# Proving the Safety of SQL Queries

Stefan Brass Christian Goldberg
*Martin-Luther-Universität Halle-Wittenberg, D-06099 Halle (Saale), Germany*
*(brass|goldberg)@informatik.uni-halle.de, Phone: +49 345 55 24740, Fax: +49 345 55 27333*

## Abstract

*Many programs need to access data in a relational database. This is usually done by means of queries written in SQL. Although the language SQL is declarative, certain runtime errors are possible. Since the occurrence of these errors depend on the data, they are not easily found during testing. The question whether a query is safe can be reduced to a consistency check. It is well known that consistency is in general undecidable, and that this applies also to SQL queries. However, in this paper, we propose a consistency check that can handle a surprisingly large subset of SQL (it uses Skolemization with sorted Skolem functions, and a few other tricks). This consistency check is also the basis for generating other semantic warnings. Furthermore, it can be used to generate test data for SQL queries.*

**Topics:** *Model Construction, Information Systems, Static Analysis, Software Quality.*

**Keywords:** *SQL, Runtime Errors, Consistency.*

## 1. Introduction

Probably a large percentage of the software that is developed today uses data stored in a relational database. The database is normally accessed with statements in the language SQL, especially queries. Although SQL is a declarative language, there are situations in which errors might occur at runtime that depend on the data. A typical case is "`SELECT-INTO`" in Embedded SQL (including, e.g., SQLJ):

```
SELECT S.EMAIL
INTO   :E
FROM   STUDENTS S
WHERE  S.FNAME = :FN
AND    S.LNAME = :LN
```

This assumes that there is a table `STUDENTS` with columns `SID` (unique student number, primary key), `FNAME` (first name), `LNAME` (last name), and `EMAIL`. If it should ever happen that the table contains two rows that match the condition (i.e. two students with the same first and last name), a runtime error (exception) occurs when the query is executed. Since this error is probably not anticipated, it will not be handled well, and the error message shown to the user will not be helpful.

If we want to develop quality software, we should make sure that such errors can never occur. In the example, if "`FNAME, LNAME`" is declared as secondary key of the table, the above query is safe. Of course, then inserting two students with the same first and last name is impossible (Since this does occur in practice, another means of identification must be used.)

Note that if a programmer wrongly assumes that first name and last name are unique, at least manually developed test data will of course satisfy this condition, and the error is not detected during testing. All assumptions about the data must be enforced using constraints.

It is true that some database interfaces (e.g., JDBC) do not support `SELECT-INTO`: There, one always gets a result set, so this error cannot occur. But the programmer would normally not use a loop if he or she assumes that the result is a singleton. A warning would certainly be useful if there can be more than one tuple in the result set, but the program contains only code for accessing the first element. See also Section 2 for runtime errors in SQL that do not depend on the interface.

But let us return to the `SELECT-INTO` example. The question whether the exception is possible can easily be reduced to a consistency test. In the example, we must check whether the query

```
SELECT *
FROM   STUDENTS S1, STUDENTS S2
WHERE  S1.FNAME = :FN
AND    S1.LNAME = :LN
AND    S2.FNAME = :FN
AND    S2.LNAME = :LN
AND    S1.SID <> S2.SID
```

can ever have a non-empty result. If it can, the database state in which the above query produces an answer provides an example in which the original query causes the

exception. If, however, the above query has always an empty result (i.e. its condition is inconsistent), the original query is safe.

It is well known that the consistency of formulas is undecidable in first-order logic, and that this applies also to database queries. For example, one can write a query (without datatype operations) that checks whether the database contains a solution to a Post's correspondence problem, see [1], Section 6.3.

Nevertheless, there are many solutions for subsets of SQL. For conjunctions of comparisons (with dense domains), it seems that the method of [8] is the state of the art. In this paper (Section 3), we show how it can be extended to handle many cases of subqueries. We also treat the problem of null values (SQL uses a three-valued logic for them). In this way, the consistency of a surprisingly large subset of SQL can be decided. Our method is based on the well-known idea of Skolemization, but further tricks are needed to keep the Herbrand universe finite as far as possible.

Runtime errors in SQL are briefly discussed in Section 2. Besides these safety problems, there are many more cases, where warnings should be generated for SQL queries that are strange and probably not intended, or at least sub-optimal, see [2]. The consistency check presented in this paper is an important subroutine for many of SQL quality tests. We are currently developing a semantic checker for SQL (called SQLLint), see

`http://dbs.informatik.uni-halle.de/sqllint/`

The current version of the prototype contains the consistency check described in this paper.

