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 the main features of languages are abundant, but formal semantics of real languages that people use are few and far between.

We consider 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. This fragment already 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 the three-valued logic employed by SQL, which is universally assumed to be necessary to handle nulls. We prove however that this is not so, as three-valued logic does not add expressive power: every SQL query in our fragment can be evaluated under the usual two-valued Boolean semantics of conditions.

1. INTRODUCTION

Providing a formal semantics of a language is a major task in programming languages research [18, 20, 28]. It enables one to formally reason about languages, verify the correctness of programs, and it becomes a fundamental 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, with all of its idiosyncrasies, is a much harder task. This was done for several languages [1, 14, 19, 27, 31, 32]; the difference is that to describe such a formal semantics one needs a book, rather than a paper (or sometimes even a second book to explain what the first one said [26]).

When it comes to the main query language used by relational DBMSs – SQL – we have the Standard [21], but this cannot serve as a formal semantics on its own, because it is written in natural language, which is inherently ambiguous. In fact, it is well known that different vendors interpret various aspects of the Standard differently (see, e.g., [4, 22]). From a practical point of view, a natural language specification is harder to implement and maintain, and it does not lend itself to proper formal reasoning, which is necessary to derive language equivalences and optimization rules. The need for a formal semantics of SQL is witnessed by the fact that there have been several attempts at providing one in the past [7, 36, 29, 8, 9, 24, 37]. However, as we shall discuss in more detail in Section 7, all of the approaches found in the literature deviate significantly from the behavior of the real SQL language and the way it is used in practice. This is due to the fact that they make at least one of the following simplifying assumptions: set semantics (that is, rows in tables do not repeat) and absence of null values.

While theoretical work in databases typically views relations as sets of tuples, SQL tables are *bags* (a.k.a. multisets) where the same tuple can occur multiple times. Of course we can enforce set semantics in SQL by appropriately removing duplicates, but this is computationally expensive and therefore avoided in practice unless really necessary. But the reason for dealing with bags in real life scenarios is not limited to efficiency: the number of occurrences of tuples in tables reflects the actual data distribution, and preserving this information is crucial in applications where query answers are further processed to produce relevant data analytics.

As for nulls, we know that data in the real world is far from perfect: some values may be missing or unavailable due to faulty sensors, human or software errors, design choices, or genuine lack of information. Thus, we need to accept the presence of nulls as a fact of life, and if we want to capture the meaning of SQL queries in the real world we cannot just pretend that nulls do not exist.

It is well known that, for handling nulls, SQL uses three truth values [13]: true ($\mathbf{t}$), false ($\mathbf{f}$), and unknown ($\mathbf{u}$). With few exceptions, atomic comparisons where one of the arguments is SQL's null value `NULL` result in $\mathbf{u}$. Truth values are propagated through the logical connectives of conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$) using the truth

Proceedings of the VLDB Endowment, Vol. 10, No. 11

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

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

	$\neg$
t	f
f	t
u	u

Figure 1: Truth tables for SQL's 3VL.

tables of Kleene logic [6], which are given in Figure 1. As a matter of fact, SQL mixes this particular three-valued logic (3VL) with the usual Boolean logic: the **WHERE** clause is evaluated under 3VL, but then **f** and **u** are conflated, and only tuples for which the condition evaluates to **t** are returned. Furthermore, it does not apply the same treatment of nulls uniformly: in set operations like difference and intersection, two values (including **NULL**) are considered equal if they are *syntactically* the same.

Disregarding these aspects of the SQL language can lead to wrong equivalences among queries, as the following example illustrates.

Example 1. Consider two relations R and S with a single attribute A, and compute their difference $R-S$. One option is to use a **NOT IN** subquery:

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

By translating **NOT IN** into **NOT EXISTS**, as for example [7, 36] would suggest, we get

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

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

Q_3: `SELECT R.A FROM R EXCEPT SELECT S.A FROM S`

While these queries are equivalent on databases without nulls, they are not in the presence of nulls. Indeed, consider a database D where $R = \{1, \texttt{NULL}\}$ and $S = \{\texttt{NULL}\}$; then $Q_1(D) = \varnothing$, $Q_2(D) = \{1, \texttt{NULL}\}$ and $Q_3(D) = \{1\}$.

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

The main goal of this paper is to give a formal semantics of the core of SQL and to provide evidence that it correctly captures the real behavior of the language. Here we look at the basic fragment that includes **SELECT-FROM-WHERE** queries without aggregation but with correlated subqueries in both **FROM** and **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 it 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 *experimental*: one needs to implement it, and compare its behavior with that of a real RDBMS on a very large number of queries.

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.

Example 2. 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, e.g., by writing `SELECT R.A, R.A FROM R`, where we again refer to a relation R with attribute A. How about using such an expression as a subquery? Consider the following query:

```
SELECT * FROM (SELECT R.A, R.A FROM R) AS T
```

This will be accepted by PostgreSQL, but it will result in a compile-time error in some of the commercial RDBMSs. On the other hand, if the same query is used as a subquery in

```
SELECT * FROM R WHERE EXISTS
  ( SELECT * FROM (SELECT R.A, R.A FROM R) AS T )
```

then suddenly it is fine, even with RDBMSs where the subquery alone refused to compile. Thus, not only is $\star$ 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, concerning language equivalences and expressiveness. One is to provide, for the very first time, a formal proof that basic SQL can be captured by relational algebra (RA). As mentioned earlier, existing translations [7, 36] 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. It is commonly believed that 3VL is really necessary to model SQL's behavior, but using our formal semantics we show that this is not so: basic SQL queries have the same expressive 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 formal, rather than natural language, semantics to the basic fragment of SQL (consisting of 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 real 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 common belief, three-valued logic is not necessary to model SQL behavior, even for handling nulls.

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 and its experimental validation is described in Section 4. The formal proof of equivalence between basic SQL and RA is in Section 5. Section 6 shows how to eliminate three-valued logic from SQL. Related work is discussed in Section 7 and concluding remarks are given in Section 8.

2. 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.

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 `SELECT A,A 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

  ```
  SELECT R.A, S.A FROM R, (SELECT A FROM R) AS 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:

- N of names, which will serve as names of tables and their columns, and
- C of data values that, along with `NULL`, will populate databases.

We refer to the elements of N as *names*, and to pairs of elements of N (i.e., elements of N^2) as *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 *record* is a tuple of elements of $\mathsf{C} \cup \{\texttt{NULL}\}$, and a *table* of arity $k > 0$ is a bag of records of length k. A *schema* is a set $\mathsf{R} \subset \mathsf{N}$ of (base) table names, where each $R \in \mathsf{R}$ is associated with a non-empty tuple $\ell(R)$ of distinct attribute names from N. A *database* D maps each R to a (base) table R^D of arity $|\ell(R)|$. We write $R(A_1, \ldots, A_n)$ to indicate that $\ell(R) = (A_1, \ldots, A_n)$.

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

```
SELECT A, B AS C
FROM   R, (SELECT B FROM T) AS U
WHERE  A = B
```

The fully annotated version of this query will be

```
SELECT R.A AS A, U.B AS C
FROM   R AS R, (SELECT T.B AS B FROM T AS T) AS U
WHERE  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 `SELECT` 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 basic SQL. This fragment includes:

- the usual `SELECT-FROM-WHERE` queries;
- constants and `NULL`s in the `SELECT` list, along with (fully qualified) attribute names;
- `NULL`s handled according to SQL's 3-valued logic;
- correlated subqueries in `WHERE` connected with `EXISTS`, `IN` and their negations;
- correlated subqueries in `FROM`;
- set and bag semantics of queries;
- operations of union, intersection, and difference (in both set and bag flavors); and
- arbitrary Boolean combinations of conditions.

Notations and conventions. A *term* t is either a constant in C, or `NULL`, or a full name in N^2. We let $\bar{t}$ stand for tuples of terms. We shall adopt the following conventions:

N	ranges over names in N.
A	ranges over full names (elements of N^2).
α	ranges over tuples of terms.
β	ranges over tuples of names.
R	ranges over names of base tables in a database.
c	ranges over constants (elements of 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 τ range over tuples of (references to) tables.

The syntax of basic SQL is given in Figure 2, where both *queries* Q and *conditions* θ are defined by mutual recursion: queries have conditions in the `WHERE` clause, and a condition may involve a query within `EXISTS` or `IN`.

