Semantic 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|hinneburg)@informatik.uni-halle.de

Abstract. We investigate classes of SQL queries which are syntactically correct, but certainly not intended, no matter for which task the query was written. For instance, queries that are contradictory, i.e. always return the empty set, are quite often written in exams of database courses. Current database management systems, e.g. Oracle, execute such queries without any warning. In this paper, we try to give a complete list of such errors. Of course, questions like the satisfiability are in general undecidable, but a significant subset of SQL queries can actually be checked. This also applies to the other errors explained in this paper. We are currently developing a tool that does such checks and believe that it will be very helpful in teaching SQL (especially for distance learing and online courses).

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 or 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, e.g. 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, and that this applies also to database queries. E.g. 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 (like `+`, `*`).

However, although the task is in general undecidable, 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 there 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. At least, a good error message could speed up the debugging process. Also in some contexts, SQL queries or subqueries must return not more than one row. Otherwise a runtime error is generated, and the application is terminated. Certainly it would be good to prove that all queries in an application program can never violate this condition.

The main contribution of this paper is a list of semantic errors that represents years of experience while correcting hundreds of exams that contained SQL queries. However, we have also tried to explain the general principles from which these errors can be derived (as far as possible). Therefore, it is not simply by chance whether an error appears on our list, but the list has a certain degree of completeness.

2 Unnecessary Complications

Of course, in general it is difficult to state that a syntactically correct query is semantically wrong if one does not know the task for which the query was written. However, queries can be considered as "probably not intended" when they are unnecessarily complicated. Suppose the user wrote a query Q, and there is an equivalent query Q' that is significantly simpler, and basically can be derived from Q by deleting certain parts. There might be the following reasons why the user did not write Q':

- The user knew that Q' is not a correct formulation of the task at hand. In this case, Q is of course also not correct, but the error might be hidden in the more complicated query, so that the user did not realize this. A warning would certainly be helpful in this case.
- The user did not know that Q' is equivalent. Since Q' is not a completely different query, but results from Q by deleting certain parts, this shows that the user does not yet master SQL. Again, a warning would be helpful. Often, the simpler query will actually run faster (e.g. the Oracle query optimizer does not remove unnecessary joins).
- The user knew that Q' is equivalent, but he or she believed that Q would run faster. Since SQL is a declarative language this should only be the last resort. With modern optimizers, this should not happen often in practice. If it is necessary, there probably should be some comment, and this could also be used to shut off the warning. Although we know at least one case where a more complicated query actually does run faster on Oracle 9i, SQL does not make any guarantees about how a query is evaluated. Thus, in the next Oracle version or when one uses a different DBMS, it might be that the relative speed of Q and Q' dramatically changes.
- The user knew that Q' is equivalent, but he or she thought that Q would be clearer for the human reader and easier to maintain. One must be careful to define the possible transformations from Q to Q' such that this does not happen. For instance, it might be clearer to use explicit tuple variables in attribute references, even if the attribute name is unique. Of course, removing the tuple variable in this case cannot be considered as producing a different, shorter query. Even more obviously, we would not require that meaningful names for tuple variables are shortened or that comments are removed. Also certain optional keywords (e.g. "`AS`") are a matter of taste. Unfortunately, this means that every possible "shortening transformation" of SQL queries must be considered separately (as done below).

Actually, "equivalence" in the sense of requiring exactly the same query result in all database states would make the condition still too strict. First, we not only want to minimize the query, but also the query result. Consider the following query which is quite typical for beginning SQL programmers:

```
SELECT EMPNO, ENAME, JOB
FROM   EMP
WHERE  JOB = 'MANAGER'
```

The last column in the query result is superfluous, we know that it must always be "`MANAGER`". Therefore, no information is lost when this column is removed. Actually, we will see that when we have to decide between a short query and a short query result, it is likely that we will prefer the short result. Of course, trying to minimize the query result without loss of information does not mean that we apply compression algorithms or difficult encodings. The important requirement is that from the shorter query result, the user can reconstruct the original query result with "very little intellectual effort" — less than what would be required for reading the long version. This statement is a bit fuzzy, but it can be made precise by listing the operations that are permitted for reconstructing the original query result. In this paper, we only need constant relations (in the case of inconsistent conditions) and projections. In the example, we would use

$$\pi_{\text{EMPNO, ENAME, JOB} \leftarrow \text{'MANAGER'}}.$$

Furthermore, it is better to exclude certain unusual states when we require that the result of both queries (Q and Q') is the same. For example, it happens sometimes that students declare a tuple variable, and then do not use it and forget to delete it:

```
SELECT DISTINCT DNAME
FROM   DEPT, EMP
```