Finally, for testing database applications, interesting test data must be generated. It seems that this problem has not been discussed often in the literature, where [13] is a noteable exception. Our consistency check can also be used to generate test data, and in contrast to [13] it is not necessary to define beforehand how many rows are needed in each table.

## 2. Possible Runtime Errors in SQL

As explained in the introduction, if the `INTO`-clause is used for storing the query result in program variables, the query must never return more than one row. Suppose the given query is

```
SELECT t_1, ..., t_k
INTO   v_1, ..., v_k
FROM   R_1 X_1, ..., R_n X_n
WHERE  φ
```

In order to make sure that there are never two solutions, we duplicate the tuple variables and check the following query for consistency. If it is consistent (including the constraints as explained in Section 3.3), a runtime error can occur, and the constructed model gives an example:

```
SELECT *
FROM   R_1 X_1, ..., R_n X_n,
       R_1 X'_1, ..., R_n X'_n
WHERE  φ AND φ'
AND    (X_1 ≠ X'_1 OR ··· OR X_n ≠ X'_n)
```

The formula $\varphi'$ results from $\varphi$ by replacing each $X_i$ by $X'_i$. We use $X_i \neq X'_i$ as an abbreviation for requiring that the primary key values of the two tuple variables are different (we assume that primary keys are always `NOT NULL`). If one of the relations $R_i$ has no declared key, it is always possible that there are several solutions (if the condition $\varphi$ is consistent).

If the given query uses "`SELECT DISTINCT`", one needs to add a test that the result tuples differ:

```
(t_1 ≠ t'_1 OR ··· OR t_k ≠ t'_k
 OR t_1 IS NULL AND t'_1 IS NOT NULL
 OR t'_1 IS NULL AND t_1 IS NOT NULL
 ...
 OR t_k IS NULL AND t'_k IS NOT NULL
 OR t'_k IS NULL AND t_k IS NOT NULL)
```

The same problem can occur with conditions of the form `A = (SELECT ...)`: Whenever a subquery is used as scalar expression, it must not return multiple rows. If the subquery is non-correlated (i.e. does not access tuple variables from the outer query), we can use exactly the same test as above. If the query is correlated, it might not be completely clear what knowledge from the outer condition should be used (as usual for runtime errors, evaluation order becomes important here). In order to be safe, we propose to ignore the outer condition. Let the subquery have the form

```
SELECT t_1, ..., t_k
FROM   R_1 X_1, ..., R_n X_n
WHERE  φ
```

If it accesses the tuple variables $S_1\,Y_1, \ldots, S_m\,Y_m$ from the outer query, we would require that the following query is inconsistent (after adding the constraints):

```
SELECT *
FROM   R_1 X_1, ..., R_n X_n,
       R_1 X'_1, ..., R_n X'_n,
       S_1 Y_1, ..., S_m Y_m
WHERE  φ AND φ'
AND    (X_1 ≠ X'_1 OR ··· OR X_n ≠ X'_n)
```

Further runtime errors, which can be handled with similar methods, are:

1. Using `SELECT INTO` or `FETCH` without an indicator variable when the corresponding result column can be null.
2. Possibly type conversion errors from strings to numbers when the string does not have a numeric format.
3. Datatype operators can have the usual problems (e.g., division by zero). If the critical operation appears in the `SELECT`-clause, one can assume that the `WHERE`-condition is satisfied. However, if it appears in the `WHERE`-clause, only the satisfaction of the constraints can be safely assumed, since the optimizer is free to choose any evaluation sequence. Note that an approach relying on testing is impossible in this case since the evaluation sequence may change in future versions of the DBMS.

In this paper, only queries are considered. Of course, updates can also generate runtime errors if they violate a constraint. However, in this case the analysis is insofar more difficult, as the statement in itself is usually not safe, and one must consider the surrounding program code (i.e. checks done before the update is executed).

## 3. Inconsistent Conditions

In this section, we present an algorithm for detecting inconsistent conditions in SQL queries. Since the problem is in general undecidable, we can handle only a subset of all queries. However, our algorithm is reasonably powerful and can decide the consistency of surprisingly many queries.

To be precise, consistency in databases means that there is a finite model, i.e. a relational database state (sometimes called a database instance), such that the query result is not empty.

