# How standard is the SQL Standard?

# How “Standard” is the SQL Standard?

Paolo Guagliardo and Leonid Libkin

University of Edinburgh

## 1 Introduction

The Structured Query Language (SQL) has been an international standard for more than three decades, and it is now supported in all major RDBMSs. While it is well known that database vendors typically implement custom extensions of SQL that are not part of the ISO Standard, it is reasonable to expect statements that use only “standard” features of the language to behave the same regardless of which system they are executed on; after all, this is the whole point of having a standard. However, the reality is that even simple statements may compile in some systems and not in others, or produce different results.

To illustrate how puzzling and counterintuitive things may get, let us consider a base table `R` with one column named `A`, and the query

$Q$ = **`SELECT`** `R.A, R.A` **`FROM`** `R`

In all of the most popular RDBMSs on the market today – Oracle, SQL Server, DB2, MySQL and PostgreSQL – $Q$ will give the same answer. But now consider

$Q'$ = **`SELECT`** `*` **`FROM`** `(` $Q$ `) R1`

where the expression for $Q$ is plugged in as a subquery. All of the above systems, except PostgreSQL, will report a compilation error, even though $Q'$ simply asks to print the result of the query $Q$ that, on its own, runs without problems. PostgreSQL, on the other hand, compiles $Q'$ and outputs the same answer produced by $Q$. Then it gets even more interesting. Consider

$Q''$ = **`SELECT`** `*` **`FROM`** `R` **`WHERE EXISTS`** `(` $Q'$ `)`

where we plug in the expression for $Q'$ that did not compile on all systems except PostgreSQL. Now this query suddenly works, even on those systems that did not compile $Q'$, with the sole exception of MySQL that keeps reporting error.

Understanding what is going on here is not straightforward. Part of the issue is due to the fact that the only official specification of SQL is the ISO Standard, which is a massive document entirely written in natural language. Aside from it, there are no other tools, such as a reference implementation or a standardized set of test cases, to assess the level of compliance of an RDBMS w.r.t. the Standard.

In this short paper, we describe some of the discrepancies that exist between different SQL implementations, and we discuss our recent and ongoing efforts [1] towards a formal semantics of the SQL language, arguing how this could be used to improve the current state of affairs.

## 2 Compliance with the Standard

The simplest discrepancies between SQL implementations are due to operators or constructs missing from the language, or having different syntactic forms than in the Standard. While still important for portability, such kind of differences are typically well documented and relatively easy to deal with.

More problematic are the cases in which a feature is supported by a system, but it does not behave as expected. Examples are integer division in MySQL and Oracle, which produces a decimal number in place of an integer, and division by zero in MySQL, which returns **NULL** instead of throwing a runtime error. Sometimes, this can be partly mitigated by modifying the system's configuration; for instance, MySQL has a "strict mode" that forces a runtime error for division by zero in data-modification statements but has no effect in queries, and an "ANSI mode" that restores some sanity by enforcing standard quotation marks, pipes for string concatenation, and fully-specified **GROUP BY** clauses. On the other hand, this also means that the behavior w.r.t. the Standard depends on the particular mode the system runs in, which further complicates things.

Some behaviors are deeply built into individual systems as a result of poor or fundamentally flawed design choices, which cannot be changed via configuration parameters. One instance of this is Oracle's astonishing choice of implementing **NULL** as the empty string. For example, in Oracle, the query

```
SELECT * FROM R WHERE 'a' <> ''
```

produces an empty table even if `R` is non-empty, because `''` is **NULL** and every comparison involving **NULL** evaluates to the truth value *unknown*, so no rows are returned. In a world where text processing is very common and SQL statements are often machine-generated, it is not hard to imagine scenarios where this could have devastating effects: e.g., a large, complex transaction may be aborted simply because it attempted to insert the result of a string operation – which happened to produce the empty string – into a column declared as **NOT NULL**.

A similar shortsighted implementation choice can be found in MySQL, which construes **TRUE** and **FALSE** as the integers 1 and 0, respectively. But the world's most popular open source database truly surpasses itself with its type checking system and implicit casting rules, which result in meaningless queries like

```
SELECT * FROM R WHERE TRUE + 4 - '5'
```

to successfully compile and execute: **TRUE** is 1, the string `'5'` is implicitly converted to the integer 5, hence the **WHERE** condition above evaluates to 0, which is **FALSE**, and so no rows are returned.

There are then cases when RDBMSs deviate from the Standard for sensible reasons. To illustrate this point, let us go back to the queries $Q'$ and $Q''$ given in the introduction. According to the Standard, the behavior of **SELECT** * depends on the context in which it occurs: under **EXISTS**, the star amounts to having an arbitrary constant in its place, while everywhere else it is the same as explicitly listing all attributes. This means that the semantics of SQL is non-compositional (the same query can behave differently in different contexts) and it is why DB2, Oracle and SQL Server – which follow the Standard in this respect – report error for $Q'$ but not for $Q''$. Indeed, replacing $\star$ in $Q'$ with an explicit list of attributes results in a reference to `A` that is ambiguous, because there are two columns with that name in `R1`. This does not happen in $Q''$, where $\star$ is replaced by a constant.

MySQL and PostgreSQL take different routes that deviate from the Standard but restore compositionality: the former always turns the star into an explicit list of attributes, even under **EXISTS**; the latter interprets it as "return all columns without referencing them by name" (which in our opinion is the most reasonable choice). This is why MySQL reports error for both $Q'$ and $Q''$, while PostgreSQL executes them successfully.

