Detecting Logical Errors in SQL Queries

Stefan Brass and Christian Goldberg

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
(brass|goldberg)@informatik.uni-halle.de

Abstract. Checking the consistency of query conditions is an old problem, and there are already many proposed solutions. Since the problem is in general undecidable, there is not a unique best one. However, for conjunctions of comparisons, it seems that the method of Guo, Sun, and Weiss (1996) is the state of the art. In this paper, we show how it can be extended to handle many cases of subqueries (and also null values). In this way, the consistency of a surprisingly large subset of SQL can be decided. We apply this consistency check to the task of finding semantic errors in SQL queries. In particular, we discuss possible runtime errors in SQL queries and show how a test for such errors can be reduced to a consistency check. We believe that future database management systems will perform such checks and that the generated warnings will help to develop code with fewer bugs in less time.

1 Introduction

Errors in SQL queries can be classified into syntactic errors and semantic errors. A syntactic error means that the entered character string is not a valid SQL query. Then any DBMS will print an error message because it cannot execute the query. Thus, the error is certainly detected and usually easy to correct.

A semantic error means that a legal SQL query was entered, but the query does not always produce the intended results, and is therefore incorrect for the given task. Semantic errors can be further classified into cases where the task must be known in order to detect that the query is incorrect, and cases where there is sufficient evidence that the query is incorrect no matter what the task is. Our focus in this paper is on this latter class.

For instance, consider the following query:

```
SELECT *
FROM   EMP
WHERE  JOB = 'CLERK' AND JOB = 'MANAGER'
```

This is a legal SQL query, and it is executed, e.g., in Oracle9i and DB2 V8.1 without any warning. However, the condition is actually inconsistent, so the query result will be always empty. Since nobody would use a database in order to get an always empty result, we can state that this query is incorrect without actually knowing what the task of the query was. Such cases do happen. For

example, in one exam exercise that we analyzed, 10 out of 70 students wrote an inconsistent condition. That is not an exceptional case: In another course, there was an exercise in which 14 out of 148 students made this error. A list of semantic errors that can be detected without knowledge of the task is contained in [2]. That paper contains also some statistics about the occurrence of each error type in exams. In the exams that we analyzed, 28% of the solutions contained a semantic error that would be detectable without knowledge of the task, and 32% contained a syntax error (there was 9% overlap of both error categories). This shows that a tool for detecting semantic errors would be very useful — not only for teaching, but it could speed up debugging also in practice: Many of the students will afterwards become professional software engineers, and they do not immediately make fewer errors.

Of course, inconsistent conditions are easy to find by running the query on example data. However, that is not true for all errors in SQL queries. In Section 3 we discuss a check for the possibility of runtime errors in SQL. Since these errors do not occur in every execution, they are not necessarily found during testing. Proving that runtime errors cannot occur would increase the reliability of the software. Our check for runtime errors as well as the checks for many of the other error types discussed in [2] reduce the problem to a consistency check.

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 SQL subsets. For conjunctions of comparisons, it seems that the method of Guo, Sun, and Weiss (1996) is the state of the art. In this paper (Section 2), 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. Related work is further discussed in Section 4.

2 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 θ 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 2.5, before that, they are excluded. Aggregations are not treated in this paper, they are subject of our future research.

2.1 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 [13].

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, so it 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 φ_i is consistent.

Now a conjunction of the above atomic formulas can be tested for satisfiability with the method of [13]. 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 [13] is fast. (However, the DNF computation that we apply before [13] can lead to an exponential increase in size.)

2.2 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(`<u>`SID`</u>`, FIRSTNAME, LASTNAME)`
- `EXERCISES(`<u>`ENO`</u>`, TOPIC)`
- `GRADES(`<u>`SID`</u>`→STUDENTS, `<u>`ENO`</u>`→EXERCISES, POINTS)`

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

```
SELECT S.FIRSTNAME, S.LASTNAME
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

```
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., [8]), 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 required 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 a 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 $f_{\texttt{G}}$ 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:

- $f_{\texttt{S}}$: `STUDENTS`,
- $f_{\texttt{G}}$: `EXERCISES` $\rightarrow$ `GRADES`,
- $f_{\texttt{E1}}$: `EXERCISES`,
- $f_{\texttt{E2}}$: `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,

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

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 2.4 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 with equality is decidable (this was proven 1928 by Bernays and Schönfinkel). However, as the example shows, our method can sometimes handle even heavily nested subqueries, because the set of Skolem terms does not necessarily become infinite. In this way, the sorted logic used in SQL differs from the classical approach.

