Semantic Errors in SQL Queries: A Quite Complete List

Stefan Brass, Christian Goldberg

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany

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 obviously not intended. However, current database management systems (DBMS) execute such queries without any warning. In this paper, we give an extensive list of conditions that are strong indications of semantic errors. Of course, questions like the satisfiability are in general undecidable, but a significant subset of SQL queries can actually be checked. We believe that future DBMS will perform such checks and that the generated warnings will help to develop application programs with fewer bugs in less time.

Key words: Databases, SQL, queries, bugs, errors, semantic errors, logical errors, software correctness, static analysis, teaching, database courses, SQL exams

1 Introduction

SQL is today the standard language for relational and object-relational databases. Application programs typically contain a relatively large number of SQL queries and updates, which are sent to the DBMS for execution. As any program code, SQL queries can contain errors (Updates are not considered in this paper, but they are usually much simpler than queries.).

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

Email address: (brass|goldberg)@informatik.uni-halle.de (Stefan Brass, Christian Goldberg).

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, since there is often no independent specification of the goal of the query. For instance, consider this query:

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

This is a legal SQL query, and it is executed e.g. in Oracle9i and DB2 V8.1 without any warning. But the condition is inconsistent: The query result will be always empty. Since nobody would use a database in order to get a certainly empty result, we can state that this query is incorrect without actually knowing the task of the query. 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. 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.

We currently develop a similar tool for SQL queries (called `sqllint`). 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. Furthermore, runtime errors are possible in SQL, e.g., in some contexts, SQL queries or subqueries must return not more than one row. The occurrence of this error depends on the database state (the data), therefore it is not necessarily found during testing. Certainly it would be good to prove that all queries in a program can never violate this condition. Our tool does not depend on the data, it only takes the schema information (including constraints) and an SQL query as input. Therefore, it is not necessary to check the queries in each execution. Furthermore, we do not need test data:

All input to our tool is (part of) the program code that is anyway written during software development. The only exception is the specification of "soft keys" (see Section 5) which would be useful for some of our checks.

While the title of this paper is "Semantic Errors", people from compiler construction have advised us that what we compute are really "warnings", since the queries are still executable, and in some cases only "notices" about bad style. Nevertheless, this information would help to improve the software quality of database application programs, which are a significant subset of the software developed today.

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 (except possibly in Section 4).

While our experience so far has only been with errors made by students, not professional programmers, most of the students will become programmers, and they will not immediately make fewer errors. In the exam solutions that we analyzed, 24% contained a syntax error, 18% contained a semantic error of the type we consider in this paper (detectable without knowledge of the task), 11% contained both, and 10% contained a semantic error that was only detectable if the task was known (for more details, see Section 9). Thus, with the techniques described in this paper, in 18% of the cases one could have got a sensible error message, while a standard DBMS simply executes the query (with possibly a wrong result, which might or might not be detected).

The paper is structured by reasons why SQL queries can be suspicious: Unnecessary complications (Section 2), inefficient formulations (Section 3), violations of standard patterns (Section 4), many duplicates (Section 5), and the possibility of runtime errors (Section 6). Furthermore we suggest some style checks in Section 7. While we do not have space to give algorithms for all errors (and many are quite obvious), we show how to detect two representative errors in Section 8. Section 9 contains some statistics how often each type of error appeared in exams. Related work is discussed in Section 10.

In the examples, we use a database schema for storing information about employees and departments:

```
EMP(EMPNO, ENAME, JOB, SAL, COMM, MGR→EMP, DEPTNO→DEPT)
DEPT(DEPTNO, DNAME, LOC)
```

This is a slightly modified version of an example schema that comes with the Oracle DBMS. The column `MGR` can be null.

2 Unnecessary Complications

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 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 (at least one widely used commercial DBMS 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.

  Note that in this point SQL differs from usual programming languages: Of course, a C program of 100 lines might be better than a program of 10 lines for the same task, because the longer program runs significantly faster. But as just explained, for SQL the relative speed is not forseeable, and any such code optimizations are at least fragile.
- The user knew that Q' is equivalent, but 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. Removing the tuple variable in this case cannot be considered as producing a different, shorter query. Obviously, we would also not require that meaningful names for tuple variables are shortened or that comments are removed. Furthermore, using certain optional keywords (e.g. "`AS`") is 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.