In this paper, we assume that the given SQL query contains no datatype operations, i.e. all atomic formulas are of the form $t_1\ \theta\ t_2$ where $\theta$ is a comparison operator (=, <>, <, <=, >, >=), and $t_1, t_2$ are attributes (possibly qualified with a tuple variable) or constants (literals). Null values and `IS NULL` are treated in Section 3.5, before that, they are excluded. Aggregations are not treated in this paper, they are subject of our future research.

### 3.1. Conditions Without Subqueries

If the query contains no subqueries, the consistency can be decided with methods known in the literature, especially the algorithms of Guo, Sun and Weiss [8].

The condition then consists of the above atomic formulas connected with `AND`, `OR`, `NOT`. We first push negation down to the atomic formulas, where it simply "turns around" the comparison operator. In this way, `NOT` is eliminated from the formula. Then, we translate the formula in disjunctive normal form: $\varphi_1 \vee \cdots \vee \varphi_n$ is consistent iff at least one of the $\varphi_i$ is consistent.

Now a conjunction of the above atomic formulas can be tested for satisfiability with the method of [8]. They basically create a directed graph in which nodes are labeled with "Tuplevariable.Attribute" (maybe a representative for an equivalence class with respect to $=$) and edges are labeled with $<$ or $\leq$. Then they compute an interval of possible values for each node. Note that SQL data types like `NUMERIC(1)` also restrict the interval of possible values.

Unfortunately, if there is only a finite number of values that can be assigned to nodes, inequality conditions ($t_1$ <> $t_2$) between the nodes become important and can encode graph-coloring problems. Therefore, we cannot expect an efficient algorithm if there are many <>-conditions. Otherwise, the method of [8] is fast. (However, the DNF transformation that we apply before [8] can lead to an exponential increase in size.)

### 3.2. Subqueries

For simplicity, we treat only `EXISTS` subqueries. Other kinds of subqueries (`IN`, >=`ALL`, etc.) can be reduced to the `EXISTS` case. For example, Oracle performs such a query rewriting before the optimizer works on the query. We also assume without loss of generality that all occurring tuple variables have different names.

Let us first classify variables as existential or universal, depending on how they would be quantified ($\exists$ or $\forall$) in tuple relational calculus if the query were converted to prenex normal form:

**Definition 1** *Given a query $Q$, let us call a tuple variable in $Q$ existential if it is declared in the* `FROM`*-clause of the main query or of a subquery that is nested inside an even number (including 0) of* `NOT`*s, and universal otherwise.*

**Example 1** *In the examples, we use a database schema that contains the grades students received on homework exercises. It consists of three relations:*

- STUDENTS(SID, FNAME, LNAME, EMAIL)
- EXERCISES(ENO, TOPIC)
- GRADES(SID→STUDENTS, ENO→EXERCISES, POINTS)

*The following SQL query lists students that received full credit (10 points) for all exercises:*

```
SELECT S.FNAME, S.LNAME
FROM   STUDENTS S
WHERE  NOT EXISTS
       (SELECT *
        FROM   EXERCISES E
        WHERE  NOT EXISTS
               (SELECT *
                FROM   GRADES G
                WHERE  G.SID = S.SID
                AND    G.ENO = E.ENO
                AND    G.POINTS = 10))
```

*S and G are existential tuple variables, and E is a universal tuple variable. Our algorithm will find a model in which the table EXERCISES is empty. Since such models are not very typical, one could automatically add conditions to the query that enforce the existence of at least one tuple in each relation. In the example, it would suffice to add (at the end of the WHERE-clause):*

```
AND EXISTS (SELECT *
            FROM   EXERCISES E1)
```

*However, for the sake of the example, let us suppose that we want to make sure that there are at least two exercises with specific numbers, therefore we add instead*

```
AND EXISTS (SELECT *
            FROM   EXERCISES E1
            WHERE  E1.ENO = 1)
AND EXISTS (SELECT *
            FROM   EXERCISES E2
            WHERE  E2.ENO = 2)
```

*Now E1 and E2 are further existential tuple variables.* □

In automated theorem proving (see, e.g., [4]), it is a well-known technique to eliminate existential quantifiers by introducing Skolem constants and Skolem functions. This simply means that a name is given to the tuples that are required to exist. For tuple variables that are not contained in the scope of a universal quantifier (such as S, E1, and E2 in the example), a single tuple is sufficient in the database state. However, for an existential tuple variable like G that is declared within the scope of a universal tuple variable (E) a different tuple might be required for every value for E. Therefore, a function $f_{\mathtt{G}}$ is introduced that takes a value for E as a parameter and returns a value for G. Such a function is called a Skolem function. Formally, there are also Skolem functions $f_{\mathtt{S}}$, $f_{\mathtt{E1}}$, and $f_{\mathtt{E2}}$, but these functions have no parameters (they are Skolem constants).