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 [13] (see Section 2.1):

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 φ)

   by $\sigma(\varphi)$ with a substitution σ that replaces the existential tuple variable X_i by $f_{X_i}(Y_{i,1},\ldots,Y_{i,m_i})$, where $Y_{i,1},\ldots,Y_{i,m_i}$ 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 φ)

   by ($\sigma_1(\varphi)$ OR ... OR $\sigma_k(\varphi)$)*, where σ_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).*

In the above example, we would first substitute S by $f_{\texttt{S}}$, E1 by $f_{\texttt{E1}}$, E2 by $f_{\texttt{E2}}$, and G by $f_{\texttt{G}}(\texttt{E})$. Since E is of type EXERCISES, and $f_{\texttt{E1}}$ and $f_{\texttt{E2}}$ are the only elements of $\mathcal{T}_Q(\texttt{EXERCISES})$, the disjunction consists of two cases with E replaced by $f_{\texttt{E1}}$ and by $f_{\texttt{E2}}$, respectively. To illustrate this, let us look at the query with the substitutions:

```
SELECT f_S.FIRSTNAME, f_S.LASTNAME
FROM   STUDENTS S -- replaced by f_S
WHERE  NOT(   EXISTS(SELECT *
                     FROM   EXERCISES E -- version for f_E1
                     WHERE  NOT EXISTS(
                            SELECT *
                            FROM   GRADES G -- f_G(f_E1)
                            WHERE  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 EXISTS(SELECT *
                     FROM   EXERCISES E -- version for f_E2
                     WHERE  NOT EXISTS(
                            SELECT *
                            FROM   GRADES G -- f_G(f_E2)
                            WHERE  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    EXISTS(SELECT * FROM EXERCISES E1 -- replaced by f_E1
              WHERE  f_E1.ENO = 1)
AND    EXISTS(SELECT * FROM EXERCISES E2 -- replaced by f_E2
              WHERE  f_E2.ENO = 2)
```

Formally, we would directly construct the flat form of the above query, in which the SELECT and FROM clauses and the EXISTS and 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 the opposite direction, note that NOT EXISTS ($\forall$) conditions are only more difficult to satisfy if the database state contains more tuples. Therefore, with a single level NOT EXISTS subquery, we need one tuple for each of the tuple variables in the outer query, but we would introduce no additional tuples for the relations listed under NOT EXISTS. It is the basic idea of Skolemization that we can give names to the tuples that the formula requires to exist, and then reduce the given model to all the named elements.

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.*

Example 2. For instance, the following query is very similar to a query written in an exam. The task is to find students that have not yet submitted any homework.

```
SELECT S.FIRSTNAME, S.LASTNAME
FROM   STUDENTS S, GRADES G
WHERE  S.SID = G.SID
AND    NOT EXISTS
       (SELECT *
        FROM   GRADES X
        WHERE  X.SID = S.SID)
```

The subquery correctly requires that there is no entry in GRADES for the student in question, but the join in the outer query requires that there is such an entry. This is clearly inconsistent. Skolem constants $f_{\mathtt{S}}$ and $f_{\mathtt{G}}$ are constructed, and in the subquery, X is replaced by $f_{\mathtt{G}}$ (the only Skolem term of sort GRADES). Thus, we have to check the following condition for consistency:

$$f_{\mathtt{S}}\mathtt{.SID} = f_{\mathtt{G}}\mathtt{.SID}$$
$$\mathtt{AND\ NOT(}f_{\mathtt{G}}\mathtt{.SID} = f_{\mathtt{S}}\mathtt{.SID)}$$

Obviously, the formula is inconsistent. □

2.3 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 need only a single level of NOT EXISTS and therefore 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.

2.4 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:

```
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 2.1 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.

2.5 Null Values

Null values are handled in SQL with a three-valued logic.

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)
```

However, this query is satisfiable in SQL if the attribute `B` can be null: It has a model in which `R` contains, e.g., one tuple `t` with `t.A=1` and `t.B` is 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:

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

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

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

- NTF(φ_1 AND φ_2) $\equiv$ NTF(φ_1) AND NTF(φ_2)
- NTF(φ_1 OR φ_2) $\equiv$ NTF(φ_1) OR NTF(φ_2)
- NTT(φ_1 AND φ_2) $\equiv$ NTT(φ_1) AND NTT(φ_2)
- NTT(φ_1 OR φ_2) $\equiv$ NTT(φ_1) OR NTT(φ_2)
- NTF(NTF(φ)) $\equiv$ NTF(φ)
- NTT(NTF(φ)) $\equiv$ NTF(φ)
- NTF(NTT(φ)) $\equiv$ NTT(φ)
- NTT(NTT(φ)) $\equiv$ NTT(φ)

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 [13] to the remaining atomic formulas (that are not already satisfied because of the null values).

