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. Queries that are contradictory, i.e. always return the empty set, are quite often written in exams of database courses. However, such queries are executed in current database management systems (e.g., Oracle) without any warning. Of course, questions like the satisfiability are in general undecidable, but we give a quite simple algorithm that can handle a surprisingly large subset of SQL queries. We then analyze unnecessary logical complications. Furthermore, 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. In this case, 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 the Oracle8i DBMS 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.

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 that checks whether the database contains a solution to a Post's correspondence problem, see [1], Section 6.3. This query does not contain any datatype operations.

However, although the task is in general undecidable, we will show that many cases that occur in practice can be detected with relatively simple algorithms.

Our work is also inspired by the program `lint`, which is or was a semantic checker for the "C" programming language. Today C compilers do most of the checks that `lint` was developed for, but in earlier times, C compilers checked just enough so that they could generate machine code. We are still at this development stage with SQL today. Printing warnings for strange SQL queries is very uncommon in current database management systems.

The program `lint` tried to find also errors like uninitialized variables. This is a clearly undecidable task, and therefore `lint` sometimes produced error messages for programs that were correct (and missed some other errors). But in general, `lint` was a useful tool. In case of a wrong warning, a good programmer will think about possible alternative formulations that are easier to verify. Such formulations will often also be easier to understand by other programmers who later have to read the code. If there was no better formulation, one could put a special comment into the program that suppressed the warning.

We believe that such a tool would be useful not only in teaching, but also in application software development.

Our paper is structured as follows: In Section 2, we show how the consistency of queries in an SQL subset can be checked by combining the well-known Skolemization method with an efficient algorithm for deciding the consistency of conjunctive queries. We also explain how to extend the method to the three-valued logic used in SQL, and how to handle program variables in Embedded SQL. In the short Section 3, other unnecessary logical complications are discussed. In Section 4 we show how the consistency check can also be applied in order to verify that a certain kind of runtime error cannot occur: In some contexts, SQL queries or subqueries must return not more than one row. In Section 5, we give some pointers to related work.

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). It should be quite easy to extend it at least to linear equations (e.g., `X.A = 2*Y.A+5*Y.B`). 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 labelled with "Tuplevariable.Attribute" (maybe a representative for an equivalence class with respect to $=$) and edges are labelled 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 are only finitely many 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.

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 a subquery that is nested inside an even number (including 0) of* `NOT`*s, and universal otherwise. For instance, the tuple variables in the outermost (main) query are existential.*

Example 1. The following SQL query lists all locations of departments, such that all departments at the same location have at least one "Salesman":

```
SELECT DISTINCT L.LOC
FROM   DEPT L
WHERE  NOT EXISTS(SELECT *
                  FROM   DEPT D
                  WHERE  D.LOC = L.LOC
                  AND    NOT EXISTS(SELECT *
                                    FROM   EMP E
                                    WHERE  E.DEPTNO = D.DEPTNO
                                    AND    E.JOB = 'SALESMAN'))
```

L and E are existential tuple variables, and D is a universal tuple variable. □

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 L in the example), a single tuple is required in the database state. However, for an existential tuple variable like E that is declared within the scope of a universal tuple variable (D) a different tuple might be required for every value for D. Therefore, a function $f_{\mathtt{E}}$ is introduced that takes a value for D as a parameter and returns a value for E. Such a function is called a Skolem function. There is also a Skolem function $f_{\mathtt{L}}$ for L, but this function has no parameters (it is a Skolem constant).

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 $f_{\mathtt{E}}$ in the example assumes that a tuple from the relation DEPT is given, and returns a tuple from the relation EMP.

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 is one Skolem constant and one Skolem function:

- $f_{\mathtt{L}}$: DEPT,
- $f_{\mathtt{E}}$: DEPT $\rightarrow$ EMP.

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, $\mathcal{T}_Q = \{f_{\mathtt{L}},\ f_{\mathtt{E}}(f_{\mathtt{L}})\}$.

Of course, in general it is possible that infinitely many terms can be constructed. Then we cannot predict how large a model (database state/instance) must be and our method is not applicable. However, this 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. The problem of an infinite $\mathcal{T}_Q$ is treated further in Section 2.4.