Let us make precise what parameters $Y$ the Skolem function $f_X$ for a tuple variable $X$ must have:

**Definition 2** *An existential tuple variable $X$ depends on a universal tuple variable $Y$ iff*

1. *the declaration of $X$ appears inside the scope of $Y$, and*
2. *$Y$ appears in the subquery in which $X$ is declared (including possibly nested subqueries).*

The second part of the condition is not really required, but it reduces the number of parameters which will help us to handle more queries.

In contrast to the classical case of automated theorem proving, we use a "sorted" logic: Each tuple variable can range only over a specific relation. Therefore our Skolem functions have parameter and result types. For example, the function $f_{\mathtt{G}}$ in the example assumes that a tuple from the relation EXERCISES is given, and returns a tuple from the relation GRADES.

**Definition 3** *Given a query $Q$, a set of sorted Skolem constants and functions $\mathcal{S}_Q$ is constructed as follows: For each existential tuple variable $X$ ranging over relation $R$, a Skolem constant/function $f_X$ of sort $R$ is introduced. Let $Y_1, \ldots, Y_n$ be all universal tuple variables, on which $X$ depends, and let $Y_i$ range over relation $S_i$. Then $f_X$ has $n$ parameters of sort $S_1, \ldots, S_n$.*

In the example, there are three Skolem constants and one Skolem function:

- $f_{\mathtt{S}}$: STUDENTS,
- $f_{\mathtt{G}}$: EXERCISES → GRADES,
- $f_{\mathtt{E1}}$: EXERCISES,
- $f_{\mathtt{E2}}$: EXERCISES.

**Definition 4** *Given a query $Q$, and a relation $R$, let $\mathcal{T}_Q(R)$ be the set of all terms of sort $R$ that can be built from the constants and function symbols in $\mathcal{S}_Q$ respecting the sorts. Let $\mathcal{T}_Q$ be the union of the $\mathcal{T}_Q(R)$ for all relation symbols $R$ appearing in $Q$.*

$\mathcal{T}_Q$ is a kind of Herbrand universe. In Example 1,

- $\mathcal{T}_Q(\mathtt{STUDENTS}) = \{f_{\mathtt{S}}\}$,
- $\mathcal{T}_Q(\mathtt{EXERCISES}) = \{f_{\mathtt{E1}}, f_{\mathtt{E2}}\}$,

- $\mathcal{T}_Q(\texttt{GRADES}) = \{f_{\texttt{G}}(f_{\texttt{E1}}),\ f_{\texttt{G}}(f_{\texttt{E2}})\}$.

Of course, in general it is possible that infinitely many terms can be constructed. If we really cannot predict how large a model (database state) must be, our method is not applicable. However, we show in Section 3.4 how we can often prove that different elements of the Herbrand universe must be the same tuple. But first note that an infinite Herbrand universe requires at least a nested `NOT EXISTS` subquery (otherwise only Skolem constants are produced, no real functions). The case with only a single level of `NOT EXISTS` subqueries corresponds to the quantifier prefix $\exists^*\forall^*$, for which it is well known that the satisfiability of first order logic is decidable (this was proven 1928 by Bernays and Schönfinkel). However, as the example shows, our method can sometimes handle subquery structures that go beyond the $\exists^*\forall^*$ quantifier prefix: Since we use a sorted logic, the set of Skolem terms is not necessarily infinite even when there are proper function symbols.