The "DISTINCT" is also a typical example where the wrong patch was applied to a problem noticed by the student (many duplicates). The above query returns always the same result as the following one, except when EMP is empty:

```
SELECT DISTINCT DNAME
FROM   DEPT
```

Therefore, we will require the equivalence only for states in which all relations are non-empty. It might even be possible to assume that all columns contain at least two different values.

Some types of errors produce many duplicates. More powerful query simplifications can be used if these duplicates are not considered as important for the equivalence (at least if the simpler query Q' produces less duplicates than the more difficult query Q). E.g. in the above example, we would want to delete the unused tuple variable even if DISTINCT were not specified. Duplicates are further considered in Section 5.

Now we give a list of all cases in which a query can be obviously simplified under this slightly weakened notion of equivalence. In each of these cases, a warning should be given to the user.

2.1 Entire Query Unnecessary

Error 1: Inconsistent conditions. As noted above, nobody would pose a query to a database if he or she knew beforehand that the query result is empty, no matter what the database state is. In this case, the entire query is superfluous. However, students do write queries with inconsistent conditions in exams.

Note that we must also consider integrity constraints when we talk inconsistencies. It might well be that the query condition itself is consistent, but it contradicts declared integrity constraints. Even then the query result will always be empty.

It is actually possible that a query result is not empty, but still does not depend on the database state. Again, the entire query is superfluous. This never happened in exams we corrected so far, but let us mention this for completeness. An example is:

```
SELECT COUNT(*)
FROM   DEPT
WHERE  1 = 2
```

2.2 Unnecessarily Complicated SELECT Clause

Error 2: Unnecessary duplicate elimination. We try to teach our students to use `DISTINCT` only when necessary. E.g. we would consider the following query as suboptimal, because the presence of the key `EMPNO` in the query result ensures that there are no duplicates even without `DISTINCT`:

```
SELECT DISTINCT EMPNO, ENAME, JOB
FROM   EMP
```

This might be a matter of style, but at least the Oracle 8i optimizer does not remove the unnecessary duplicate elimination, and therefore, the query runs slower with `DISTINCT`. Also, when a query actually does produce duplicates, it is often helpful to understand why. Always writing “`DISTINCT`” shadows possible errors.

Error 3: Constant output column. An output column is unnecessary if it contains a single value that is constant and can be derived from the query without any knowledge about the database state. This was already illustrated at the beginning of this section.

Error 4: Duplicate output column. An output column is also unnecessary if it is always identical to another output column, e.g.

```
SELECT X.EMPNO, X.DEPTNO, Y.DEPTNO, Y.DNAME
FROM   EMP X, DEPT Y
WHERE  X.DEPTNO = Y.DEPTNO
```

In general, it might be possible to say that an output column is unnecessary if it can be computed from the remaining columns. However, if the computation rule is not trivial, it might be useful if SQL computes the derived value.

2.3 Unnecessary Complications in the FROM Clause

The next three errors are cases where tuple variables are declared under `FROM` that are not really necessary.

Error 5: Unused tuple variables. First, it happens quite often that a student first declares a tuple variable under from, but does not use it at all. It might be that he/she simply forgot to delete it. However, some students also seem to think that tuple variables used in subqueries must also be declared in the main query.

Error 6: Unnecessary joins. If only the key attributes of a tuple variable X are accessed, and this key is equated with the foreign key of another tuple variable Y, X is not needed. An example is:

```
SELECT EMPNO, ENAME, X.DEPTNO
FROM   DEPT X, EMP Y
WHERE  X.DEPTNO = Y.DEPTNO AND Y.JOB = 'MANAGER'
```

Error 7: Tuple variables that are always identical. If the key attributes of two tuple variables X and Y over the same relation are equated, the two tuple variables must always point to the same tuple. Then the two tuple variables can be merged. Often, this will lead to an inconsistent condition.