(1) First, we not only want to minimize the query, but also the query result. The following query 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. 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

$$\texttt{OLD\_RESULT} = \pi_{\texttt{EMPNO}, \texttt{ENAME}, \texttt{JOB} \leftarrow \texttt{'MANAGER'}}(\texttt{NEW\_RESULT}).$$

(2) 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 this one, except when $\texttt{EMP} = \emptyset$:

```
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.

(3) 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 condition.

Nobody would pose a query if he or she knew beforehand that the query result is empty, no matter what the database state is. An example was given in the introduction.

Note that it is also possible that the `WHERE`-clause itself is consistent, but it contradicts a constraint in the database schema. For instance, since SQL has no enumeration types, one typcially restricts columns of type "`CHAR`" with constraints like

```
CHECK(SEX = 'M' OR SEX = 'F')
```

Now the condition `SEX = 'W'` as part of a large conjunction in the `WHERE`-clause would immediately make the query inconsistent. Yet, no DBMS we are aware of would give a warning.

As mentioned before, the problem is in general undecidable [1]. But e.g. for SQL queries that contain only one level of subqueries and no aggregations, it is decidable (see, e.g., [3]). In general, one could apply heuristic assumptions (e.g., that a certain number of tuples per relation suffices) to generate warnings that are not always accurate, but still useful. There is a large body of literature about satisfiability tests and model construction, also in the context of databases [4,5,20,24,16,28,21].

In general, one could also construct other queries that have a constant result for all database states (under the assumption that relations are not empty). But this is at least very uncommon (it did not occur in the analyzed exams).

2.2 SELECT Clause

Error 2: Unnecessary DISTINCT.

One should use an explicit duplicate elimination only if necessary. It sometimes can be proven that a query cannot return duplicates, e.g.

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

Because `EMPNO` is a key of `EMP`, the query could not return duplicates, even without `DISTINCT`. Then `DISTINCT` should not be used, because the query then will run slower (many optimizers of current DBMS do not remove the unnecessary duplicate elimination).

Theoretitions sometimes recommend to write always "`DISTINCT`", but that shadows possible errors: When a query does produce duplicates, it is often helpful to understand why.

Algorithms for this error are discussed in Section 8.

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.

Of course, if a constant (datatype literal) is written as a term in the `SELECT`-list, this is obviously intended, and no warning should be given. Also `SELECT *` might yield a constant column, yet is shorter than listing the remaining columns explicitly, and again, no warning should be printed.

However, at least the following typical case should be caught: The conjunctive normal form of the `WHERE`-clause contains `A = c` with an attribute `A` and a constant `c` as one part of the conjunction, and `A` also appears as a term in the `SELECT`-list. Of course, if `A = B` appears in addition to `A = c`, also the value of `B` is obvious, and using `B` in the `SELECT`-list should generate a warning. And so on.

With a higher warning level, one should run a model generator, and for each output column `A` run the model generator again with the additional condition `A <> c`, where `c` is the value of `A` in the first model. This would ensure that every output column can generate at least two different values (in possibly different database states).

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

2.3 FROM Clause: Unnecessary Tuple Variables

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

Error 5: Unused tuple variable.
It is a simple syntax check to ensure that all tuple variables declared under `FROM` are really accessed somewhere in the query. If this condition is violated, one would typically also get Error 27 (missing join condition), but since "forgetting" a declared tuple variable happens quite often (it is Number 5 on our Top 10 list), a more specific error message might be preferable. (See also the discussion about equivalence at the beginning of this section.)

Error 6: Unnecessary join.
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.

Since joins are an expensive operation, there is a large body of literature about join elimination in query optimization see, e.g., [2,25]. However, at least one widely used commercial system does not seem to do a join elim-

ination. Furthermore, one could argue that this is not the purpose of the query optimizer, at least if the join explicitly appears in the query (and not as part of a view definition that is used in the query). An optimizer might well work with the principle "garbage in, garbage out": It is not the purpose of the query optimizer to correct user errors.

But even when the query optimizer solves the performance problem, the query will look simpler with one fewer tuple variable, therefore that formulation is preferable. This error is number 4 on our Top 10 list.

Another case of unnecessary join is that two tuple variables are declared over the same relation, but one can merge them without changing the semantics of the query. In this case, the unnecessary join is very likely an indication for a real error. An extreme case of this is the next error type.

Error 7: Tuple variables 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. In a higher warning level, one could use a model generator to check for every pair of distinct tuple variables over the same relation that there are solutions with different values for the primary key.

Of course, this is formally a special case of Error 6 (unnecessary join), and it leads often to Error 1 (inconsistent condition). However, since it still appears quite often, it deserves a more specific warning.

2.4 WHERE Clause

Error 8: Implied, tautological, or inconsistent subcondition.
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.

This condition becomes even more strict if it is applied not to the given formula, but to the DNF of the formula. Then the check for unncessary logical complications can be easily reduced to a series of consistency tests. Let the DNF of the `WHERE` condition be $C_1 \vee \cdots \vee C_m$ with $C_i = (A_{i,1} \wedge \cdots \wedge A_{i,n_i})$. Then no part of the condition is unnecessary 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) $\neg A_{i,j} \wedge (C_i - \{A_{i,j}\}) \wedge \neg(C_1 \vee \cdots \vee C_{i-1} \vee C_{i+1} \vee \cdots \vee C_m)$ for $i = 1, \ldots, m$, $j = 1, \ldots, n_i$, where $(C_i - \{A_{i,j}\}) = A_{i,1} \wedge \cdots \wedge A_{i,j-1} \wedge A_{i,j+1} \wedge \cdots \wedge A_{i,n_i}$ (otherwise $A_{i,j}$ could be replaced by `TRUE`).

Let us explain this a bit. There are really six cases to consider:

(a) The entire formula $C_1 \vee \cdots \vee C_m$ can be replaced by `FALSE`, i.e. it is inconsistent. This is Error 1 above.

(b) The entire formula $C_1 \vee \cdots \vee C_m$ can be replaced by `TRUE`, i.e. it is a tautology. This is checked with condition (1) above: The negation of the formula must have a model.

(c) One of the C_i can be replaced by `FALSE`, i.e. removed from the disjunction. For example, `SAL > 500 OR SAL > 700` is equivalent to `SAL > 500`. This is checked with condition (2) above: For each C_i, there must be a model in which only C_i is true, but all other C_j, $j \neq i$, are false.

(d) One of the C_i can be replaced by `TRUE`. In this case, the entire disjunction can be replaced by `TRUE`, thus condition (1) checks also this case.

(e) One of the $A_{i,j}$ can be replaced by `FALSE`. In this case, the entire conjunction $C_i = A_{i,1} \wedge \cdots \wedge A_{i,n_i}$ can be replaced by `FALSE`. Thus condition (2) also checks this case.

(f) One of the $A_{i,j}$ can be replaced by `TRUE`, i.e. it can be removed from the conjunction. For instance, consider

`(SAL < 500 AND COMM > 1000) OR SAL >= 500`.

This can be simplified to "`COMM > 1000 OR SAL >= 500`", i.e. the underlined condition can be replaced by `TRUE`. Checking for such cases is the purpose of condition (3): It must be possible that $A_{i,j}$ makes the conjunction C_i false, when the rest of the conjunction is true, and the rest of the disjunction is false. This ensures that the truth value "`FALSE`" of $A_{i,j}$ is really important.

Error 9: Comparison with NULL.

At least in Oracle, it is syntactically valid to write `A = NULL`, but this condition has a constant truth value (null/unknown). In other systems, this would be a syntax error.

Error 10: NULL value in IN/ANY/ALL subquery.

`IN` conditions and quantified comparisons with subqueries (`ANY`, `ALL`) can have a possibly surprising behaviour if the subquery returns a null value (among other values). An `IN` and `ANY` condition is then null/unknown, when it otherwise would be false, and an `ALL` condition is null/unknown, when it otherwise would be true (`ALL` is treated like a large conjunction, and `IN`/`ANY` like a large disjunction). For instance, the result of the following query is always empty as there exists one employee who is not a subordinate (`MGR IS NULL` for the president of the company):