Once we know how many tuples each relation must have, we can easily reduce the general case (with subqueries) to a consistency test for a simple formula as treated in [8] (see Section 3.1):

**Definition 5** *Let a query $Q$ be given, and let $\mathcal{T}_Q$ be finite. The flat form of the* `WHERE`*-clause is constructed as follows:*

1. *Replace each tuple variable $X$ of the main query by the corresponding Skolem constant $f_X$.*
2. *Next, treat subqueries nested inside an even number of* `NOT`*: Replace the subquery*

   `EXISTS (SELECT ...` `FROM` $R_1\,X_1$`, ...,` $R_n\,X_n$ `WHERE` $\varphi$`)`

   *by*

   `(`$\sigma(\varphi)$ `AND NE_`$R_1$ `AND ... AND NE_`$R_n$`)`

   *where the substitution $\sigma$ replaces each existential tuple variable $X_i$ by $f_{X_i}(Y_{i,1},\ldots,Y_{i,m_i})$. Here $Y_{i,1},\ldots,Y_{i,m_i}$ are all universal tuple variables on which $X_i$ depends.*
3. *Finally treat subqueries that appear within an odd number of negations as follows: Replace the subquery*

   `EXISTS (SELECT ...` `FROM` $R_1\,X_1$`, ...,` $R_n\,X_n$ `WHERE` $\varphi$`)`

   *by*

   `((`$\sigma_1(\varphi)$ `OR ... OR` $\sigma_k(\varphi)$`)` `AND NE_`$R_1$ `AND ... AND NE_`$R_n$`)`

   *where $\sigma_i$ are all substitutions that map the variables $X_j$ to a term in $\mathcal{T}_Q(R_j)$. Note that $k = 0$ is possible, in which case the empty disjunction can be written* `1=0` *(falsity).*

The predicates `NE_`$R_i$ ("$R_i$ is nonempty") are needed only in very exceptional cases, see Example 2 below. A simple analysis of the example query shows that they are not required (no existential tuple variables are introduced in disjunctive context). Therefore, let us compute the flat form of the example without the `NE_`$R_i$ predicates. We then first substitute `S` by $f_{\texttt{S}}$, `E1` by $f_{\texttt{E1}}$, `E2` by $f_{\texttt{E2}}$, and `G` by $f_{\texttt{G}}$(`E`). Since `E` is of type `EXERCISES`, and $f_{\texttt{E1}}$ and $f_{\texttt{E2}}$ are the only elements of $\mathcal{T}_Q$(`EXERCISES`), the disjunction consists of two cases with `E` replaced by $f_{\texttt{E1}}$ and by $f_{\texttt{E2}}$, respectively:

```
SELECT fS.FNAME, fS.LNAME
FROM   STUDENTS S
       -- replaced by fS
WHERE
NOT(EXISTS
    (SELECT *
     FROM   EXERCISES E
            -- version for fE1
     WHERE  NOT EXISTS(
            SELECT *
            FROM   GRADES G -- fG(fE1)
            WHERE  fG(fE1).SID = fS.SID
            AND    fG(fE1).ENO = fE1.ENO
            AND    fG(fE1).POINTS = 10))
    OR
    EXISTS
    (SELECT *
     FROM   EXERCISES E
            -- version for fE2
     WHERE  NOT EXISTS(
            SELECT *
            FROM   GRADES G -- fG(fE2)
            WHERE  fG(fE2).SID = fS.SID
            AND    fG(fE2).ENO = fE2.ENO
            AND    fG(fE2).POINTS = 10))
)
AND   EXISTS(SELECT *
             FROM   EXERCISES E1
                    -- replaced by fE1
             WHERE  fE1.ENO = 1)
AND   EXISTS(SELECT *
             FROM   EXERCISES E2
                    -- replaced by fE2
             WHERE  fE2.ENO = 2)
```

Formally, we would directly construct the flat form of the above query, in which the SELECT/FROM clauses and the EXISTS/WHERE keywords are removed:

```
NOT(     NOT(     f_G(f_E1).SID = f_S.SID
              AND f_G(f_E1).ENO = f_E1.ENO
              AND f_G(f_E1).POINTS = 10)
      OR NOT(     f_G(f_E2).SID = f_S.SID
              AND f_G(f_E2).ENO = f_E2.ENO
              AND f_G(f_E2).POINTS = 10))
AND   (f_E1.ENO = 1)
AND   (f_E2.ENO = 2)
```

This is logically equivalent to

```
          f_G(f_E1).SID = f_S.SID
    AND   f_G(f_E1).ENO = f_E1.ENO
    AND   f_G(f_E1).POINTS = 10
    AND   f_G(f_E2).SID = f_S.SID
    AND   f_G(f_E2).ENO = f_E2.ENO
    AND   f_G(f_E2).POINTS = 10
    AND   f_E1.ENO = 1
    AND   f_E2.ENO = 2
```