2.4 Unnecessary Complications in the WHERE Clause

Error 8: Implied, tautological, or inconsistent subconditions.
The `WHERE`-condition is unnecessarily complicated if a subcondition (some node in the operator tree) can be replaced by `TRUE` or `FALSE` and the condition is still equivalent. E.g. it happens sometimes that a condition is tested under `WHERE` that is actually a constraint on the relation, e.g.

```
SELECT DNAME
FROM   DEPT
WHERE  DNAME IS NOT NULL
```

If `DNAME` is declared as `NOT NULL`, the test is unnecessary.

Error 9: Unnecessarily general comparison operator. Consider the query:

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

In this case, one could write `=` instead of `>=`. We have also seen students writing `IN` here, which is again quite confusing. Another quite common case is to use `LIKE` when the comparison string contains no wildcards "`%`" or "`_`" (the only reason could be that one needs a no-pad comparison semantics, but that happens very seldom).

Error 10: Unnecessary SELECT arguments in EXISTS-subqueries. In `EXISTS`-subqueries, the `SELECT`-list is not important. Therefore, it should be something simple (e.g. "`*`" or "`1`" or a single attribute). The following is unnecessarily complicated:

```
SELECT DNAME
FROM   DEPT D
WHERE  NOT EXISTS(SELECT EMPNO, ENAME, JOB
                  FROM   EMP E
                  WHERE  E.DEPTNO = D.DEPTNO)
```

Error 11: IN/EXITS condition can be replaced by comparison.
Consider the following query:

```
SELECT ENAME
FROM   EMP X
WHERE  X.EMPNO NOT IN (SELECT Y.EMPNO
                       FROM   EMP Y
                       WHERE  Y.JOB = 'MANAGER')
```

It is equivalent to

```
SELECT ENAME
FROM   EMP X
WHERE  X.JOB <> 'MANAGER'
```

The point here is that the two tuple variables over the same relation are matched on their key. This is very similar to Error 7 above, but here a subquery is involved.

2.5 Unnecessary Complications in Aggregation Functions

Error 12: Unnecessary DISTINCT in aggregations. The aggregations `MIN` and `MAX` never need `DISTINCT`. For the aggregations `SUM` and `AVG` it is very unusual to use `DISTINCT`: When there are duplicates, they are most probably significant. However, when `DISTINCT` is used, it might not always be necessary, e.g. in:

```
SELECT COUNT(DISTINCT EMPNO)
FROM   EMP
```

Error 13: Unnecessary argument of COUNT. There are two versions of the aggregation function `COUNT`: One with an argument, and one without an argument (written as `COUNT(*)`). We would prefer the version without argument whenever this is equivalent, i.e. when there is no `DISTINCT` and when the argument cannot be null. However, that might be a matter of taste. But we have seen quite a number of times something like the following:

```
SELECT COUNT(JOB)
FROM   EMP
WHERE  JOB = 'MANAGER'
```

In this example, only duplicates are counted. Listing "`JOB`" as the argument of `COUNT` obscures the meaning of the query. The following is equivalent and much clearer:

```
SELECT COUNT(*)
FROM   EMP
WHERE  JOB = 'MANAGER'
```

2.6 Unnecessary Complications in the GROUP BY Clause

Error 14: GROUP BY with singleton groups. If it can be proven that each group consists only of a single row, the entire aggregation is unnecessary. An example is:

```
SELECT   EMPNO, ENAME, MAX(SAL)
FROM     EMP
GROUP BY EMPNO, ENAME
```

Students sometimes write this when asked to compute the employee with maximum salary (especially when it was emphasized in the course that all attributes used under `SELECT` outside of aggregations must appear under `GROUP BY`).

Error 15: GROUP BY with only a single group. If it can be proven that there is always only a single group, the `GROUP BY` clause is unnecessary, except when the `GROUP BY` attribute should be printed under `SELECT`. An example is

```
SELECT   COUNT(*)
FROM     EMP
WHERE    JOB = 'MANAGER'
GROUP BY JOB
```

Error 16: Unnecessary GROUP BY attributes. If a grouping attribute is functionally determined by other such attributes and if it does not appear under `SELECT` or `HAVING` outside of aggregations, it can be removed from the `GROUP BY` clause.