Observe that β and β' in queries provide explicit names for the tables in `FROM` and for the terms in `SELECT`, respectively.

The fragment we consider is parameterized by a collection $\mathcal{P}$ of predicates on base types. We assume that equality ($=$) of values is always available, for all types. Other operations can be type-specific, such as comparisons $<$ and $\leq$ for integers, or the lexicographic ordering and `LIKE` predicates for strings. All we assume is that there is a well-defined semantics of such predicates for non-null values of base types.

3. FORMAL SEMANTICS

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 3 (concatenation of tuples is denoted by juxtaposition). For example, for $Q =$ `SELECT * 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

$$\tau : \beta \;\coloneqq\; T_1 \;\texttt{AS}\; N_1,\; \ldots,\; T_k \;\texttt{AS}\; N_k \quad \text{for } \tau = (T_1, \ldots, T_k),\ \beta = (N_1, \ldots, N_k),\quad k > 0$$
$$\alpha : \beta' \;\coloneqq\; t_1 \;\texttt{AS}\; N'_1,\; \ldots,\; t_m \;\texttt{AS}\; N'_k \quad \text{for } \alpha = (t_1, \ldots, t_m),\ \beta' = (N'_1, \ldots, N'_m),\ m > 0$$

QUERIES

$$\begin{aligned} Q \;\coloneqq\;& \texttt{SELECT}\ [\texttt{DISTINCT}]\ \alpha : \beta'\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta \\ \mid\;& \texttt{SELECT}\ [\texttt{DISTINCT}]\ \star\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta \\ \mid\;& Q\ (\texttt{UNION} \mid \texttt{INTERSECT} \mid \texttt{EXCEPT})\ [\texttt{ALL}]\ Q \end{aligned}$$

CONDITIONS

$$\begin{aligned} \theta \;\coloneqq\;& \texttt{TRUE} \mid \texttt{FALSE} \mid P(t_1, \ldots, t_k),\ P \in \mathcal{P} \\ \mid\;& t\ \texttt{IS}\ [\texttt{NOT}]\ \texttt{NULL} \\ \mid\;& \bar{t}\ [\texttt{NOT}]\ \texttt{IN}\ Q \mid \texttt{EXISTS}\ Q \\ \mid\;& \theta\ \texttt{AND}\ \theta \mid \theta\ \texttt{OR}\ \theta \mid \texttt{NOT}\ \theta \end{aligned}$$

Figure 2: Syntax of basic SQL with a collection of predicates $\mathcal{P}$

$$\ell(R) = \text{tuple of names provided by the schema}$$
$$\ell(\tau) = \ell(T_1) \cdots \ell(T_k) \quad \text{for } \tau = (T_1, \ldots, T_k)$$
$$\ell\left(\begin{array}{l}\texttt{SELECT}\ [\texttt{DISTINCT}]\ \alpha : \beta' \\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta\end{array}\right) = \beta'$$
$$\ell\big(\texttt{SELECT}\ [\texttt{DISTINCT}]\ \star\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta\big) = \ell(\tau)$$
$$\ell\big(Q_1\ (\texttt{UNION} \mid \texttt{INTERSECT} \mid \texttt{EXCEPT})\ [\texttt{ALL}]\ Q_2\big) = \ell(Q_1)$$

Figure 3: Output attributes of basic SQL queries

conditions of the form $\bar{t}$ `IN` Q, for example, the query Q can refer to full names in $\bar{t}$, whose values come from elsewhere. The standard way to account for this in programming semantics [18, 28] is to define an *environment* η that provides values for such parameters. In our case, the parameters are full names, so η 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 η and produces the output of Q. Then, for a query without parameters, we are looking at $[\![Q]\!]_D = [\![Q]\!]_{D,\varnothing}$.

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 `SELECT` $\star$. Normally $\star$ means that all attributes have to be returned, but if such a query occurs under `EXISTS`, then $\star$ 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 $Q =$ `SELECT * FROM (SELECT R.A, R.A FROM R) AS T` will fail due to the ambiguity of the reference to $R.A$,[1] but the query `SELECT * FROM R 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 $\star$ in the `SELECT` clause of queries, we introduce an additional Boolean input to $[\![Q]\!]$. If Q is the outermost query nested inside an `EXISTS` condition, this switch is set to 1, otherwise to 0. Then, when Q is of the form `SELECT` $\star \ldots$, value 1 indicates that $\star$ is to be replaced with an arbitrary constant, and 0 that it must be expanded into a list of full names (provided by the `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 $[\![Q]\!]_D = [\![Q]\!]_{D,\varnothing,0}$.

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 $M.N$ mentioned in the `SELECT` or `WHERE` clause of queries is a reference to some attribute N in some table M. How are references resolved? Each `SELECT-FROM-WHERE` block defines a *scope*, and scopes are nested according to the structure of the query. Then, for each reference $M.N$, we first look for a match (i.e., a table M with an attribute N) in the `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 `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 $N\bullet(N_1, \ldots, N_n)$ 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

$$\ell(\tau : \beta) = N_1\bullet\ell(T_1) \cdots N_k\bullet\ell(T_k)$$

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 $\bar{A} = (A_1, \ldots, A_m)$ and a record $\bar{r} = (a_1, \ldots, a_m)$ of the same length, we define the environment $\eta_{\bar{A},\bar{r}}$ that maps each non-repeated element A_i of $\bar{A}$ to the corresponding value a_i of $\bar{r}$; if A_i occurs more than once in $\bar{A}$, then $\eta_{\bar{A},\bar{r}}$ 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 η and a tuple of full names $\bar{A}$, we denote by $\eta \Uparrow \bar{A}$ the environment obtained by removing from η the bindings for all elements of $\bar{A}$. That is, $\eta \Uparrow \bar{A}$ is undefined on every $A \in \bar{A}$, and it is otherwise identical to η. Given two environments η and η', by $\eta; \eta'$ we mean η overridden by η'. That is, $\eta; \eta'(A) = \eta(A)$ if η is defined on A and η' is not; otherwise $\eta; \eta'(A) = \eta'(A)$. Finally, we can put all of the above together and define

$$\eta \overset{\bar{r}}{\oplus} \bar{A} = (\eta \Uparrow \bar{A}); \eta_{\bar{A},\bar{r}}$$

which updates η by first unbinding the full names in $\bar{A}$ and then overriding the result by $\eta_{\bar{A},\bar{r}}$.

[1] This is the behavior prescribed by the Standard, but not all RDBMSs follow it; see Section 4.

$$[\![t]\!]_\eta = \begin{cases} \eta(A) & \text{if } t = A \\ c & \text{if } t = c \in \mathsf{C} \\ \texttt{NULL} & \text{if } t = \texttt{NULL} \end{cases}$$
$$[\![(t_1, \ldots, t_n)]\!]_\eta = \big([\![t_1]\!]_\eta, \ldots, [\![t_n]\!]_\eta\big)$$

Figure 4: Semantics of SQL terms

Operations on tables. To describe the semantics of SQL queries, we will use some of the standard operations on bags [3, 16, 23]. We denote by $\#(\bar{r}, T)$ the number of occurrences (multiplicity) of a record $\bar{r}$ in table T; if $\bar{r}$ does not occur in T, then $\#(\bar{r}, T) = 0$. We also write $\bar{r} \in_k T$ for $\#(\bar{r}, T) = k$. In addition, we use $\bar{r} \in T$ to indicate that $\bar{r} \in_k T$ for some $k > 0$, and $\bar{r} \notin T$ for $\bar{r} \in_0 T$.

The bag operations $\cup$, $\cap$ and $-$ are defined as follows:

$$\begin{aligned} \#(\bar{t}, T_1 \cup T_2) &= \#(\bar{t}, T_1) + \#(\bar{t}, T_2) \\ \#(\bar{t}, T_1 \cap T_2) &= \min\big(\#(\bar{t}, T_1), \#(\bar{t}, T_2)\big) \\ \#(\bar{t}, T_1 - T_2) &= \max\big(\#(\bar{t}, T_1) - \#(\bar{t}, T_2), 0\big) \end{aligned}$$

Cartesian product $\times$ multiplies the number of occurrences of tuples: that is, $\#\big((\bar{t}_1, \bar{t}_2), T_1 \times T_2\big) = \#(\bar{t}_1, T_1) \cdot \#(\bar{t}_2, T_2)$. Finally, the *duplicate elimination* operation ε turns a bag into a set by removing all but one occurrence of each tuple; formally $\#(\bar{t}, \varepsilon(T)) = \min\big(\#(\bar{t}, T), 1\big)$.