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 `L` by $f_{\mathtt{L}}$ and `E` by $f_{\mathtt{E}}$(`D`). Since `D` is of type `DEPT` and $f_{\mathtt{L}}$ is the only element of $\mathcal{T}_Q$(`DEPT`), the disjunction consists of a single case with `D` replaced by f_L. Thus, the flat form of the above query is

```
NOT(f_L.LOC = f_L.LOC
    AND NOT(f_E(f_L).DEPTNO = f_L.DEPTNO
            AND f_E(f_L).JOB = 'SALESMAN'))
```

This is logically equivalent to

$$f_{\mathtt{E}}(f_{\mathtt{L}})\mathtt{.DEPTNO} = f_{\mathtt{L}}\mathtt{.DEPTNO\ AND\ } f_{\mathtt{E}}(f_{\mathtt{L}})\mathtt{.JOB = 'SALESMAN'}$$

A model (database state/instance) will have two tuples, one ($f_{\mathtt{L}}$) in DEPT, and another ($f_{\mathtt{E}}(f_{\mathtt{L}})$) in EMP. The requirements are that their attributes DEPTNO are equal and that the attribute JOB of the tuple in EMP has the value 'SALESMAN'.

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/instance 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/instance 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 was written in an exam. The task was to find columns that are not indexed.

```
SELECT X.TABLE_NAME, X.COLUMN_NAME
FROM   COLS X, USER_IND_COLUMNS Y
WHERE  X.TABLE_NAME = Y.TABLE_NAME
AND    X.COLUMN_NAME = Y.COLUMN_NAME
AND    NOT EXISTS
       (SELECT *
        FROM   USER_IND_COLUMNS Z
        WHERE  X.TABLE_NAME = Z.TABLE_NAME
        AND    X.COLUMN_NAME = Z.COLUMN_NAME)
```

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

```
f_X.TABLE_NAME = f_Y.TABLE_NAME
AND f_X.COLUMN_NAME = f_Y.COLUMN_NAME
AND NOT(f_X.TABLE_NAME = f_Y.TABLE_NAME
        AND f_X.COLUMN_NAME = f_Y.COLUMN_NAME)
```

Obviously, the formula is inconsistent. □

2.3 Integrity Constraints

Consider the following query:

```
SELECT ...
FROM   EMP X, EMP Y
WHERE  X.EMPNO = Y.EMPNO
AND    X.JOB = 'MANAGER' AND Y.JOB = 'PRESIDENT'
```

This query is inconsistent, but we need to know that EMPNO is a key of EMP in order to prove that. The above algorithm constructs just any model of the query, not necessarily a database state/instance that satisfies all constraints. However, it is easy to add conditions to the query that ensure that all constraints are satisfied. For example, instead of the above query, we would check the following one which explicitly requires that there is no violation of the key:

```
SELECT ...
FROM   EMP X, EMP Y
WHERE  X.EMPNO = Y.EMPNO
AND    X.JOB = 'MANAGER' AND Y.JOB = 'PRESIDENT'
AND    NOT EXISTS(SELECT *
                  FROM   EMP A, EMP B
                  WHERE  A.EMPNO = B.EMPNO
                  AND    (A.ENAME <> B.ENAME OR
                          A.JOB <> B.JOB OR ...))
```

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 instanciated 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, EMP has a foreign key MGR (manager) that references the relation itself. This is expressed as the following condition:

```
NOT EXISTS(SELECT *
           FROM   EMP E
           WHERE  E.MGR IS NOT NULL
           AND    NOT EXISTS (SELECT *
                              FROM   EMP M
                              WHERE  E.MGR = M.EMPNO))
```

We now get a Skolem function $f_{\mathtt{M}} : \mathtt{EMP} \rightarrow \mathtt{EMP}$, which will generate infinitely many terms (if there is at least one Skolem constant of type EMP).

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 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, when `EXISTS` subqueries are eliminated in Definition 5, we add the operator `NTF`:

$$\texttt{NTF}(\sigma_1(\varphi)\ \texttt{OR}\ \ldots\ \texttt{OR}\ \sigma_k(\varphi)).$$

In Example 3, a Skolem constant $f_{\texttt{X}}$ is introduced for the tuple variable `X`, and the elimination of the subquery gives the following formula:

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

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):

- $\texttt{NOT NTF}(\varphi) \equiv \texttt{NTT}(\texttt{NOT}\ \varphi)$
- $\texttt{NOT NTT}(\varphi) \equiv \texttt{NTF}(\texttt{NOT}\ \varphi)$

