Detecting Logical Errors in SQL Queries

Stefan Brass 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. Although such queries are obviously not intended, they 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 Oracle9i DBMS without any warning. However, the condition is actually inconsistent, so the query result will be always empty. Since nobody would use a database to get a fixed 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. However, 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.

2 Inconsistent Conditions

The full version of the paper contains 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/instance, such that the query result is not empty.

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). Our algorithm can treat null values. Aggregations are excluded, they are subject of our future research.

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 [6].

We treat subqueries with a variant of the Skolemization method, well known from the field of automated theorem proving. For space reasons, we can only give an example here. First, it suffices to treat `EXISTS` subqueries. Other kinds of subqueries (`IN`, `>=ALL`, etc.) can be reduced to the `EXISTS` case.

Let us use the following SQL query as an example. It 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'))
```

The idea of Skolemnization is to introduce names (constants, function symbols) for values that are required to exist. Tuple variables are existential iff they are declared in a subquery that is nested in an even number (including 0) of `NOT` (`L` and `E` in the example). Otherwise they are universal (`D` in the example).

For existential variables that are not in the scope of a universal quantifier (such as `L` in the example), a single tuple is required in the database state. Thus, we introduce a Skolem constant $f_{\mathtt{L}}$ of type `DEPT` for `L`. 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 Skolem function $f_{\mathtt{E}}$ is introduced that takes a value for `D` as a parameter and returns a value for `E`. The function $f_{\mathtt{E}}$ has the parameter type `DEPT` and the result type `EMP`. The the Herbrand universe (the set of terms that can be constructed with these constants and function symbols) is $\mathcal{T}_Q = \{f_{\mathtt{L}},\ f_{\mathtt{E}}(f_{\mathtt{L}})\}$. We write $\mathcal{T}_Q(R)$ for the terms of type R.

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.

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 [6]. 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

   $$\texttt{EXISTS (SELECT} \ \ldots \ \texttt{FROM} \ R_1\, X_1\texttt{,} \ \ldots\texttt{,} \ R_n\, X_n \ \texttt{WHERE} \ \varphi\texttt{)}$$

   by $\sigma(\varphi)$ with a substitution σ that maps the existential variable X_i to $f_{X_i}(Y_{i,1}, \ldots, Y_{i,m_i})$, where $Y_{i,1}, \ldots, Y_{i,m_i}$ are all universal variables in the scope of which this subquery appears and which appear in the subquery.

3. Finally treat subqueries that appear within an odd number of negations as follows: Replace the subquery

   $$\texttt{EXISTS (SELECT} \ \ldots \ \texttt{FROM} \ R_1\, X_1\texttt{,} \ \ldots\texttt{,} \ R_n\, X_n \ \texttt{WHERE} \ \varphi\texttt{)}$$

   by $\texttt{(}\sigma_1(\varphi) \ \texttt{OR} \ \ldots \ \texttt{OR} \ \sigma_k(\varphi)\texttt{)}$, 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 $f_{\mathtt{E}}(\mathtt{D})$. Since `D` is of type `DEPT` and $f_{\mathtt{L}}$ is the only element of $\mathcal{T}_Q(\mathtt{DEPT})$, the disjunction consists of a single case with `D` replaced by f_L. Thus, the flat form of the above query is

$$\begin{array}{l}\texttt{NOT(}f_{\mathtt{L}}\texttt{.LOC = }f_{\mathtt{L}}\texttt{.LOC}\\ \quad\texttt{AND NOT(}f_{\mathtt{E}}(f_{\mathtt{L}})\texttt{.DEPTNO = }f_{\mathtt{L}}\texttt{.DEPTNO}\\ \qquad\quad\texttt{AND }f_{\mathtt{E}}(f_{\mathtt{L}})\texttt{.JOB = 'SALESMAN'))}\end{array}$$

This is logically equivalent to

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

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.

Constraints can be handled by adding to the query the condition that violations of the constraint do not exist.

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 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 different possible formalizations of the requirement for "no unnecessary logical complications". A quite strict version is that whenever in the DNF of the query condition, a subcondition is replaced by "true" or "false", the result is not equivalent to the original condition. This can be reduced to a series of consistency checks. 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 our criterion 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 $i = 1, \dots, m$ (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 $i = 1, \dots, m$, $j = 1, \dots, n_i$ (otherwise $A_{i,j}$ could be replaced by "true").

Another type of unnecessary logical complication is to use a "too general" comparison operator. (e.g. "`>=`" when "`=`" would suffice). Furthermore, unnecessary joins are an important type of logical complication that was already studied extensively in the literature.

4 Possible Runtime Errors

In conditions of the form `A = (SELECT ...)`, the subquery must not return more than one value, otherwise a runtime error occurs. Furthermore, when `SELECT ... INTO ...` is used in Embedded SQL, the query must return at most one row.

Errors of this type are difficult to find during testing, because they do not always occur. 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 subquery be given:

$$\begin{array}{ll} \texttt{SELECT} & t_1\texttt{,}\ \dots\texttt{,}\ t_k \\ \texttt{FROM} & R_1\, X_1\texttt{,}\ \dots\texttt{,}\ R_n\, X_n \\ \texttt{WHERE} & \varphi \end{array}$$

Let $S_1\, Y_1$, ..., $S_m\, Y_m$ be the variables from the outer query that are accessed in the subquery ($m = 0$ is possible for uncorrelated subqueries). 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 (after adding the constraints), the runtime error can occur:

$$\begin{array}{ll} \texttt{SELECT} & * \\ \texttt{FROM} & R_1\, X_1\texttt{,}\ \dots\texttt{,}\ R_n\, X_n\texttt{,}\ R_1\, X_1'\texttt{,}\ \dots\texttt{,}\ R_n\, X_n'\texttt{,}\ S_1\, Y_1\texttt{,}\ \dots\texttt{,}\ S_m\, Y_m \\ \texttt{WHERE} & \varphi\ \texttt{AND}\ \varphi' \\ \texttt{AND} & (X_1 \neq X_1'\ \texttt{OR}\ \cdots\ \texttt{OR}\ X_n \neq X_n') \end{array}$$

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. 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). "`SELECT DISTINCT`" and "`GROUP BY`" are treated in the full version of this paper.

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 check whether the subquery cannot return more than a single row for any given assignment of the tuple variables of the outer query (not necessarily an assignment that satisfies the conditions of the outer query).

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, the above strict condition seems right.

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 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 compagnion paper (in these proceedings). 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.

References

[1] F. Bry, R. Manthey: Checking Consistency of Database Constraints: a Logical Basis. In *Proceedings of the 12th International Conference on Very Large Data Bases*, 13–20, 1986.

[2] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, B. Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. *Proceedings of the 25th VLDB Conference*, 687-698, 1999.

[3] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15:162–207, 1990.

[4] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[5] T. Gaasterland, P. Godfrey, J. Minker. An Overview of Cooperative Answering. Journal of Intelligent Information Systems 21:2, 123–157, 1992.

[6] S. Guo, W. Sun, and M. A. Weiss. Solving satisfiability and implication problems in database systems. *ACM Transactions on Database Systems 21*, 270–293, 1996.