Explanation of the semantics

We now explain the key elements of the semantics, presented in Figures 4–7. The semantic function $[\![\cdot]\!]$ takes different inputs depending on the syntactic construct under consideration: for queries Q the inputs are a database D, an environment η and a Boolean variable x whose value is either 0 or 1; for conditions θ, the inputs are just the database and the environment; for terms t, the only input is the environment.

Terms (see Figure 4) The semantics of a term is given by the environment η: 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 $\bar{t}$ of terms is simply the tuple of values obtained by interpreting each term in $\bar{t}$.

Queries (see Figure 5) A base table R obviously denotes its interpretation in the database, i.e., R^D. The evaluation of a `SELECT-FROM-WHERE` block starts by computing the Cartesian product of the tables produced by the elements of τ, each of which is either a base table or the output of a subquery. When the `WHERE` θ clause is added, the tuples satisfying θ are selected from the product. Observe that in this case the environment changes: when the condition θ 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 `FROM` clause has precedence over the outer scopes. For each record in the product that satisfies θ, the revised environment is then applied to the `SELECT` list α, which may also contain parameters, to produce the final output.

As discussed before, if the `SELECT` list is "$\star$", 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 `EXISTS` condition.

Conditions (see Figure 6) 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 `TRUE` and `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 `NULL`. For equality, which is always assumed to be among the available predicates, we have that $[\![t_1 = t_2]\!]_{D,\eta}$ is **u** if one of $[\![t_1]\!]_\eta$ or $[\![t_2]\!]_\eta$ is `NULL`; if both are elements $c_1, c_2 \in \mathsf{C}$, then the semantics is simply the result of the comparison $c_1 = c_2$.

The condition $\bar{t}$ `IN` Q is the disjunction of all the equalities $\bar{t} = \bar{s}$ for every $\bar{s}$ in the output of Q, while `EXISTS` Q tests for non-emptiness. Note that, among conditions, only the basic predicates $P \in \mathcal{P}$ and $\bar{t}$ `IN` Q can produce the truth value **u**; this is then propagated through the connectives $\wedge$, $\vee$ and $\neg$ following the truth tables of SQL's 3VL (Figure 1), which corresponds to what is known as the Kleene logic [6].

Operations (see Figure 7) `UNION ALL`, `INTERSECT ALL`, and `EXCEPT ALL` are the bag operations $\cup$, $\cap$, and $-$ we described before. Without the keyword `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 the queries Q_1–Q_3 from the introduction produce exactly the same results as they should, namely $\varnothing$, $\{1, \texttt{NULL}\}$ and $\{1\}$ on a database with $R = \{1, \texttt{NULL}\}$ and $S = \{\texttt{NULL}\}$.

As for the queries in Example 2, the first will be rejected because it will force a `SELECT` list containing an ambiguous reference (i.e., a full name that is repeated in the `FROM` clause of the scope against which it resolves). On the other hand, the second will be allowed, because the `SELECT` $\star$ subquery is nested directly under `EXISTS`, so $\star$ will be replaced by an arbitrary constant and no ambiguity occurs.

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 queries for validating the semantics.

4. EXPERIMENTAL VALIDATION

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, 22]. 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, `SELECT` $\star$ behaves in the same way regardless of the context in which the query is used. This means that the extra Boolean switch is

$$
\begin{aligned}
[\![R]\!]_{D,\eta,x} &= R^D \\
[\![\tau : \beta]\!]_{D,\eta,x} &= [\![T_1]\!]_{D,\eta,0} \times \cdots \times [\![T_k]\!]_{D,\eta,0} \quad \text{for } \tau = (T_1, \ldots, T_k) \\
\left[\!\!\left[\begin{array}{ll} \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,x} &= \left\{ \underbrace{\bar{r}, \ldots, \bar{r}}_{k \text{ times}} \;\middle|\; \bar{r} \in_k [\![\tau : \beta]\!]_{D,\eta,0},\ [\![\theta]\!]_{D,\eta'} = \mathbf{t},\ \eta' = \eta \overset{\bar{r}}{\oplus} \ell(\tau : \beta) \right\} \\
\left[\!\!\left[\begin{array}{ll} \texttt{SELECT} & \alpha : \beta' \\ \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,x} &= \left\{ \underbrace{[\![\alpha]\!]_{\eta'}, \ldots, [\![\alpha]\!]_{\eta'}}_{k \text{ times}} \;\middle|\; \eta' = \eta \overset{\bar{r}}{\oplus} \ell(\tau : \beta),\ \bar{r} \in_k \left[\!\!\left[\begin{array}{ll} \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,x} \right\} \\
\left[\!\!\left[\begin{array}{ll} \texttt{SELECT} & \star \\ \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,0} &= \left[\!\!\left[\begin{array}{ll} \texttt{SELECT} & \ell(\tau : \beta) : \ell(\tau) \\ \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,0} \\
\left[\!\!\left[\begin{array}{ll} \texttt{SELECT} & \star \\ \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,1} &= \left[\!\!\left[\begin{array}{ll} \texttt{SELECT} & c \ \texttt{AS}\ N \\ \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,1} \quad \text{for arbitrary } c \in \mathsf{C} \text{ and } N \in \mathsf{N} \\
\left[\!\!\left[\begin{array}{l} \texttt{SELECT DISTINCT}\ \alpha : \beta' \mid \star \\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta \end{array} \right]\!\!\right]_{D,\eta,x} &= \varepsilon\left(\left[\!\!\left[\begin{array}{l} \texttt{SELECT}\ \alpha : \beta' \mid \star \\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta \end{array} \right]\!\!\right]_{D,\eta,x} \right)
\end{aligned}
$$

Figure 5: Semantics of basic SQL – Queries

$$
\begin{aligned}
[\![P(t_1, \ldots, t_k)]\!]_{D,\eta} &= \begin{cases} \mathbf{t} & \text{if } P\big([\![t_1]\!]_\eta, \ldots, [\![t_k]\!]_\eta\big) \text{ holds and } [\![t_i]\!]_\eta \neq \texttt{NULL} \text{ for all } i \in \{1, \ldots, k\} \\ \mathbf{f} & \text{if } P\big([\![t_1]\!]_\eta, \ldots, [\![t_k]\!]_\eta\big) \text{ does not hold and } [\![t_i]\!]_\eta \neq \texttt{NULL} \text{ for all } i \in \{1, \ldots, k\} \\ \mathbf{u} & \text{if } [\![t_i]\!]_\eta = \texttt{NULL} \text{ for some } i \in \{1, \ldots, k\} \end{cases} \\
[\![t\ \texttt{IS NULL}]\!]_{D,\eta} &= \begin{cases} \mathbf{t} & \text{if } [\![t]\!]_\eta = \texttt{NULL} \\ \mathbf{f} & \text{if } [\![t]\!]_\eta \neq \texttt{NULL} \end{cases} \\
[\![t\ \texttt{IS NOT NULL}]\!]_{D,\eta} &= \neg [\![t\ \texttt{IS NULL}]\!]_{D,\eta} \\
[\![(t_1, \ldots t_n) = (t'_1, \ldots, t'_n)]\!]_{D,\eta} &= \bigwedge_{i=1}^{n} [\![t_i = t'_i]\!]_{D,\eta} \qquad [\![(t_1, \ldots t_n) \neq (t'_1, \ldots, t'_n)]\!]_{D,\eta} = \bigvee_{i=1}^{n} [\![t_i \neq t'_i]\!]_{D,\eta} \\
[\![\bar{t}\ \texttt{IN}\ Q]\!]_{D,\eta} &= \begin{cases} \mathbf{t} & \text{if } \exists \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{t} \\ \mathbf{f} & \text{if } \forall \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{f} \\ \mathbf{u} & \text{if } \nexists \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{t} \text{ and } \exists \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} \neq \mathbf{f} \end{cases} \\
[\![\bar{t}\ \texttt{NOT IN}\ Q]\!]_{D,\eta} &= \neg [\![\bar{t}\ \texttt{IN}\ Q]\!]_{D,\eta} \\
[\![\texttt{EXISTS}\ Q]\!]_{D,\eta} &= \begin{cases} \mathbf{t} & \text{if } [\![Q]\!]_{D,\eta,1} \neq \varnothing \\ \mathbf{f} & \text{if } [\![Q]\!]_{D,\eta,1} = \varnothing \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{TRUE}]\!]_{D,\eta} &= \mathbf{t} & [\![\theta_1\ \texttt{AND}\ \theta_2]\!]_{D,\eta} &= [\![\theta_1]\!]_{D,\eta} \wedge [\![\theta_2]\!]_{D,\eta} & [\![\texttt{NOT}\ \theta]\!]_{D,\eta} &= \neg [\![\theta]\!]_{D,\eta} \\
[\![\texttt{FALSE}]\!]_{D,\eta} &= \mathbf{f} & [\![\theta_1\ \texttt{OR}\ \theta_2]\!]_{D,\eta} &= [\![\theta_1]\!]_{D,\eta} \vee [\![\theta_2]\!]_{D,\eta}
\end{aligned}
$$