```
SELECT X.EMPNO, X.ENAME, X.JOB
FROM   EMP X
WHERE  X.EMPNO NOT IN (SELECT Y.MGR
                       FROM   EMP Y)
```

This is a trap that many SQL programmers are not aware of. Indeed, it is quite common to assume that the above query is equivalent to

```
SELECT X.EMPNO, X.ENAME, X.JOB
FROM   EMP X
WHERE  NOT EXISTS (SELECT *
                   FROM   EMP Y
                   WHERE  Y.MGR = X.EMPNO)
```

This equivalence does not hold: The `NOT EXISTS`-query would work as intended. This problem is also mentioned in [7].

A warning should be printed whenever there is a database state in which the result of such a subquery includes a null value. We could also heuristically assume that if a column is not declared as `NOT NULL`, then every typical database state contains at least one row in which it is null. Now a query would be strange if it returns an empty result for all typical database states. (Of course, it might be that the programmer searches for untypical states, but then he/she could use a comment to switch off the warning).

Error 11: Unnecessarily general comparison operator.

Consider the query:

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

We suggest to give a warning, whenever `<=` or `>=` in a condition can be replaced by `=`, and the resulting query is equivalent to the original query. In the example, the case `>` can never happen. If one views `A >= B` as an abbreviation of `A > B OR A = B`, this would be an instance of Error 8 (inconsistent subcondition). In the same way, it would make sense to replace e.g. `>=` by `>` if this does not change the semantics of the query. However, here the improvement in readability is not so big.

Another variant of an unnecessary general operator is to use `IN` here instead of `=`. Remember that `IN` is the same as `= ANY`. When a subquery returns exactly one result in all database states, quantified comparisons (`ANY`/`ALL`) should be replaced by the corresponding simple comparison.

We have also seen (in semi-professional software)