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

- $\texttt{NTF}(\varphi_1\ \texttt{AND}\ \varphi_2) \equiv \texttt{NTF}(\varphi_1)\ \texttt{AND}\ \texttt{NTF}(\varphi_2)$
- $\texttt{NTF}(\varphi_1\ \texttt{OR}\ \varphi_2) \equiv \texttt{NTF}(\varphi_1)\ \texttt{OR}\ \texttt{NTF}(\varphi_2)$
- $\texttt{NTT}(\varphi_1\ \texttt{AND}\ \varphi_2) \equiv \texttt{NTT}(\varphi_1)\ \texttt{AND}\ \texttt{NTT}(\varphi_2)$
- $\texttt{NTT}(\varphi_1\ \texttt{OR}\ \varphi_2) \equiv \texttt{NTT}(\varphi_1)\ \texttt{OR}\ \texttt{NTT}(\varphi_2)$
- $\texttt{NTF}(\texttt{NTF}(\varphi)) \equiv \texttt{NTF}(\varphi)$
- $\texttt{NTT}(\texttt{NTF}(\varphi)) \equiv \texttt{NTF}(\varphi)$
- $\texttt{NTF}(\texttt{NTT}(\varphi)) \equiv \texttt{NTT}(\varphi)$
- $\texttt{NTT}(\texttt{NTT}(\varphi)) \equiv \texttt{NTT}(\varphi)$

Next, the formula is as usual converted to DNF. After that, we must check the satisfiablility of conjunctions of atomic formulas that have possibly 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).

2.6 Program Variables in SQL Statements

In order to be practical, a tool for consistency checks must also be able to check SQL queries in application programs. Then the queries can contain program variables. In general, the program variables can be treated like attributes of a new relation. However, the user should at least be warned if a variable can have only a single value in a consistent state, or if two variables must always have the same value. Such restrictions are unlikely: The programmer could as well insert the only possible value of the program variable, or merge the two variables. If

necessary, the condition that refers only to the program variables could be tested outside the SQL query in an `if`-statement.

This more general consistency check can be done as follows: The satisfiability test constructs a model, i.e. concrete values c_i for each program variable v_i. Then one can run it again (once for each program variable) and add $v_i \neq c_i$ to the condition. In the same way, if the constructed model assigns the same value to two distinct program variables v_i and v_j, one can add the condition $v_i \neq v_j$.

3 Unnecessary Logical Complications

Sometimes, a subcondition is inconsistent, but the entire condition is consistent (e.g., because of a disjunction). Of course, also the opposite can happen: Subconditions that are tautologies. Both kinds of unnecessary complications indicate logical misconceptions and it is quite likely that the query will not behave as expected.

Furthermore, implied subconditions are unnecessary complications. In certain circumstances, implied subconditions can help the optimizer to find a better execution plan, but then they should better be clearly marked as optimizer hint. In exams, it happens quite often that students add a condition, such as "`A IS NOT NULL`" that is already enforced as a constraint.

There are (at least) three possible formalizations of the requirement for "no unnecessary logical complications":

Definition 6.

1. *A query condition φ satisfies criterion 1 iff it is consistent, its negation is consistent, and whenever a single subcondition of φ is negated, it stays consistent.*
2. *A query condition φ satisfies criterion 2 iff it is not equivalent to true or false, and whenever a subcondition is replaced by $0 = 0$ (true) or $1 = 0$ (false), the resulting formula is not equivalent to φ.*
3. *A query condition φ satisfies criterion 3 iff its disjunctive normal form satisfies criterion 2.*

Theorem 2. *If a query condition satisfies criterion 2, it also satisfies criterion 1. Furthermore, it is obvious that criterion 3 implies criterion 2.*

In both cases, the opposite is not true. For example, `(A=2 AND A>1) OR B=1` satisfies criterion 1, but not criterion 2: The underlined subcondition can be replaced by `true`, but every subcondition can be negated without the entire condition becoming inconsistent. The condition `(A=1 OR B>2) AND B>0` satisfies criterion 2, but not criterion 3. However, whereas criterion 2 can always be reached by making the formula shorter, reaching criterion 3 might actually make the formula longer: Consider, e.g., `(A=1 OR B=1) AND (A=2 OR C=2)`. But for conjunctions of atomic formulas, all three criteria are equivalent.