2.7 Unnecessary Complications in the HAVING Clause

In the `HAVING`-clause, the same errors as in the `WHERE`-clause are possible. In addition, conditions that are possible under `WHERE` are better written there (see Error 18 below).

2.8 Unnecessary Complications in the ORDER BY Clause

Error 17: Unnecessary ORDER BY terms. Suppose that the order by clause is `ORDER BY` $t_1, \ldots, t_n$. Then t_i is unnecessary if it is functionally determined by $t_1, \ldots, t_{i-1}$. This especially includes the case that t_i has only one possible value.

3 Inefficient Formulations

Although SQL is a declarative language, students should be trained to help the system to execute the query efficiently. Errors 2 and 12 (`DISTINCT` when no duplicates are possible) also fall in this category. However, in the following two cases the query does not get shorter by choosing the more efficient formulation.

Error 18: Inefficient HAVING. If a condition uses only `GROUP BY` attributes and no aggregation function, it can be written under `WHERE` or under `HAVING`. It is much cheaper to check it already under `WHERE`. E.g. in one homework, a join was done under `HAVING`, and it was syntactically correct SQL, because the student added the join attributes under `GROUP BY`.

```
SELECT   D.DNAME, COUNT(*)
FROM     DEPT D, EMP E
GROUP BY D.DEPTNO, D.DNAME, E.DEPTNO
HAVING   D.DEPTNO = E.DEPTNO
```

Error 19: Inefficient UNION. A `UNION` should be replaced by a `UNION ALL` if one can prove that the results of the two queries are always disjoint.

4 Missing Join Conditions

Error 20: Missing Join Conditions. Missing join conditions are a type of semantic error that is mentioned in most text books. However, it is not completely clear how a test for missing join conditions should be formally done. A simple approach is to create a graph with the tuple variables X as nodes. Edges are drawn between nodes for tuple variables X and Y if there is any atomic formula in which both tuple variables appear. The graph then should be connected, with the possible exception of nodes X such that there is a condition $X.A = c$ with a key attribute A and a constant c. This is the weakest form of the test. One could also draw edges only for equality conditions. Actually, we have seen a query that contained an error because a tuple variable had a composed key and only one of the attributes was joined. Thus, the strictest form is to require joins only over foreign key-key pairs, except in the case of self joins. In addition, one probably should convert the `WHERE`-condition first to DNF, and require sufficient join conditions in each conjunction. We are currently investigating which of the possible variants is best (i.e. catching the most errors while not producing too many wrong warnings). Note that in some cases, joins can also be done via subqueries.

In one exam exercise, 11 out of 70 students had an error of this type, although students had been warned in the course about missing join conditions.

Error 21: Uncorrelated EXISTS-Subqueries. If an `EXISTS`-subquery makes no reference to a tuple variable from the outer query, is is either globally true or globally false. Thus, it either does not change the query result, or it makes the query result empty. This is a very unusual behaviour. Actually, uncorrelated `EXISTS`-subqueries are simply missing join conditions (possibly for anti-joins).

5 Duplicates

Query results that contain many duplicates are difficult to read. It is unlikely that such a query is really intended. Furthermore, duplicates are often an indication for another error, e.g. missing join conditions. Of course, if we could give a more specific warning, that would be preferable.

Error 22: Many duplicates. Consider the following example:

```
SELECT JOB
FROM   EMP
```

This query will produce many duplicates without any order. It is quite clear that it would have been better to chose one of the following formulations:

– If the number of duplicates is not important:

```
SELECT DISTINCT JOB
FROM   EMP
```

– If it is important:

```
SELECT   JOB, COUNT(*)
FROM     EMP
GROUP BY JOB
```

There are two possible ways to detect this problem: First, we could run the query on an example database state. If it produces more than 50% duplicates, we could give a warning. Techniques developed in query optimization for estimating the result size can also be used: If the estimated size is extremely large, one should warn the user before the query is really executed.