```
A LIKE '%'
```

where "`A IS NOT NULL`" has the same function.

Error 12: LIKE without wildcards.

If `LIKE` is used without wildcards "`%`" and "`_`", it can and should be replaced by "`=` (there is a small semantic difference with blank-padded vs. non blank-padded comparison semantics in some systems). This could be seen as a special case of Error 11, but it is so common that it should be treated separately.

Error 13: Unnecessarily complicated SELECT in EXISTS-subquery.
In EXISTS-subqueries, the SELECT list is not important. Therefore, it should be something simple (e.g. "*" or "1" or a single attribute). Also using DISTINCT would be strange.

Error 14: IN/EXISTS condition can be replaced by comparison.
Consider the following query:

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

The WHERE-condition can be equivalently replaced by `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 Aggregation Functions

Error 15: Unnecessary aggregation function.
An aggregation function is unnecessary if it has only a single distinct input value (and is not sensitive to duplicates or duplicates do not occur or are explicitly eliminated). An example is:

```
SELECT   MAX(SAL)
FROM     EMP
GROUP BY SAL
```

This query is equivalent to

```
SELECT DISTINCT SAL
FROM   EMP
```

(See also Error 22.) However, in seldom cases, the aggregation function might be needed because of syntactic restrictions of SQL (in aggregation queries, only GROUP BY-attributes can be used in the SELECT-list outside of aggregation functions). But then, it would be clearer to add the attribute to the GROUP BY clause (this would not change the groups since the attribute has a unique value within each group). At least, there should be a comment for such unusual situations, which can also switch off the warning.

Error 16: Unnecessary DISTINCT in aggregation function.
MIN and MAX never need DISTINCT. When DISTINCT is used in other aggregation functions, it might not be necessary because of keys. See also Error 33.

Error 17: 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. That might be a matter of taste, but at least when counting duplicates is important, the meaning of the query is obscured by a `COUNT` argument, e.g.

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

In this example, only duplicates are counted.

2.6 GROUP BY Clause

Error 18: Unnecessary GROUP BY in EXISTS subquery.
In `EXISTS` subqueries, the `GROUP BY` clause is unnecessary if it is used without `HAVING` (the grouping has no influence on the existence of rows). If the `GROUP BY` should be syntactically necessary because of aggregations in the `SELECT`-clause of the subquery, this is an instance of Error 13.

Error 19: GROUP BY with singleton groups.
If it can be proven that each group consists only of a single row, the entire aggregation is unnecessary.

Error 20: GROUP BY with only a single group.
If there is always only a single group, the `GROUP BY` clause is unnecessary, except when the `GROUP BY` attribute should be printed under `SELECT`.

Error 21: Unnecessary GROUP BY attribute.
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.

Error 22: GROUP BY can be replaced by DISTINCT.
If exactly the `SELECT`-attributes are listed under `GROUP BY`, and no aggregation functions are used, the `GROUP BY` clause can be replaced by `SELECT DISTINCT` (which is shorter and clearer).

2.7 HAVING Clause

In the `HAVING`-clause, the same errors as in the `WHERE`-clause are possible. See also Error 25 below.

2.8 UNION/UNION ALL

Error 23: UNION can be replaced by OR.
If the two `SELECT`-expressions use the same `FROM`-list the same `SELECT`-list, and mutually exclusive `WHERE`-conditions, `UNION ALL` can be replaced by a single query with the `WHERE`-conditions connect by `OR`. There are similar conditions for `UNION` (one must be careful with null values here).

2.9 ORDER BY Clause

Error 24: Unnecessary ORDER BY term.
Suppose that the order specification 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, the programmer should help the system to execute the query efficiently. Most of the unnecessary complications above also lead to a longer runtime, if the optimizer does not discover them (e.g., errors 2, 5, 6, 7, 8, 12, 14, 16, 18, 19, 20, 23). In the following two cases the query does not get shorter by choosing the more efficient formulation.

Error 25: 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`. An example is (all our examples actually appeared in exams or homeworks, maybe with a different database schema):

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

Doing the join in this way under `HAVING` is possible, but has awful performance: The entire cartesian product must be constructed and sorted to execute the `GROUP BY`.

Error 26: Inefficient UNION.
`UNION` should be replaced by `UNION ALL` if the results of the two queries are always disjoint, and that none of the two queries returns duplicates.

4 Violations of Standard Patterns