Figure 6: Semantics of basic SQL – Conditions

$$
\begin{aligned}
[\![Q_1\ \texttt{UNION ALL}\ Q_2]\!]_{D,\eta,x} &= [\![Q_1]\!]_{D,\eta,0} \cup [\![Q_2]\!]_{D,\eta,0} & [\![Q_1\ \texttt{UNION}\ Q_2]\!]_{D,\eta,x} &= \varepsilon\big([\![Q_1\ \texttt{UNION ALL}\ Q_2]\!]_{D,\eta,x}\big) \\
[\![Q_1\ \texttt{INTERSECT ALL}\ Q_2]\!]_{D,\eta,x} &= [\![Q_1]\!]_{D,\eta,0} \cap [\![Q_2]\!]_{D,\eta,0} & [\![Q_1\ \texttt{INTERSECT}\ Q_2]\!]_{D,\eta,x} &= \varepsilon\big([\![Q_1\ \texttt{INTERSECT ALL}\ Q_2]\!]_{D,\eta,x}\big) \\
[\![Q_1\ \texttt{EXCEPT ALL}\ Q_2]\!]_{D,\eta,x} &= [\![Q_1]\!]_{D,\eta,0} - [\![Q_2]\!]_{D,\eta,0} & [\![Q_1\ \texttt{EXCEPT}\ Q_2]\!]_{D,\eta,x} &= \varepsilon\big([\![Q_1]\!]_{D,\eta,0}\big) - [\![Q_2]\!]_{D,\eta,0}
\end{aligned}
$$

Figure 7: Semantics of basic SQL – Operations

no longer needed and we just need to provide the semantics $[\![Q]\!]_{D,\eta}$. The rule for `SELECT` $\star$ then simply changes to

$$\left[\!\!\left[\begin{array}{ll} \texttt{SELECT} & * \\ \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta} = \left[\!\!\left[\begin{array}{ll} \texttt{FROM} & \tau : \beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta}$$

Other systems slightly change the syntax; for example Oracle uses `MINUS` instead of `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 four components:

1. the correctness criterion;
2. the query generator for the experiments;
3. the implementation of the formal semantics; and
4. the 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 [35], 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 analyze the structure and features of their queries and, based on these, generate a large number of queries that look somewhat like those found in benchmarks.

Towards that goal, we look at the characteristics of the TPC-H benchmark. There eight base tables in total, but on average each benchmark query uses only 3.2, and all queries but one use 6 or fewer. Each query uses relatively few `WHERE` conditions per block, in fact only three queries use more than 8 conditions, and no query exceeds 3 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:

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

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

Implementation of the semantics. We implemented the semantics of Figures 4–7 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 `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 [12]. 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 `SELECT` $\star$ 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 4–7 is correct.*

To sum up, our experiments validate the semantics of Section 3, and allow us to proceed to use this semantics in two applications that formally prove results about real-life SQL.

5. EQUIVALENCE WITH 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 procedural query language, relational algebra. First shown by Codd in 1971 [11], 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 the capabilities of relational calculus, but its core, called basic SQL here, is based on relational calculus. The claim of equivalence between relational calculus and algebra is often extended to include also basic SQL, but these claims tend to be supported only by a few examples of translating SQL queries into RA, rather than detailed translations [2, 33, 15].

Detailed translations of SQL to RA that appeared in the literature [7, 36] made simplifying assumptions that deviate significantly from the behavior of SQL specified by the Standard. As already explained in the introduction, typical simplifications are using set semantics, and omitting nulls and the three-valued logic associated with them. Thus, one can legitimately question whether the textbook equivalence of basic SQL and RA has really been proved.

Of course the main obstacle to having such a proof was the lack of a formal semantics of SQL accounting for its real-life, rather than simplified, features. Now that we have a formal semantics that has been experimentally validated, we can provide a proof that basic SQL is equivalent to RA.

However, even the basic fragment of SQL as we defined it has some features that go beyond RA. Indeed, their data models differ slightly (for RA, it is assumed that attributes

$$[\![R]\!]_D = R^D$$

$$[\![\pi_\beta(E)]\!]_D = \Big\{ \underbrace{[\![\beta]\!]_{\eta^{\bar{a}}_{\ell(E)}}, \ldots, [\![\beta]\!]_{\eta^{\bar{a}}_{\ell(E)}}}_{k \text{ times}} \;\Big|\; \bar{a} \in_k [\![E]\!]_D \Big\}$$

$$[\![\sigma_\theta(E)]\!]_D = \Big\{ \underbrace{\bar{a}, \ldots, \bar{a}}_{k \text{ times}} \;\Big|\; \bar{a} \in_k [\![E]\!]_D \text{ and } [\![\theta]\!]_{D,\eta^{\bar{a}}_{\ell(E)}} = \mathbf{t} \Big\}$$

$$[\![\rho_{\beta\to\beta'}(E)]\!]_D = [\![E]\!]_D$$

$$[\![E_1 \;\mathsf{op}\; E_2]\!]_D = [\![E_1]\!]_D \;\mathsf{op}\; [\![E_2]\!]_D \quad \text{for } \mathsf{op} \in \{\cup, \cap, -, \times\}$$

$$[\![\varepsilon(E)]\!]_D = \varepsilon\big([\![E]\!]_D\big)$$

$$[\![t]\!]_\eta = \begin{cases} \eta(t) & \text{if } t \in \mathsf{N} \\ c & \text{if } t = c \in \mathsf{C} \\ \texttt{NULL} & \text{if } t = \texttt{NULL} \end{cases} \quad ; \quad [\![(t_1, \ldots, t_k)]\!]_\eta = \big([\![t_1]\!]_\eta, \ldots, [\![t_k]\!]_\eta\big)$$

$$[\![P(t_1, \ldots, t_k)]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } P\big([\![t_1]\!]_\eta, \ldots, [\![t_k]\!]_\eta\big) \text{ holds and } [\![t_i]\!]_\eta \neq \texttt{NULL} \text{ for all } i \in \{1, \ldots, k\} \\ \mathbf{f} & \text{if } P\big([\![t_1]\!]_\eta, \ldots, [\![t_k]\!]_\eta\big) \text{ does not hold and } [\![t_i]\!]_\eta \neq \texttt{NULL} \text{ for all } i \in \{1, \ldots, k\} \\ \mathbf{u} & \text{if } [\![t_i]\!]_\eta = \texttt{NULL} \text{ for some } i \in \{1, \ldots, k\} \end{cases}$$

$$[\![\mathsf{null}(t)]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } [\![t]\!]_\eta = \texttt{NULL} \\ \mathbf{f} & \text{if } [\![t]\!]_\eta \neq \texttt{NULL} \end{cases}$$

$$[\![\texttt{TRUE}]\!]_{D,\eta} = \mathbf{t} \; ; \; [\![\texttt{FALSE}]\!]_{D,\eta} = \mathbf{f} \; ; \; [\![\theta_1 \wedge \theta_2]\!]_{D,\eta} = [\![\theta_1]\!]_{D,\eta} \wedge [\![\theta_2]\!]_{D,\eta} \; ; \; [\![\theta_1 \vee \theta_2]\!]_{D,\eta} = [\![\theta_1]\!]_{D,\eta} \vee [\![\theta_2]\!]_{D,\eta} \; ; \; [\![\neg\theta]\!]_{D,\eta} = \neg[\![\theta]\!]_{D,\eta}$$

(under three-valued interpretation of $\wedge, \vee, \neg$)

Figure 8: Semantics of relational algebra

cannot repeat [2, 33]), 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 expressive power of data manipulating queries in basic SQL is the same as RA under bag semantics.