3 Possible Runtime Errors

Sometimes, SQL queries must not return more than one value, otherwise a runtime error occurs. This might be difficult to find during testing, because the error

does not always appear. Especially, if the programmer wrongly assumes that the data always satisfies the necessary condition, the query will run correctly in all test database states. But if the condition is not enforced by a constraint, sooner or later users will enter data that make the query crash. It would be good if a tool could verify that such errors do not occur. Of course, the problem is in general undecidable, thus we cannot expect to handle all queries.

The test can be easily reduced to a consistency check. Let us first consider `SELECT INTO` in embedded SQL, which works only if the query returns not more than one row:

```
SELECT t1, ..., tk
INTO   v1, ..., vk
FROM   R1 X1, ..., Rn Xn
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 2.3), the runtime error can occur, and the constructed model gives an example:

```
SELECT *
FROM   R1 X1, ..., Rn Xn, R1 X1', ..., Rn Xn'
WHERE  φ AND φ'
AND    (X1 ≠ X1' OR ··· OR Xn ≠ Xn')
```

The formula φ' results from φ 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 φ is consistent).

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

```
SELECT *
FROM   R1 X1, ..., Rn Xn, R1 X1', ..., Rn Xn'
WHERE  φ AND φ'
AND    (t1 <> t1' OR ··· OR tk <> tk'
        OR t1 IS NULL AND t1' IS NOT NULL
        OR t1' IS NULL AND t1 IS NOT NULL
        ...
        OR tk IS NULL AND tk' IS NOT NULL
        OR tk' IS NULL AND tk IS NOT NULL)
AND    (X1 ≠ X1' OR ··· OR Xn ≠ Xn')
```

In the same way, `GROUP BY` queries can be treated: Then $t_1, \ldots, t_n$ are the `GROUP BY` attributes.

The same problem occurs with conditions of the form `A = (SELECT ...)`, when the subquery returns more than one value. Actually, 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. 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)
```

However, it might be possible to interpret the restriction in a more liberal way. Consider the following artificial query:

```
SELECT X.SID, X.FIRSTNAME, X.LASTNAME
FROM   STUDENTS X, STUDENTS Y
WHERE  10 = (SELECT Z.POINTS FROM GRADES Z
             WHERE Z.ENO = 1 AND (Z.SID = X.SID OR Z.SID = Y.SID)
AND    X.SID = Y.SID
```

If the conditions are evaluated in the sequence in which they are written down, and there are at least two students with 10 points for Exercise 1, this would give a runtime error. If the condition on the tuple variables in the outer query is evaluated first, there would be no error. Even if the conditions were written in the opposite sequence, it is not clear whether this query should be considered as OK. After all, the query optimizer should have the freedom to choose an evaluation sequence. This is a general problem with runtime errors, also known from programming languages. The SQL-92 standard does not address this problem. If one should decide that some part of the outer condition is evaluated before the subquery, one could add that part to our test query.

In Oracle9i, the example does not generate a runtime error: It seems that the condition in the outer query is evaluated first or pushed down into the subquery (independent of the sequence of the two conditions). However, one can construct an example with two subqueries, where Oracle generates a runtime error for φ_1 `AND` φ_2, but not for φ_2 `AND` φ_1. Therefore, in order to be safe, one should require that the subquery returns a single value for any given assignment of the tuple variables in the outer query (not necessarily one that satisfies other conditions of the query). This is what we have encoded in the test above.

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 has not a numeric format.
3. In addition, datatype operators have the usual problems (e.g., division by zero). If the division 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. It were useful if DBMS vendors would use a four-valued logic with an additional "`ERROR`" value, and equations like "`ERROR AND FALSE = FALSE`". Then an error message would only be printed if the entire `WHERE`-condition evaluates to "`ERROR`". This would improve the declarativity of SQL. We do not know whether DBMS already behave in this way (at least Oracle9i does not).

4 Related Work

As far as we know, there is not yet a tool for checking given SQL queries for semantic/logical errors without knowledge about the application. However, the question is strongly related to the two fields of semantic query optimization and cooperative answering.