The test for each of the three criteria can be reduced to a series of consistency checks. Let us consider criterion 3, and let the DNF of the query condition be

$C_1 \vee \cdots \vee C_m$, where $C_i = (A_{i,1} \wedge \cdots \wedge A_{i,n_i})$. Then criterion 3 is satisfied iff the following formulas are all consistent:

1. $\neg(C_1 \vee \cdots \vee C_m)$, the negation of the entire formula (otherwise the entire formula could be replaced by "true"),
2. $C_i \wedge \neg(C_1 \vee \cdots \vee C_{i-1} \vee C_{i+1} \vee \cdots \vee C_m)$, for $i = 1, \ldots, m$ (otherwise C_i could be replaced by "false"),
3. $A_{i,1} \wedge \cdots \wedge A_{i,j-1} \wedge \neg A_{i,j} \wedge A_{i,j+1} \wedge \cdots \wedge A_{i,n_i} \wedge \neg(C_1 \vee \cdots \vee C_{i-1} \vee C_{i+1} \vee \cdots \vee C_m)$ for $i = 1, \ldots, m$, $j = 1, \ldots, n_i$ (otherwise $A_{i,j}$ could be replaced by "true").

Another type of unnecessary logical complication is to use a "too general" comparison operator. For instance, we saw something like the following in more than one homework:

```
SELECT ENAME, SAL
FROM   EMP
WHERE  SAL >= (SELECT MAX(SAL) FROM EMP)
```

Here, the "`>=`" can be replaced by "`=`", which would make the condition clearer. Also "`IN`" was used in such queries, although it is guaranteed here that the subquery can return only a single value. The converse case is treated in the next section.

Of course, unnecessary joins are another important type of logical complication that should be avoided. This was already studied extensively in the literature.

4 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.

It would be good if a tool could verify that such errors do not occur. Of course, the problem is in general undecidable.

The test can be easily reduced to a consistency check. Let the following general query be given:

```
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:

```
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 t1, ..., tk
FROM   R1 X1, ..., Rn Xn
WHERE  φ
```

If it access 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   R1 X1, ..., Rn Xn, R1 X1', ..., Rn Xn', S1 Y1, ..., Sm Ym
WHERE  φ AND φ'
AND    (X1 ≠ X1' OR ··· OR Xn ≠ Xn')
```

However, it might be possible to interpret the restriction in a more liberal way. Consider a database with relations `R(A, B)` and `S(A, B)` (`A` is in both cases the primary key). Let the query be:

```
SELECT *
FROM   R X, R Y
WHERE  X.B = (SELECT S.B FROM S WHERE S.A = X.A OR S.A = Y.A)
AND    X.A = Y.A
```

If the conditions are evaluated in the sequence in which they are written down, 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).

5 Related Work

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

Actually, Oracle's precompiler for Embedded SQL (Pro*C/C++) has an option for semantic checking, but this means only that it checks whether tables and columns exist and that the types match.

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] seems new. We also can handle null values and query parameters. 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 decription 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 optimzation, 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, there usually is a database state given, 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.

The SQL Tutor system discussed in [18, 19] assumes knowledge about the specific task in form of a correct query.

Further studies about errors in database queries, especially psychological aspects, are [21, 20, 15, 10, 7].

6 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 some kinds of such semantic errors: Inconsistent conditions, unnecessary logical complications, 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 the technical report [2].

A prototype 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

Of course, without the students in my database courses, this work would have been impossible. 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 very helpful discussions. Among other contributions, he suggested the problem of the program variables.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1994.
2. Stefan Brass, Christian Goldberg, Alexander Hinneburg. *Detecting Semantic Errors in SQL Queries*. Technical Report, University of Halle, 2003.
3. François Bry, 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, 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, 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 determinats 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, 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. Mitrovic. A knowledge-based teaching system for SQL. In *ED-MEDIA 98*, pages 1027–1032, 1998.
19. Antonija Mitrovic, Brent Martin, and Michael Mayo. Using evaluation to shape its design: Results and experiences with SQL-Tutor. *User Modeling and User-Adapted Interaction*, 12:243–279, 2002.
20. A. Rizzo, S. Bagnara and Michele Visciola. Human error detecting processes. *Int. Journ. Man-Machine Studies 27*, 555-570, 1987.
21. C. Welty. Correcting user errors in SQL. *International Journal of Man-Machine Studies 22:4*, 463-477, 1985.