Syntax of bag relational algebra. 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:

$$\begin{array}{ll} E \coloneqq R & \text{(base relation)} \\ \mid \pi_\beta(E) & \text{(projection)} \\ \mid \sigma_\theta(E) & \text{(selection)} \\ \mid E \times E & \text{(product)} \\ \mid E \cup E & \text{(union)} \\ \mid E \cap E & \text{(intersection)} \\ \mid E - E & \text{(difference)} \\ \mid \rho_{\beta\to\beta'}(E) & \text{(renaming)} \\ \mid \varepsilon(E) & \text{(duplicate elimination)} \end{array}$$

In relational algebra, terms are given by $t \coloneqq N \mid c \mid \texttt{NULL}$ where N now ranges over N, and c ranges over C as before. Note that β, β' in the grammar above are tuples of names.

For a given collection of predicates $P \in \mathcal{P}$, the conditions θ in selections are given by

$$\theta \coloneqq \texttt{TRUE} \mid \texttt{FALSE} \mid P(\bar{t}) \mid \mathsf{const}(t) \mid \mathsf{null}(t) \mid \theta \wedge \theta \mid \theta \vee \theta \mid \neg\theta$$

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 $\mathsf{null}(t)$ tests if the value of a term is null, and $\mathsf{const}(t)$ is the negation of $\mathsf{null}(t)$.

We define the *signature* $\ell(E)$ of an expression, i.e., the list of attribute names of the table that E generates, as follows:

$$\begin{aligned} \ell(R) &= \text{list of attributes of base relation } R \\ \ell(E_1 \times E_2) &= \ell(E_1)\,\ell(E_2) \\ \ell(E_1 \;\mathsf{op}\; E_2) &= \ell(E_1) \text{ for } \mathsf{op} \in \{\cup, \cap, -\} \\ \ell\big(\pi_\beta(E)\big) &= \beta \\ \ell\big(\mathsf{op}(E)\big) &= \ell(E) \text{ for } \mathsf{op} \in \{\sigma, \varepsilon\} \\ \ell(\rho_{\beta\to\beta'}(E)) &= \beta' \end{aligned}$$

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

Semantics of bag relational algebra. The semantics of well-defined RA expressions is given in Figure 8. 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 discussed in Section 3.

The environment η 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 $\eta^{\bar{a}}_\beta$ is defined so that $\eta^{\bar{a}}_\beta(N_i) =$

a_i for $i \in \{1, \ldots, m\}$. Since no repetitions in β are allowed, $\eta^{\bar{a}}_{\beta}$ is always well defined.

Note that the semantics of projection is the standard one under bag semantics; for example, for a base table $R(A, B)$ with $R^D = \{(a, b), (a, c)\}$ we get $[\![\pi_A(R)]\!]_D = \{a, a\}$.

In the semantics of conditions, equality is interpreted under 3VL, while $\mathsf{null}(x)$ has a two-valued ($\mathbf{t}/\mathbf{f}$) interpretation. Boolean connectives follow the rules of SQL's 3VL.

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 **SELECT** [**DISTINCT**] $\alpha : \beta'$ **FROM** $\tau : \beta$ **WHERE** θ so that the names in β' do not repeat, and for each $A = N_1.N_2 \in \alpha$, the name N_1 occurs in β.

Thus, we simply disallow repetition of column names in query/subquery results, force the attributes in **SELECT** to be listed explicitly, and only allow data from relations/queries in the **FROM** clause to appear in **SELECT**. Observe that we do not forbid duplication of columns per se: for example, one can write `SELECT R.A AS A1, R.A AS A2 FROM R`; we only require that the columns be named differently in the output.

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 the translations of [7, 36]. Translating RA 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:

$$\begin{aligned}\theta \coloneqq\ & \mathbf{TRUE} \mid \mathbf{FALSE} \mid P(\bar{t}) \mid \mathsf{const}(t) \mid \mathsf{null}(t) \\ & \mid \theta \wedge \theta \mid \theta \vee \theta \mid \neg\theta \mid \bar{t} \in E \mid \mathsf{empty}(E)\end{aligned}$$

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

To extend the semantics, we now require every expression, not just conditions, to carry an environment η, which only changes in one case:

$$[\![\sigma_\theta(E)]\!]_{D,\eta} = \left\{ \underbrace{\bar{a}, \ldots, \bar{a}}_{k \text{ times}} \;\middle|\; \bar{a} \in_k [\![E]\!]_D \text{ and } [\![\theta]\!]_{D,\eta;\eta^{\bar{a}}_{\ell(E)}} = \mathbf{t} \right\}$$

The semantics of the additional conditions is as follows:

$$[\![\bar{t} \in E]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } \exists \bar{r} \in [\![E]\!]_{D,\eta} \colon [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{t} \\ \mathbf{f} & \text{if } \forall \bar{r} \in [\![E]\!]_{D,\eta} \colon [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{f} \\ \mathbf{u} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{empty}(Q)]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } [\![Q]\!]_{D,\eta} = \varnothing \\ \mathbf{f} & \text{if } [\![Q]\!]_{D,\eta} \neq \varnothing \end{cases}$$

where, for $\bar{r} = (r_1, \ldots, r_m)$ and $\bar{s} = (s_1, \ldots, s_m)$, we define $[\![\bar{r} = \bar{s}]\!]_{D,\eta}$ as $\bigwedge\{[\![r_i = s_i]\!]_{D,\eta} \mid 1 \le i \le m\}$.

Some expressions of SQL-RA have *parameters*: for instance, expressions E in $\bar{t} \in E$ can refer to values in $\bar{t}$. 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 $\mathsf{param}(E)$, and the set of parameters of a condition θ with respect to a set of attribute names $\mathcal{A}$, denoted by $\mathsf{param}(\theta, \mathcal{A})$, are defined by mutual recursion as follows. Below op stands for binary operations $\times, \cup, \cap, -$; conn for connectives $\wedge, \vee$, and names, applied to a set or tuple of terms returns those terms that are names (as opposed to constants and nulls).

$$\begin{aligned}\mathsf{param}(R) &= \varnothing \\ \mathsf{param}(E_1 \mathsf{\ op\ } E_2) &= \mathsf{param}(E_1) \cup \mathsf{param}(E_2) \\ \mathsf{param}\big(\pi_\alpha(E)\big) &= \mathsf{param}(E) \\ \mathsf{param}\big(\sigma_\theta(E)\big) &= \mathsf{param}\big(\theta, \{A \mid A \in \ell(E)\}\big) \\ \mathsf{param}\big(P(t_1, \ldots, t_k), \mathcal{A}\big) &= \mathsf{names}(\{t_1, \ldots, t_k\}) - \mathcal{A} \\ \mathsf{param}(\theta_1 \mathsf{\ conn\ } \theta_2, \mathcal{A}) &= \mathsf{param}(\theta_1, \mathcal{A}) \cup \mathsf{param}(\theta_2, \mathcal{A}) \\ \mathsf{param}(\neg\theta, \mathcal{A}) &= \mathsf{param}(\theta, \mathcal{A}) \\ \mathsf{param}\big(\mathsf{empty}(E), \mathcal{A}\big) &= \mathsf{param}(E) - \mathcal{A} \\ \mathsf{param}(\bar{t} \in E, \mathcal{A}) &= \big(\mathsf{names}(\bar{x}) \cup \mathsf{param}(E)\big) - \mathcal{A}\end{aligned}$$

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

Proposition 1. *For every data manipulation query in basic SQL, there is an equivalent SQL-RA query.*

The translation closely follows the structure of queries, once we have resolved two mismatches. The first is about the use of full names (i.e., elements of N^2) in basic SQL vs. the use of names from N in SQL-RA. The second is about SQL projection vs. SQL-RA projection.