A model (database state) will look as follows (with any SID $s$ and arbitrary values in the open table entries):

| STUDENTS | | | |
|---|---|---|---|
| Tuple | SID | FIRSTNAME | LASTNAME |
| $f_S$ | s | | |

| EXERCISES | | |
|---|---|---|
| Tuple | ENO | TOPIC |
| $f_{E1}$ | 1 | |
| $f_{E2}$ | 2 | |

| GRADES | | | |
|---|---|---|---|
| Tuple | SID | ENO | POINTS |
| $f_G(f_{E1})$ | s | 1 | 10 |
| $f_G(f_{E2})$ | s | 2 | 10 |

As in this example, it is always possible to construct a database state that produces an answer to the query from a model of the flat form of the query. The database state will have one tuple in relation $R$ for each term in $\mathcal{T}_Q(R)$ (and no other tuples). It is possible that two of the constructed tuples are completely identical (i.e. there can be fewer tuples than elements in $\mathcal{T}_Q(R)$). In exceptional cases, it might even happen that a relation $R$ must be made empty although $\mathcal{T}_Q(R)$ is non-empty:

**Example 2** *In contrast to standard predicate logic, domains can be empty in SQL. Consider:*

```
SELECT * FROM DUMMY
WHERE  EXISTS
       (SELECT * FROM R WHERE 1=2)
OR NOT EXISTS
       (SELECT * FROM R)
```

*This is consistent in SQL because* R *can be empty. In this situation the predicate* NE_R *is needed in our construction (because of the first subquery, we would generate a Skolem constant of type* R*, but* NE_R *permits to invalidate or effectively remove it).*

**Theorem 1** *Let a query $Q$ be given such that $\mathcal{T}_Q$ is finite. $Q$ is consistent iff the flat form of $Q$ is consistent.*

For the proof, see: http://dbs.informatik.uni-halle.de/sqllint/ccheck05.pdf

### 3.3. Integrity Constraints

The above algorithm constructs just any model of the query, not necessarily a database state that satisfies all constraints. However, it is easy to add conditions to the query that ensure that all constraints are satisfied. For instance, consider the following constraint on GRADES:

```
CHECK(POINTS >= 0)
```

Then the following condition would be added to each query that references GRADES:

```
AND NOT EXISTS
        (SELECT *
         FROM   GRADES
         WHERE  NOT(POINTS >= 0))
```

The original query is consistent relative to the constraints iff this extended query is consistent.

Note that pure "for all" constraints like keys or CHECK-constraints do not need nested subqueries and thus never endanger the termination of the method. No new Skolem functions are constructed, the conditions are only instantiated for each existing Skolem term of the respective sort (relation). This is also what one would intuitively expect.

Foreign keys, however, require the existence of certain tuples, and therefore might sometimes result in an infinite set $\mathcal{T}_Q$. This is subject of the next section.

### 3.4. Restrictions and Possible Solutions

As mentioned above, the main restriction of our method is that the set $\mathcal{T}_Q$ must be finite, i.e. no tuple variable over a relation $R$ may depend directly or indirectly on a tuple variable over the same relation $R$. This

is certainly satisfied if there is only a single level of subqueries.

However, GRADES has a foreign key ENO that references EXERCISES. This can be enforced in models by adding the following condition to all queries:

```
AND NOT EXISTS
    (SELECT *
     FROM   GRADES C
     WHERE  NOT EXISTS
            (SELECT *
             FROM   EXERCISES P
             WHERE  P.ENO = C.ENO))
```

We now get a Skolem function

$$f_{\mathtt{P}} : \mathtt{GRADES} \rightarrow \mathtt{EXERCISES}.$$

In itself this would be no problem, and actually there will never be a problem if the foreign keys are not cyclic and the query itself contains only a single level of NOT EXISTS. But in Example 1, the query introduces the Skolem function

$$f_{\mathtt{G}} : \mathtt{EXERCISES} \rightarrow \mathtt{GRADES}.$$

Together we can now generate infinitely many terms: $f_{\mathtt{E1}}$, $f_{\mathtt{G}}(f_{\mathtt{E1}})$, $f_{\mathtt{P}}(f_{\mathtt{G}}(f_{\mathtt{E1}}))$, $f_{\mathtt{G}}(f_{\mathtt{P}}(f_{\mathtt{G}}(f_{\mathtt{E1}})))$, and so on.