## 3 Formal Semantics of SQL Queries

The SQL Standard is written in natural language, which is inherently ambiguous, and to think that it may be the source of misunderstandings among developers, implementers, as well as database researchers, is not implausible. Moreover, from a practical point of view, a natural language specification is harder to implement and maintain, and it does not lend itself to 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 several attempts at providing one have been made in the past (see [1] for relevant references). However, all of the approaches found in the literature make at least one of the following unrealistic assumptions: set semantics (that is, rows in tables do not repeat) and absence of null values. Recently, we proposed a formal semantics of a significant fragment of SQL, by taking into account bag semantics and nulls, and working with the real language, rather than a theoretical reconstruction of it. To illustrate the salient points of our formal semantics, we refer again to the behavior of **SELECT** $\star$ discussed in the previous sections.

In SQL, queries $Q$ and conditions $\theta$ are defined by mutual recursion: queries have conditions in the **WHERE** clause, and a condition may involve a query within **EXISTS** or **IN**. The semantic function $[\![\cdot]\!]$ we define takes different inputs depending on the syntactic construct under consideration: for conditions, the inputs are a database $D$ and an environment $\eta$ that binds attribute references to actual values; for queries we additionally need a Boolean flag $x$ that indicates whether we are within an **EXISTS** predicate. Then, the behavior of $\star$ in **SELECT** is given by:

$$\left[\!\!\left[\begin{array}{r}\textbf{SELECT}\ \star\\ \textbf{FROM}\ \tau\\ \textbf{WHERE}\ \theta\end{array}\right]\!\!\right]_{D,\eta,x} = \begin{cases} [\![\textbf{SELECT}\ c\ \textbf{AS}\ N\ \textbf{FROM}\ \tau\ \textbf{WHERE}\ \theta]\!]_{D,\eta,x} & \text{if } x = 1\\ \\ [\![\textbf{SELECT}\ \ell(\tau)\ \textbf{FROM}\ \tau\ \textbf{WHERE}\ \theta]\!]_{D,\eta,x} & \text{if } x = 0\end{cases}$$

where $c$ is an arbitrary constant and $N$ is an arbitrary name, while $\ell$ is a labeling function producing the list of attributes in (the Cartesian product of) the tables in the **FROM** clause. The Boolean flag $x$ is set to 1 in **EXISTS** conditions as follows:

$$[\![\textbf{EXISTS}\ Q]\!]_{D,\eta} = \begin{cases}\mathbf{t} & \text{if } [\![Q]\!]_{D,\eta,1} \neq \varnothing\\ \mathbf{f} & \text{if } [\![Q]\!]_{D,\eta,1} = \varnothing\end{cases}$$

and it is reset to 0 in other rules of the semantics (see [1] for details).

Our formal semantics can also be easily adapted from one system to another. For example, in PostgreSQL and MySQL, the semantic function for queries will be defined only w.r.t. a database $D$ and an environment $\eta$, while the flag $x$ is no longer needed, because in these two systems the behavior of $\star$ is independent of the context. Then, the semantics of **SELECT** $\star$ queries in MySQL is given by

$$[\![\texttt{SELECT}\ \star\ \texttt{FROM}\ \tau\ \texttt{WHERE}\ \theta]\!]_{D,\eta} = [\![\texttt{SELECT}\ \ell(\tau)\ \texttt{FROM}\ \tau\ \texttt{WHERE}\ \theta]\!]_{D,\eta}$$

while for PostgreSQL we have

$$[\![\texttt{SELECT}\ \star\ \texttt{FROM}\ \tau\ \texttt{WHERE}\ \theta]\!]_{D,\eta} = [\![\texttt{FROM}\ \tau\ \texttt{WHERE}\ \theta]\!]_{D,\eta}$$

Observe that the absence of **SELECT** in the r.h.s. of the above equation is not a typo; refer to [1] for details.

The semantics was experimentally validated with PostgreSQL and Oracle on a large number of randomly generated queries. Its immediate applications were theoretical: providing the first formal proof that basic SQL queries and relational algebra have the same expressive power, and showing that – contrary to common belief – three-valued logic is not really necessary for handling nulls.

## 4 Outlook

The goal of the SQL Standard is to guarantee interoperability between RDBMSs, but the discrepancies existing in current implementations are significant enough to severely affect the portability of SQL code. The problems we discussed here are just the tip of the iceberg, as we have not even touched on grouping, aggregation, constraints, transactions, data definition statements, recursive queries, common table expressions, updates, and many other features of the language.

Our ambitious goal is to make the Standard more clear, formal and accessible, in the interest of researchers and practitioners alike. The semantics we presented in [1], and briefly outlined here, is a first important step towards this goal and we expect it to lead to a cluster of tools aimed at assessing the level of compliance of an RDBMS w.r.t. the Standard in a transparent way. Examples of such tools are a reference implementation and a technology compatibility kit for SQL.

This work has already yielded very positive outcomes. First, it has led to our cooperation with Neo4j, a world-leading company in field of graph databases, to formalize the semantics of the graph query language Cypher [2]. Second, we are currently negotiating with the standardization committee ISO/IEC JTC1/SC32 the possibility of including a formal specification of SQL to the Standard, in the form of a (non-normative) technical report.

## References

1. Guagliardo, P., Libkin, L.: A formal semantics of SQL queries, its validation, and applications. PVLDB **11**(1) (2017) 27–39
2. Francis, N., et al.: Cypher: An evolving query language for property graphs. In: SIGMOD'18 – Industrial Track, ACM (2018) To appear.