To address the first mismatch, we can simulate all of the full names used in a query with extra names from N, since we have an infinite supply of them. For a given SQL query Q, we define an injective mapping $\chi \colon \mathsf{N}^2 \to \mathsf{N} - (\mathsf{N}_Q \cup \mathsf{N}_{\text{base}})$, where N_Q is the set of all names occurring in the rename list of each **SELECT** clause in Q, and N_{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 $N_1, \ldots, N_m \in \mathsf{N}$ and a name $N \in \mathsf{N}$, we let

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

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 χ as defined above, and an SQL environment $\eta \colon \mathsf{N}^2 \to \mathsf{C} \cup \{\mathbf{NULL}\}$, the SQL-RA environment corresponding to η w.r.t. χ, denoted by η^χ, is the partial mapping $\eta \circ \zeta \colon \mathsf{N} \to \mathsf{C} \cup \{\mathbf{NULL}\}$, where ζ is the left inverse of χ.

When it comes to projection, SQL's **SELECT** α may have repetitions of attributes, in which case we cannot use it directly in SQL-RA. But it always comes together with a renaming β 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

$$R \overset{\chi}{\mapsto} R \quad \text{if } R \text{ is the name of a base relation}$$

$$(T_1,\ldots,T_k):(N_1,\ldots,N_k) \overset{\chi}{\mapsto} \rho^{\chi}_{N_1}(E_1)\times\cdots\times\rho^{\chi}_{N_k}(E_k) \quad \text{if } T_i \overset{\chi}{\mapsto} E_i, \text{ for } 1\le i\le k$$

$$\begin{array}{rll}\texttt{SELECT}\,[\texttt{DISTINCT}] & \alpha:\beta' & \\ \texttt{FROM} & \tau:\beta & \overset{\chi}{\mapsto}\ [\varepsilon]\,\pi^{\chi(\alpha)}_{\beta'}\big(\sigma_{\theta'}(E)\big) \quad \text{if } \tau:\beta \overset{\chi}{\mapsto} E \text{ and } \theta \overset{\chi}{\mapsto} \theta' \\ \texttt{WHERE} & \theta & \end{array}$$

$$t \overset{\chi}{\mapsto} \begin{cases} t & \text{if } t \in \mathsf{C}\cup\{\texttt{NULL}\} \\ \chi(t) & \text{otherwise}\end{cases}$$

$$b \overset{\chi}{\mapsto} b \qquad \text{if } b = \texttt{TRUE} \text{ or } \texttt{FALSE}$$

$$t\ \texttt{IS}\,[\texttt{NOT}]\,\texttt{NULL} \overset{\chi}{\mapsto} [\neg]\,\mathsf{null}(\hat{t}) \qquad \text{if } t \overset{\chi}{\mapsto} \hat{t}$$

$$P(t_1,\ldots,t_n) \overset{\chi}{\mapsto} P(\hat{t}_1,\ldots,\hat{t}_n) \qquad \text{if } t_i \overset{\chi}{\mapsto} \hat{t}_i \text{ and } t'_i \overset{\chi}{\mapsto} \hat{t}'_i$$

$$\texttt{EXISTS}\ Q \overset{\chi}{\mapsto} \neg\,\mathsf{empty}(E) \qquad \text{if } Q \overset{\chi}{\mapsto} E$$

$$\bar{t}\,[\texttt{NOT}]\,\texttt{IN}\ Q \overset{\chi}{\mapsto} [\neg]\big((\hat{t}_1,\ldots,\hat{t}_n)\in E\big) \qquad \text{if } Q \overset{\chi}{\mapsto} E \text{ and } \bar{t} \overset{\chi}{\mapsto} (\hat{t}_1,\ldots,\hat{t}_n)$$

For $\theta_1 \overset{\chi}{\mapsto} \theta'_1$ and $\theta_2 \overset{\chi}{\mapsto} \theta'_2$: $\quad \theta_1\ \texttt{AND}\ \theta_2 \overset{\chi}{\mapsto} \theta'_1\wedge\theta'_2$, $\quad \theta_1\ \texttt{OR}\ \theta_2 \overset{\chi}{\mapsto} \theta'_1\vee\theta'_2$, $\quad \texttt{NOT}\ \theta_1 \overset{\chi}{\mapsto} \neg\theta'_1$

For $Q_1 \overset{\chi}{\mapsto} E_1$ and $Q_2 \overset{\chi}{\mapsto} E_2$:

$$Q_1\ \texttt{UNION ALL}\ Q_2 \overset{\chi}{\mapsto} E_1 \cup \rho_{\ell(Q_2)\to\ell(Q_1)}(E_2) \qquad Q_1\ \texttt{UNION}\ Q_2 \overset{\chi}{\mapsto} \varepsilon\big(E_1 \cup \rho_{\ell(Q_2)\to\ell(Q_1)}(E_2)\big)$$

$$Q_1\ \texttt{INTERSECT ALL}\ Q_2 \overset{\chi}{\mapsto} E_1 \cap \rho_{\ell(Q_2)\to\ell(Q_1)}(E_2) \qquad Q_1\ \texttt{INTERSECT}\ Q_2 \overset{\chi}{\mapsto} \varepsilon\big(E_1 \cap \rho_{\ell(Q_2)\to\ell(Q_1)}(E_2)\big)$$

$$Q_1\ \texttt{EXCEPT ALL}\ Q_2 \overset{\chi}{\mapsto} E_1 - \rho_{\ell(Q_2)\to\ell(Q_1)}(E_2) \qquad Q_1\ \texttt{EXCEPT}\ Q_2 \overset{\chi}{\mapsto} \varepsilon(E_1) - \varepsilon\big(\rho_{\ell(Q_2)\to\ell(Q_1)}(E_2)\big)$$

Figure 9: Translation from basic SQL data manipulation queries to SQL-RA, under renaming χ

to introduce a definition that will also be important in the next section.

Definition 2. The *syntactic equality* of terms, denoted by $t_1 \doteq t_2$, is the comparison with the following semantics:

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

In other words, two terms are syntactically equal iff they refer to the same constant or `NULL`. Syntactic equality does not add expressive power, because $t_1 \doteq t_2$ is equivalent to $\big(t_1 = t_2 \wedge \mathsf{const}(t_1) \wedge \mathsf{const}(t_2)\big) \vee \big(\mathsf{null}(t_1) \wedge \mathsf{null}(t_2)\big)$.

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 tuple of distinct names that do not appear in $\ell(E)$, then we define $\pi^{\alpha}_{\beta}(E)$ as

$$\begin{cases}\rho_{\alpha\to\beta}(\pi_\alpha(E)), & \text{if } \alpha \text{ has no repetitions,} \\ \pi_\beta\Big(\sigma_{\alpha\doteq\beta}\big(E \bowtie^s \big(\bigtimes^S_{i=1,\ldots,n} \varepsilon(\rho_{\alpha_i\to\beta_i}(E))\big)\big)\Big) & \text{otherwise,}\end{cases}$$

where $\bowtie^s$ is syntactic natural join, i.e., natural join where the comparison condition on common attributes is syntactic equality. Note that the projection operation is straightforward when 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.

The translation of basic SQL data manipulation queries to SQL-RA, under renaming χ, is defined in Figure 9. The proof of correctness proceeds by induction on the structure of queries and conditions.

We then need the second component of the proof of 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 $\bar{t} \in E$ conditions, replacing them with emptiness conditions instead. Then one translates the resulting expression into a normal form where each condition is either a predicate $P(\bar{t})$, or $\mathsf{empty}(E)$, or their negations. Finally, $\sigma_{\mathsf{empty}(E)}(E')$ and $\sigma_{\neg\,\mathsf{empty}(E)}(E')$ are translated into left (anti) semijoins of E and E'.