However, it is easy to prove that $f_{\mathtt{P}}(f_{\mathtt{G}}(f_{\mathtt{E1}})) = f_{\mathtt{E1}}$: We know that

$$f_{\mathtt{G}}(f_{\mathtt{E1}})\mathtt{.ENO} = f_{\mathtt{E1}}\mathtt{.ENO} \text{ and}$$
$$f_{\mathtt{P}}(f_{\mathtt{G}}(f_{\mathtt{E1}}))\mathtt{.ENO} = f_{\mathtt{G}}(f_{\mathtt{E1}})\mathtt{.ENO}.$$

Since ENO is key of EXERCISES, the two tuples must be the same. Our claim is that adding such a check solves nearly all cases that occur in practice.

One case that is not solved are cyclic (recursive) foreign keys. Because of the undecidability, this problem can in general not be eliminated. However, one could at least heuristically try to construct a model by assuming that, e.g., 2 tuples in the critical relation $R$ suffice. Then $\mathcal{T}_Q(R)$ would consist of two constants and one would replace each subquery declaring a tuple variable over $R$ by a disjunction with these two constants. For relations not in the cycle, the original method could still be used. If the algorithm of Section 3.1 constructs a model, the query is of course consistent. If no model is found, the system can print a warning that it cannot verify the consistency. At user option, it would also be possible to repeat the step with more constants.

### 3.5. Null Values

Null values are handled in SQL with a three-valued logic. However, the consistency check [8] that we apply here is based on a two-valued logic (they assume that all attributes are not null). In this section, we will remove this restriction. First note that the presence of null values really makes a difference:

**Example 3** *The following query is inconsistent in two-valued logic (without null values):*

```
SELECT X.A
FROM   R X
WHERE  NOT EXISTS (SELECT *
                   FROM   R Y
                   WHERE  Y.B = Y.B)
```

*But in this DB state, the query result is non-empty:*

| R | |
|---|---|
| A | B |
| 1 | (null) |

We can handle null values by introducing new logic operators NTF ("null to false") and NTT ("null to true") with the following truth tables:

| $p$ | NTF($p$) | NTT($p$) |
|---|---|---|
| FALSE | FALSE | FALSE |
| NULL | FALSE | TRUE |
| TRUE | TRUE | TRUE |

In SQL, a query or subquery generates a result only when the WHERE-condition evaluates to TRUE. Thus, in Definition 5, we add the operator NTF for the query and all subqueries. In Example 3, a Skolem constant $f_{\mathtt{X}}$ is introduced for the tuple variable X, and the elimination of the subquery gives the following formula:

$$\mathtt{NTF(\ NOT\ NTF(}f_{\mathtt{X}}=f_{\mathtt{X}}\mathtt{)\ )}$$

As usual, NOT is first pushed down to the atomic formulas and is there eliminated by inverting the comparison operator. This needs the following equivalences (which can easily be checked with the truth tables):

- NOT NTF($\varphi$) $\equiv$ NTT(NOT $\varphi$)
- NOT NTT($\varphi$) $\equiv$ NTF(NOT $\varphi$)

Next the operators NTF and NTT can be pushed down to the atomic formulas by means of the following equivalences:

- NTF($\varphi_1$ AND $\varphi_2$) $\equiv$ NTF($\varphi_1$) AND NTF($\varphi_2$)
  ... (the same with NTT and both also with OR)
- NTT(NTF($\varphi$)) $\equiv$ NTF($\varphi$)
  ... (and so on: the outer operator can be deleted)

Next, the formula is as usual converted to DNF. After that, we must check the satisfiability of conjunctions of atomic formulas that have the operator NTF or NTT applied to them. An attribute can be set to null iff it appears only in atomic formulas inside NTT and the formula is not IS NOT NULL. Then it should be set to null, because these atomic formulas are already satisfied without any restrictions on the remaining attributes. Otherwise the attribute cannot be set to null. IS NULL and IS NOT NULL conditions can now be evaluated. After that, we apply the algorithm from [8] to the remaining atomic formulas (that are not already satisfied because of the null values).

## 4. Related Work

As far as we know, there is not yet a tool for checking given SQL queries for semantic/logical errors (except tutoring systems that have handcrafted examples built in).