Second, we check whether the query is guaranteed to return no duplicates. That would be the same test as for the unnecessary `DISTINCT` above. However, dupicates are not always a problem. Consider the following query:

```
SELECT ENAME
FROM   EMP
WHERE  DEPTNO = 20
```

Although it might be possible that there are two employees with a common name, this is not very likely. And when it happens, the duplicate might be important. The reason is that although the name is not a strict key, it is used in practice in order to identify employees. Thus, we need a declaration of such "soft keys". Then we simply check whether `DISTINCT` would be necessary under the assumption that these soft keys were real keys.

6 Possible Runtime Errors

In C programs, it sometimes happens that a NIL-pointer is dereferenced, and the program crashes. Actually, such runtime errors are also possible in SQL, and one should try to verify that they cannot occur. Since these problems depend on the database state, they are not easily found during testing.

Error 23: Subqueries that must not return more than one tuple. If one uses a condition of the form `A = (SELECT ...)`, it is important that the subquery returns only a single value. If this condition should ever be violated, the DBMS will generate a run-time error. This can be tested with a method very similar to the test for an unnecessary `DISTINCT` shown above, one only replaces the `SELECT`-list by “`SELECT 'yes'`.

The same problem happens if SQL is embedded in a programming language, and one uses the `SELECT ... INTO ...` syntax.

Error 24: No indicator variable for arguments that might be null. In Embedded SQL, it is necessary to specify an indicator variable if a result column can be null. If no indicator variable is specified, a runtime error results. Note that this can happen also with aggregation functions that get an empty input.

Error 25: Difficult Type Conversions Also, the very permissive type system of at least Oracle SQL can pose a problem: Sometimes strings are implicitly convered to numbers, which can generate runtime errors. In general, if one knows domains for the attributes, one could warn the user for comparisons between attributes of different domains. If there is no domain information, one could analyze an example database state for columns that are nearly disjoint.

Error 26: Possible runtime errors in datatype functions. Datatype operators have the usual problems (e.g. division by zero).

7 Other Indicators for Errors

Error 27: Wildcards Without LIKE. When “`=`” is used with a comparison string that contains “`%`”, probably “`LIKE`” was meant. For the other wildcard, “`_`”, it is not that clear, because it might more often appear in normal strings.

Error 28: Strange HAVING. `HAVING` without `GROUP BY` is strange: Such a query can have only one result or none at all. In special situations this may be a useful trick, but more often it is probably an error.

Error 29: SELECT-Clause of Subquery uses no tuple variable from the subquery. E.g. something like the following puzzled a team of software engineers for quite some time, they even thought that their DBMS contained a bug, because it did not give any error:

```
SELECT ENAME
FROM   EMP
WHERE  DEPTNO IN (SELECT EMPNO
                  FROM   DEPT
                  WHERE  LOC = 'BOSTON')
```

The underlined attribute is a typing error, correct would be `DEPTNO`. However, this is correct SQL, `EMPNO` simply references the tuple variable from the outer query. A missing join condition should have been detected here, but a more specific error message might be helpful. Furthermore, this error might also occur for other types of subqueries, when there are join conditions.

8 Style Checks

1. A warning should be printed if a tuple variable is "shadowed" by a tuple variable of the same name in a subquery.
2. A warning should be printed if a tuple variable from an outer query is accessed without its name in a subquery (i.e. only "A" instead of "$X.A$").
3. An `IN`-subquery that is correlated, i.e. accesses tuple variables from the outer query, should probably be replaced by an `EXISTS`-subquery.
4. It is strange when `GROUP BY` is used, but no aggregation function appears. This can be used for duplicate elimination, but then `DISTINCT` would be clearer. (However, it is also possible to eliminate only some duplicates with `GROUP BY`).
5. It might be a matter of taste, but a large number of unnecessary parentheses is also not helpful for reading the query. Some students who are unsure about the precedence rules completely parenthesize the `WHERE`-condition.
6. Of course, SQL queries should be portable between database management systems. However, there is a tradeoff between portability on the one hand, and conciseness and efficiency on the other hand.