This completes the equivalence proof. □

Example. We again return to the queries Q_1–Q_3 from the introduction, which provide three non-equivalent ways of expressing the 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 $R' = \rho_{A\to B}(R)$ and $S' = \rho_{A\to C}(S)$:

$$\begin{aligned} Q_1 &= \rho_{B\to A}\big(\varepsilon(R') \mathbin{\overline{\ltimes}^s} \sigma_{B=C}(R'\times S')\big)\,, \\ Q_2 &= \rho_{B\to A}\big(\varepsilon(R') \mathbin{\overline{\ltimes}^s} \sigma_{B=C\vee\mathsf{null}(B)\vee\mathsf{null}(C)}(R'\times S')\big)\,, \\ Q_3 &= \varepsilon(R) - S\,. \end{aligned}$$

The operation $\mathbin{\overline{\ltimes}^s}$ in the first two expressions above is (left) antijoin based on syntactic natural join, and it is defined as $E_1 \mathbin{\overline{\ltimes}^s} E_2 = E_1 - E_1 \cap \pi_{\ell(E_1)}(E_1 \bowtie^s E_2)$.

6. THREE-VALUED LOGIC

It is common belief that to evaluate SQL queries we need three-valued logic. The description of the semantics, which

follows the Standard, and the equivalence to three-valued RA seem to confirm this so far. But if 3VL is really necessary, what are the queries that we miss if we use the standard Boolean logic with only **t** and **f**? 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 the semantics of SQL under two-valued logic, we need to analyze when the third truth value, *unknown* (**u**), appears. There is only one base case, for predicates $P \in \mathcal{P}$. This includes equality $=$, which we always assume to be present. Then **u** propagates further through the logical connectives (**AND**, **OR**, **NOT**), and through conditions $\bar{t}$ **IN** Q, which amount to disjunctions of $\bar{t} = \bar{r}$ for $\bar{r}$ in the result of Q.

To give a two-valued semantics to predicates, we can conflate **f** and **u**. This is a rather natural decision in the context of SQL: after all, when the **WHERE** clause is evaluated under 3VL, only the tuples for which the condition evaluates to **t** are kept, while those for which it is **f** or **u** are discarded.

We thus turn the semantics $[\![\,]\!]$ of Figures 4–7 into a two-valued semantics $[\![\,]\!]^{2v}$ by changing the rules for all predicates $P \in \mathcal{P}$ as follows:

$$[\![P(t_1,\ldots,t_k)]\!]^{2v}_{D,\eta} = \begin{cases} \mathbf{t} & P([\![t_1]\!]^{2v}_\eta,\ldots,[\![t_k]\!]^{2v}_\eta) \text{ holds} \\ & \text{and } [\![t_i]\!]^{2v}_\eta \neq \texttt{NULL},\ 1 \le i \le k \\ \mathbf{f} & \text{otherwise} \end{cases}$$

For the equality predicate $=$, there is another option: we can interpret it as syntactic equality $\doteq$ (Definition 2) whose semantics is already guaranteed to produce only truth values in $\{\mathbf{t},\mathbf{f}\}$. That is, instead of defining the two-valued semantics of equality like other predicates as above, we could take $[\![t_1 = t_2]\!]^{2v}_{D,\eta} = [\![t_1 \doteq t_2]\!]_{D,\eta}$.

It turns out that it does not matter which of these two alternatives we choose for equality: in either case the resulting two-valued semantics captures the behavior of SQL.

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 exist queries Q' and Q'' such that $[\![Q]\!]_D = [\![Q']\!]^{2v}_D$ and $[\![Q]\!]^{2v}_D = [\![Q'']\!]_D$, for all databases D. This is true for either interpretation of equality under two-valued semantics.*

Proof outline. The two-valued semantics of any predicate in $\mathcal{P}$ can be expressed in the three-valued one by simply taking its conjunction with conditions checking that its arguments are not null. Since syntactic equality is expressible as well, the translations from two-valued to three-valued semantics are immediate.

We thus concentrate on producing, from a query Q (potentially with parameters), another query Q' so that $[\![Q]\!]_{D,\eta} = [\![Q']\!]^{2v}_{D,\eta}$. This is done by defining three translations by mutual induction:

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

$$[\![\theta]\!]_{D,\eta} = \mathbf{t} \quad\Leftrightarrow\quad [\![\theta^{\mathbf{t}}]\!]^{2v}_{D,\eta} = \mathbf{t}$$
$$[\![\theta]\!]_{D,\eta} = \mathbf{f} \quad\Leftrightarrow\quad [\![\theta^{\mathbf{f}}]\!]^{2v}_{D,\eta} = \mathbf{t}$$

- from queries Q to Q' by inductively replacing each condition θ by $\theta^{\mathbf{t}}$.

That is, the conditions $\theta^{\mathbf{t}}$ and $\theta^{\mathbf{f}}$ describe, under two-valued semantics, the behavior of θ under three-valued semantics (checking whether $[\![\theta]\!] = \mathbf{u}$ is captured by $\neg\theta^{\mathbf{t}} \wedge \neg\theta^{\mathbf{f}}$), and Q' simply replaces θ with $\theta^{\mathbf{t}}$, since we only select tuples for which the condition is true.

The translations of conditions are shown in Figure 10 for the case when the two-valued interpretation of equality is the same as for all other predicates $P \in \mathcal{P}$. In the **f**-translation of **IN**, we make use of the SQL construct T **AS** $N(A_1,\ldots,A_n)$ that assigns the name N to an n-ary table T and at the same time renames its attributes to $A_1,\ldots,A_n$. We assume that the names $N, A_1,\ldots,A_n$ are fresh and distinct.

When equality is interpreted as syntactic equality, we need to add the rules

$$(t_1 = t_2)^{\mathbf{t}} \;=\; (t_1 = t_2) \texttt{ AND } (t_1, t_2) \texttt{ IS NOT NULL}$$
$$(t_1 = t_2)^{\mathbf{f}} \;=\; \texttt{NOT } (t_1 = t_2) \texttt{ AND } (t_1, t_2) \texttt{ IS NOT NULL}$$

and replace the comparisons $t_i = N.A_i$ in the **f**-translation of **IN** by $(t_i = N.A_i)^{\mathbf{t}}$. In addition, we also need to change the **t**-translation of $(t_1,\ldots,t_n)$ **IN** Q into

$$\texttt{EXISTS } \big(\texttt{SELECT } \star \texttt{ FROM } Q' \texttt{ AS } N(A_1,\ldots,A_n) \texttt{ WHERE } (t_1 = N.A_1)^{\mathbf{t}} \texttt{ AND } \cdots \texttt{ AND } (t_n = N.A_n)^{\mathbf{t}}\big)$$

Then, by induction on queries and conditions, we can verify that $[\![Q]\!]_{D,\eta} = [\![Q']\!]^{2v}_{D,\eta}$ for every D and η. □

SQL and 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 [13]? We argue that at present, despite the established equivalence result, we are not yet ready to abandon SQL's 3VL.

To start with, there is a huge amount of legacy code out there that assumes query evaluation under 3VL. Suppose for a minute that SQL did switch to a two-valued interpretation. Then, legacy queries need to be rewritten so as to give the same results as they used to. Emulating old behavior turns into case analysis, which leads to more cumbersome and less efficient queries. Indeed, even simple forms of case analysis introduce extra disjunctions whenever negations occur, and it is well known and documented that commercial optimizers struggle with queries involving disjunctions [10].

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. RELATED WORK

Given the problems of using the Standard as the definition of a formal semantics, there have been attempts at formalizing SQL in the past. Several of them go via translating SQL queries into RA, whose formal semantics has been properly defined. Database texts (e.g., [2, 33, 15]) only give examples of informal SQL-to-RA translations for simple queries. Formal translations did appear [7, 36] but assumed set semantics and absence of nulls. As discussed in the introduction, these assumptions allow for simplifications that do not hold in general, such as converting a **NOT IN** subquery into a **NOT EXISTS** one, or a query with a **WHERE** condition in disjunctive normal form into a union of queries.

