \( N \text{ IS NOT NULL} \) is, as in SQL, the conjunction over all attributes \( A \) of the table named \( N \) of conditions \( A \text{ IS NOT NULL} \) (thus the condition in \( \text{WHERE} \) is actually the disjunction of conditions testing if an attribute is null).

When equality is interpreted as syntactic equality, we only need to add two rules:

\[
(t_1 = t_2)^t = (t_1 \equiv t_2) \quad \text{ AND } (t_1 = t_2)^f = \text{ NOT } (t_1 \equiv t_2) \quad \text{ AND } (t_1 = t_2) \text{ IS NOT NULL}
\]

Then, using the formal semantics of SQL, one verifies by induction on queries and conditions that \( \lfloor Q \rfloor_D, \eta = \lfloor Q' \rfloor^n_D, \eta \) for all \( D \) and \( \eta \).

\textbf{Why not use two-valued logic?} If SQL can be evaluated under two-valued semantics without losing expressiveness, do we really need three-valued logic for SQL evaluation, especially since it attracts so much criticism [12] ? We argue that at the moment, despite the established equivalence result, we are not yet ready to fully abandon SQL’s 3VL.
To start with, there is a huge amount of legacy code out there that assumes query evaluation under 3VL. Assume for a minute that SQL did switch to two-valued interpretation in all its queries. Then such legacy queries need to be rewritten to give the same results as they used to. Emulating old behavior turns into a case analysis, and leads to more cumbersome and less efficient queries. Moreover, case analysis — even simple forms of it such as expressing syntactic equality \( \equiv \) — introduces extra disjunctions in queries whenever negations occur. It is however well known and documented that commercial optimizers struggle with queries involving disjunctions [9].

Still it is tantalizing that from the point of view of expressiveness, one can eliminate the much maligned three-valued logic from the basic fragment of SQL.

7. CONCLUSION

We have produced a formal semantics of a basic fragment of SQL that behaves like the real-life SQL does, as opposed to its theoretical reconstructions with their many simplifications. We verified its behavior experimentally on a very large number of queries. Using this formal semantics, we provided two applications. We formally proved the equivalence of the basic fragment with relational algebra (something that had only been done in the past under significant simplifications that do not reflect the real behavior of the language). We also formally showed that three-valued logic is not required to achieve full expressiveness of this fragment of SQL, and somewhat surprisingly the familiar two-valued logic does the job as well.

Future work. Several questions arise from the formal semantics of SQL and its experimental validation. First, we would like to extend it to include more features of the language, especially aggregation and grouping, and extend correctness verification to them.

Some of the restricted SQL semantics [8, 23, 33] were defined for verifying correctness of SQL optimization rules. They could only do so under the restrictions imposed; thus it would be interesting to see what such verification techniques would yield without restrictions on the language.

Yet another line for future work that is brought by the availability of experimentally verified semantics is the extension of recent attempts to restore correctness of SQL query evaluation with nulls, cf. [16]. So far this has only been done for databases with marked nulls, due to the lack of formal semantics of query evaluation with SQL nulls. Now we have the tools to extend notions of certainty and possibility to handle SQL’s nulls.

8. REFERENCES