9 Algorithms

We do not have space here to give algorithms for all of these problems. Consistency tests have been studied in the literature for a long time, some references are given in the next section. We have shown in [2] that the consistency can be decided for all SQL queries that contain only a single level of `EXISTS`-subqueries, no aggregations and no datatype functions (such as `+`, `*`). Often a much deeper nesting of subqueries is possible. Furthermore, heuristics can be used if the consistency cannot be decided.

Now, tests for most of the problems explained in this paper can be reduced to a consistency test. As an example, let us consider the test whether `DISTINCT` is necessary. It is quite typical. Let the following general query be given:

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

Now we modify the query as follows (we duplicate the tuple variables and check whether there are two different assignments that produce the same result for the `SELECT` terms):

```
SELECT *
FROM   $R_1\,X_1$ , ... , $R_n\,X_n$ , $R_1\,X'_1$ , ... , $R_n\,X'_n$
WHERE  $\varphi$ AND $\varphi'$
AND    ($t_1$ = $t'_1$ OR $t_1$ IS NULL AND $t'_1$ IS NULL)
AND    ...
AND    ($t_k$ = $t'_k$ OR $t_k$ IS NULL AND $t'_k$ IS NULL)
AND    ($X_1 \neq X'_1$ OR $\cdots$ OR $X_n \neq X'_n$)
```

This query is tested for consistency with the method of [2]. If it is inconsistent, the DISTINCT is superfluous: The original query can never produce duplicates.

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, duplicate result tuples are always possible and the DISTINCT is not superfluous. The formula φ' results from φ by replacing each X_i by X'_i.

10 Related Work

It seems that the general question of detecting semantic errors in SQL queries (as defined above) is new.

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. Also "Trouble Checker" from http://www.msqlproducts.com claims semantic checking, by it concentrates on procedures and triggers, e.g. it finds loops in triggers. These checks do not cover semantic errors in the most important declarative part of SQL.

The need of semantically rich error and warning messages for SQL statements in a learning context has been investigated in [17, 18]. However, the SQL Tutor system proposed there has knowledge about the task that has to be solved (in form of a correct query). In contrast, our approach assumes no such knowledge, which makes it applicable also for software development, not only for teaching.

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 [13] 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 [12]. It can only handle conjunctions of equalities and inequalities.

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 in general.

There is a strong connection of semantic query optimization (see e.g. [6, 14, 5]), to detecting semantic errors, but 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. [9, 11]) but there the 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 [20, 19, 15, 10, 7].

11 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 shown various kinds of semantic errors that can be detected without knowing the task of the query. All errors (except 12, 17, 24, 26) did actually occur in exam or homework exercises. More complete statistics about errors in the analyzed exams will be made available on the internet:

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

This page also contains a prototype of the consistency test. We have algorithms for detecting all of the above error types (for a suitable SQL subset). We currently add these tests to our prototype system and expect that a version that finds many of the above error types will be available for demonstration at the workshop.

Acknowledgements

Of course, without the students in my database courses, this work would have been impossible. We would also like to thank Jan Van den Bussche for suggesting relevant literature and Joachim Biskup for contributing the idea that query size estimation techniques could be used. We would like to thank Sergei Haller and Ravishankar Balike for telling us about Error 29. Elvis Samson developed the prototype of the consistency test, and made several suggestions for improving the paper, which is both gratefully acknowledged. The discussions with Alexander Hinneburg have been most helpful, especially he suggested further related work and gave us an example for an SQL query that is longer than an equivalent query, but runs faster.

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. 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.
13. A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in Datalog extensions. *Journal of the ACM 48*, 971–1012, 2001.
14. Chun-Nan Hsu and Craig A. Knoblock. Using inductive learning to generate rules for semantic query optimization. In *Advances in Knowledge Discovery and Data Mining*, pages 425–445. AAAI/MIT Press, 1996.
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. A. Mitrovic. A knowledge-based teaching system for SQL. In *ED-MEDIA 98*, pages 1027–1032, 1998.
18. 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.
19. A. Rizzo, S. Bagnara and Michele Visciola. Human error detecting processes. *Int. Journ. Man-Machine Studies 27*, 555-570, 1987.
20. C. Welty. Correcting user errors in SQL. *International Journal of Man-Machine Studies 22:4*, 463-477, 1985.