Bag semantics was actively studied in the 1990s, with [3] proposing the set of RA operations on bags we used here.

$$
\begin{array}{rcl@{\qquad}rcl}
P(\bar{t})^{\mathbf{t}} & = & P(\bar{t}) & P(t_1,\ldots,t_k)^{\mathbf{f}} & = & \texttt{NOT } P(t_1,\ldots,t_k) \texttt{ AND } \bar{t} \texttt{ IS NOT NULL} \\
(\texttt{EXISTS } Q)^{\mathbf{t}} & = & \texttt{EXISTS } Q' & (\texttt{EXISTS } Q)^{\mathbf{f}} & = & \texttt{NOT EXISTS } Q' \\
(\theta_1 \wedge \theta_2)^{\mathbf{t}} & = & \theta_1^{\mathbf{t}} \wedge \theta_2^{\mathbf{t}} & (\theta_1 \wedge \theta_2)^{\mathbf{f}} & = & \theta_1^{\mathbf{f}} \vee \theta_2^{\mathbf{f}} \\
(\theta_1 \vee \theta_2)^{\mathbf{t}} & = & \theta_1^{\mathbf{t}} \vee \theta_2^{\mathbf{t}} & (\theta_1 \vee \theta_2)^{\mathbf{f}} & = & \theta_1^{\mathbf{f}} \wedge \theta_2^{\mathbf{f}} \\
(\neg\theta)^{\mathbf{t}} & = & \theta^{\mathbf{f}} & (\neg\theta)^{\mathbf{f}} & = & \theta^{\mathbf{t}} \\
(t \texttt{ IS NULL})^{\mathbf{t}} & = & t \texttt{ IS NULL} & (t \texttt{ IS NULL})^{\mathbf{f}} & = & t \texttt{ IS NOT NULL} \\
(\bar{t} \texttt{ IN } Q)^{\mathbf{t}} & = & \bar{t} \texttt{ IN } Q' & \big((t_1,\ldots,t_n) \texttt{ IN } Q\big)^{\mathbf{f}} & = & \texttt{NOT EXISTS } \big(\texttt{SELECT } \star \texttt{ FROM } Q' \texttt{ AS } N(A_1,\ldots,A_n) \texttt{ WHERE} \\
 & & & & & \quad (t_1 \texttt{ IS NULL OR } A_1 \texttt{ IS NULL OR } t_1 = N.A_1) \texttt{ AND } \cdots \\
 & & & & & \quad \cdots \texttt{ AND } (t_n \texttt{ IS NULL OR } A_n \texttt{ IS NULL OR } t_n = N.A_n)\big)
\end{array}
$$

Figure 10: Translations of conditions for the $Q \mapsto Q'$ translation

This extension of RA to bags is equivalent to comprehension-based languages that share many features with SQL [16, 23]. While such languages were used to study the expressive power of SQL, they are not SQL; rather, they are theoretical reconstructions of it that allow one to prove equivalence results but differ significantly from the real language, in particular w.r.t. handling nulls and variable bindings.

A different line of work attempted to provide a formal semantics of SQL directly, but all such attempts have fallen short of the real language. An early paper [29] looked only at set semantics, and the more recent and rigorous formalization [8, 9] – 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 [24, 37] but they too restricted the language significantly: for example, [24] works essentially with RA, rather than SQL, under set semantics, while [37] disallows nested subqueries in both **FROM** and **WHERE** and uses list semantics.

We remark that none of the above mentioned works made any effort to justify, neither experimentally nor by any other means, the semantics they proposed. So there is no evidence that these semantics reflect the real behavior of SQL, even if we take into account the specific restrictions they imposed.

On the logic side, to the best of our knowledge, the only approach to combine 3VL with Boolean logic along the lines of SQL is external Bochvar logic, which has a special connective conflating false and unknown. However, this has mainly been the subject of study of philosophical logic [25, 34] and therefore restricted to the propositional case. A two-valued logic for nulls based on collapsing **u** and **f** was also considered in [5] for a fragment of SQL, but no comparison of its expressiveness with the standard semantics was made.

8. 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 3VL is not required to achieve the full expressiveness of this fragment of SQL, and somewhat surprisingly the familiar two-valued logic does the job.

From a practical point of view, our formal semantics could be a useful tool for both users and implementers in understanding the behavior of SQL queries. It is much more concise than the natural language specification of the Standard, as well as being very easy to implement and modify. In fact, we advocate that our formal semantics (or a variant of it, if necessary) should be an integral part of the Standard and serve as the basis for a reference implementation endorsed by ISO. The compliance of a DBMS with the Standard could then be verified against this implementation, for example by means of an appropriate suite of test cases like the Technology Compatibility Kit developed by the openCypher initiative in the context of graph databases [30].

Future work. A first natural direction for future work is to extend the formal semantics, and its experimental validation, to include more features of the language, especially aggregation and grouping, but also capabilities that go beyond queries, such as schema definition, constraints and updates.

Some of the restricted SQL semantics [9, 24, 37] were defined for verifying the correctness of SQL optimization rules. They could only do so under the restrictions they imposed; thus it would be interesting to see what such verification techniques would yield without restrictions on the language.

The equivalence between two-valued and three-valued semantics of SQL raises some interesting questions too: would two-valued queries be natural for a common user to write? We do believe that people tend to think in terms of true and false only, rather than three truth values. But of course this conjecture should be confirmed (or disproved) by a proper usability study.

Yet another line for future work is the extension of recent attempts [17] to restore correctness of SQL query evaluation with incomplete data. Due to the lack of a formal semantics for query evaluation with SQL nulls, so far this has only been done for databases with marked nulls. Now we have the formal tools to extend the notions of certainty and possibility to handle SQL's nulls.

Acknowledgements. The authors would like to thank the anonymous referees for their comments. Work partially supported by EPSRC grants N023056 and M025268.

9. REFERENCES

[1] H. Abelson et al. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] J. Albert. Algebraic properties of bag data types. In *17th International Conference on Very Large Data Bases*, pages 211–219, 1991.

[4] T. Arvin. *Comparison of different SQL implementations*. `http://troels.arvin.dk/db/rdbms`, 2017.

[5] L. E. Bertossi and L. Bravo. Consistency and trust in peer data exchange systems. *TPLP*, 17(2):148–204, 2017.

[6] L. Bolk and P. Borowik. *Many-Valued Logics: Theoretical Foundations*. Springer, 1992.

[7] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. Software Eng.*, 11(4):324–345, 1985.

[8] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[9] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: Proving query rewrites with univalent SQL semantics. *PLDI*, 2017.

[10] J. Claußen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 12(2):238–260, 2000.

[11] E. F. Codd. A database sublanguage founded on the relational calculus. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971*, pages 35–68, 1971.

[12] F. Coelho. DataFiller – generate random data from database schema. `https://www.cri.ensmp.fr/people/coelho/datafiller.html`.

[13] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.

[14] C. Ellison. *A Formal Semantics of C with Applications*. PhD thesis, UIUC, 428pp, 2012.

[15] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems - The Complete Book*. Pearson Education, 2009.

[16] S. Grumbach and T. Milo. Towards tractable algebras for bags. *J. Comput. Syst. Sci.*, 52(3):570–588, 1996.

[17] P. Guagliardo and L. Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *Proceedings of the 35th ACM Symposium on Principles of Database Systems*, pages 211–223, 2016.

[18] C. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[19] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Computer Science Logic, 6th Workshop, CSL '92*, pages 274–308, 1992.

[20] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[21] International Organization for Standardization. *ISO/IEC 9075:2016: Information technology – Database languages – SQL*, 2016.

[22] K. E. Kline and D. Kline. *SQL in a Nutshell*. O'Reilly & Associates, Inc., 2001.

[23] L. Libkin and L. Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences*, 55(2):241–272, 1997.

[24] J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–248, 2010.

[25] G. Malinowski. Many-valued logic and its philosophy. In *Handbook of the History of Logic*, pages 13–94. Elsevier, 2007.

[26] R. Milner and M. Tofte. *Commentary on standard ML*. MIT Press, 1991.

[27] R. Milner, M. Tofte, and R. Harper. *Definition of standard ML*. MIT Press, 1990.

[28] J. C. Mitchell. *Concepts in programming languages*. Cambridge University Press, 2003.

[29] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3):513–534, 1991.

[30] Neo Technlogy Inc. *openCypher project*, `http://www.opencypher.org/`.

[31] M. Norrish. *C formalised in HOL*. Univ. Cambridge Techreport UCAM-CL-TR- 453, 150pp., 1998.

[32] N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, NTUA, 253pp, 1998.

[33] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[34] N. Rescher. *Topics in Philosophical Logic*. Reidel, 1969.

[35] Transaction Processing Performance Council. *TPC Benchmark™ H Standard Specification*, 2014. Revision 2.17.1.

[36] J. Van den Bussche and S. Vansummeren. Translating SQL into the relational algebra. Course notes, Hasselt University and Université Libre de Bruxelles, 2009.

[37] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 425–446, 2010.