Of course, for the special 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., [8]). 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 [16] checks for such inconsistencies but does not treat subqueries and assumes dense domains for the attributes. The algorithm in [14] 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 [13]. 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 [13], 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] (see also [4]). 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 [17] defined a logic that is more restricted than ours, but is closed under syntactic query difference.

There is a strong connection to semantic query optimization (see, e.g., [6, 5]). However, the goals are different. As far as we know, DB2 contains some semantic query optimization, but prints no warning message if the optimizations are "too good to be true". Also the effort for query optimization must be amortized when the query is executed, whereas for error detection, we would be willing to spend

more time. Finally, soft constraints (that can have exceptions) can be used for generating warnings about possible errors, but not for query optimization.

Our work is also related to the field of cooperative query answering (see, e.g., [12, 9, 11]). However, there the emphasis is more on the dialogue between DBMS and user. As far as we know, a question like the possibility of runtime errors in the query is not asked. Also, usually a database state is given in cooperative ansering, whereas we do not assume any particular state. For instance, the CoBase system would try to weaken the query condition if the query returns no answers. It would not notice that the condition is inconsistent and thus would not give a clear error message. However, the results obtained there might help to suggest corrections for a query that contains this type of semantic error.

Further studies about errors in database queries, especially psychological aspects, are [19, 18, 15, 10, 7].

5 Conclusions

There is a large class of SQL queries that are syntactically correct, but nevertheless certainly not intended, no matter what the task of the query might be. One could expect that a good DBMS prints a warning for such queries, but, as far as we know, no DBMS does this yet.

In this paper we have analyzed two kinds of such semantic errors: Inconsistent conditions and queries that might generate runtime errors. There are many further types of errors that can be detected by static analysis of SQL queries, a list is given in [2].

A prototype of (an older version of) the consistency test is available from

`http://www.informatik.uni-halle.de/~brass/sqllint/`.

A new version is currently being developed and will be made available under the same address.

Acknowledgements

Elvis Samson developed the prototype of the consistency test, and made several suggestions for improving the paper, which is both gratefully acknowledged. We would also like to thank Alexander Hinneburg for helpful comments and Jan Van den Bussche for suggesting important relevant literature. We had a very interesting and inspiring discussion with Ralph Acker from Transaction Software about runtime errors, which helped us a lot. Last but not least, without the students in our database courses, we would not have started this work.

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: *Fourth International Conference on Quality Software (QSIC'04)*, IEEE Computer Society Press, 2004. To appear.
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*, 13–20, 1986.
4. François Bry, Hendrik Decker, and Rainer Manthey: A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *Proceedings of the International Conference on Extending Database Technology*, 488–505, 1988.
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. *Proceedings of the 25th VLDB Conference*, 687-698, 1999.
6. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15:162–207, 1990.
7. Hock C. Chan. The relationship between user query accuracy and lines of code. *Int. Journ. Human Computer Studies 51*, 851-864, 1999.
8. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.
9. Wesley W. Chu, M.A. Merzbacher, and L. Berkovich. The design and implementation of CoBase. In *Proc. of ACM SIGMOD*, 517-522, 1993.
10. Hock C. Chan, Bernard C.Y. Tan, and Kwok-Kee Wei. Three important determinants of user performance for database retrieval. *Int. Journ. Human-Computer Studies 51*, 895-918, 1999.
11. 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*, 1996.
12. Terry Gaasterland, Parke Godfrey, and Jack Minker. An Overview of Cooperative Answering. Journal of Intelligent Information Systems 21:2, 123–157, 1992.
13. 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.
14. A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in Datalog extensions. *Journal of the ACM 48*, 971–1012, 2001.
15. W.J. Kenny Jih, David A. Bradbard, Charles A. Snyder, and Nancy G.A. Thompson. The effects of relational and entity-relationship data models on query performance of end users. *Int. Journ. Man-Machine Studies*, 31:257–267, 1989.
16. Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35:146–160, 1988.
17. Michael J. Minock. Knowledge Representation under the Schema Tuple Query Assumption. In *10th International Workshop on Knowledge Representation meets Databases (KRDB 2003)*.
`http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-79/`
18. A. Rizzo, S. Bagnara, and Michele Visciola. Human error detecting processes. *Int. Journ. Man-Machine Studies 27*, 555-570, 1987.
19. C. Welty. Correcting user errors in SQL. *International Journal of Man-Machine Studies 22:4*, 463-477, 1985.