Of course, for the problem of detecting inconsistent conditions, a large body of work exists in the literature. In general, all work in automated theorem proving can be applied (see, e.g., [4]). The problem whether there exists a contradiction in a conjunction of inequalities is very relevant for many database problems and has been intensively studied in the literature. Klug's classic paper [10] checks for such inconsistencies but does not treat subqueries and assumes dense domains for the attributes. The algorithm in [9] can handle recursion, but only negations of EDB predicates, not general NOT EXISTS subqueries. A very efficient method has been proposed by Guo, Sun, Weiss [8]. We use it here as a subroutine. Our main contribution is the way we treat subqueries. Although this uses ideas known from Skolemization, the way we apply it combined with an algorithm like [8], apply the relations as sorts, and detect equal terms in the Herbrand universe seems new. We also can handle null values. Consistency checking in databases has also been applied for testing whether a set of constraints is satisfiable. A classic paper about this problem is [3]. They give an algorithm which terminates if the constraints are finitely satisfiable or if they are unsatisfiable, which is the best one can do. However, the approach presented here can immediately tell whether it can handle the given query and constraints. Also in the field of description logics, decidable fragments of first order logic are used. Recently Minock [11] defined a logic that is more restricted than ours, but is closed under syntactic query difference.

Consistency of database queries has also connections to semantic query optimization (see, e.g., [5]), and cooperative query answering (see, e.g., [7, 6]). However, in semantic query optimization, only relatively simple consistency checks can be used for efficiency reasons (optimization time must be amortized during later execution). In cooperative query answering, a database state is given: The system does not really notice that the query is inconsistent, only that it yields an empty answer in the current state. In both fields, the question for possible runtime errors is not asked.

Of course, test data generation as in [13] is also a form of consistency check, and both papers have in part overlapping goals. However, the concrete methods are quite different.

In [12], the coverage of test data for SQL statements is analyzed.

## 5. Conclusions

Quality software must be reliable in the sense that it never generates runtime errors (exceptions). A check for the safety of SQL queries can be reduced to a consistency check. In this paper, we proposed a new method for checking the consistency of SQL queries: We used a state-of-the-art algorithm for conjunctions of comparisons ($=, \neq, <, >, \leq, \geq$) [8], and extended it by handling subqueries with an interesting variant of Skolemization, and null values/three-valued logic with operators "null-to-false" and "null-to-true". The method can also be used to generate test data. Our overall goal is to develop a semantic checker for SQL queries [2]. The consistency check described in this paper is already implemented in the current prototype. In future work, we intend to extend our consistency check to aggregations (and LIKE).

## References


[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1994.

[2] Stefan Brass and Christian Goldberg. Semantic errors in SQL queries: A quite complete list. In *Proceedings of the Fourth International Conference on Quality Software (QSIC'04)*, pages 250–257. IEEE Computer Society Press, 2004. Extended version to appear in Journal of Systems and Software.

[3] François Bry and Rainer Manthey. Checking consistency of database constraints: a logical basis. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB'86)*, pages 13–20. Morgan Kaufmann, 1986.

[4] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[5] Qi Cheng, Jarek Gryz, Fred Koo, Cliff Leung, Linqi Liu, Xiaoyan Qian, and Bernhard Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *Proc. of the 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 687–698, 1999.

[6] Wesley W. Chu, Hua Yang, Kuorong Chiang, Michael Minock, Gladys Chow, and Chris Larson. CoBase: A scalable and extensible cooperative information system. *Journal of Intelligent Information Systems*, 6:223–259, 1996.

[7] Terry Gaasterland, Parke Godfrey, and Jack Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992.

[8] Sha Guo, Wei Sun, and Mark A. Weiss. Solving satisfiability and implication problems in database systems. *ACM Transactions on Database Systems*, 21:270–293, 1996.

[9] Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Static analysis in datalog extensions. *Journal of the ACM*, 48:971–1012, 2001.

[10] Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35:146–160, 1988.

[11] Michael J. Minock. Knowledge representation using schema tuple queries. In *10th International Workshop on Knowledge Representation meets Databases (KRDB 2003)*, pages 51–62, 2003.

[12] María José Suárez-Cabal and Javier Tuya. Using an SQL coverage measurement for testing database applications. In *Proc. of the 12th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*, pages 253–262. ACM Press, 2004.

[13] Jian Zhang, Chen Xu, and S.-C. Cheung. Automatic generation of database instances for white-box testing. In *Proc. of the 25th International Computer Software and Applications Conference (COMPSAC'01)*, pages 161–165. IEEE Press, 2001.