Another indicator for possible errors is the violation of standard patterns for queries.

Error 27: Missing join condition.
Missing join conditions are a type of semantic error that is mentioned in most text books. However, it is seldom made precise how such a test should be formally done. The following is a strict version: First, the conditions is converted to DNF, and the test is done for each conjunction separately. One creates a graph with the tuple variables X as nodes. Edges are drawn between tuple variables for which a foreign key is equated to a key, except in the case of self-joins, where any equation suffices. 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.

Another exception are tuple variables over relations that can contain only a single tuple. Unfortunately, standard SQL does not permit to declare this constraint (it would be a key with 0 columns).

It might seem very strict to count only key-foreign key equations, but one of our programs (that was actually used for managing homework points) contained an error because a tuple variable had a composed key and only one of the attributes was joined. This error would have been detected by the above test.

Note that in some cases, joins can also be done via subqueries. Thus, tuple variables in subqueries should be added to the graph.

Error 28: Uncorrelated EXISTS-subquery.
If an `EXISTS`-subquery makes no reference to a tuple variable from the outer query, it is either globally true or globally false. This is a very unusual behaviour. Actually, uncorrelated `EXISTS`-subqueries are simply missing join conditions (possibly for anti-joins).

Error 29: IN-Subquery with only one possible result value.
Something like the following puzzled a team of professional 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. In this particular example, also missing join condition should have been detected, but that is not always the case. One would expect that the `SELECT`-clause of a subquery uses a tuple variable of the subquery (this applies not only to `IN`-subqueries, but also subqueries used as terms). This situation is also strange because the `IN`-subquery can return only a single possible value (for each given binding of the tuple variables of the outer query). In such cases, it would be more natural to use `EXISTS`.

Error 30: Condition in the subquery that can be moved up.
A condition in the subquery that accesses only tuple variables from the main query is strange.

Error 31: Comparison between different domains.
If domain information is available, a comparison between attributes of different domains is suspicious. This is another reason why in the design phase, domains should be defined, even if they are not directly supported in the DBMS. If there is no domain information, one could analyze an example database state for columns that are nearly disjoint. If that is not possible, at least comparisons between strings and numbers of different size (e.g., a `VARCHAR(10)` column and a `VARCHAR(20)` column) is suspicious.

Error 32: Strange HAVING.
Using `HAVING` without a `GROUP BY` clause is strange: Such a query can have only one result or none at all.

Error 33: DISTINCT in SUM and AVG.
For the aggregation functions `SUM` and `AVG`, duplicates are most likely significant.

Error 34: 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 35: Condition on left table in left outer join condition.
SQL permits quite arbitrary conditions as join conditions, e.g. the following is syntactically valid:

```
SELECT D.DNAME, COUNT(E.EMPNO)
FROM   DEPT D LEFT OUTER JOIN EMP E
       ON D.LOC = 'NEW YORK' AND D.DEPTNO = E.DEPTNO
```

However, many programmers will be surprised to find that departments outside New York are printed, but with 0 employees. Conditions on the left table in the left outer join condition are almost always an error. They exclude all possible join partners in the right table without any condition on the right table.

Note, however, that conditions on the right table do make sense in the left outer join condition. Such conditions are problematic in the WHERE-clause, as the next case shows.

Error 36: Outer join can be replaced by inner join.
Conditions on attributes of the right table of a left outer join are usually an error if the appear under WHERE (they might make sense under ON):

```
SELECT M.ENAME, M.SAL, E.ENAME, E.SAL
FROM   EMP M LEFT OUTER JOIN EMP E
       ON M.EMPNO = E.MGR
WHERE  E.SAL > M.SAL
```

The WHERE-condition has the truth value unknown/null when E.SAL is null. Therefore all tuples specifically generated by the outer join are eliminated by the WHERE-condition. Thus, the query is equivalent to one that uses a standard (inner) join.

5 Duplicates

Error 37: Many 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.)
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
```

However, duplicates are not always bad. Consider, e.g.:

```
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 for identifying employees. We call such attributes or attribute combinations "soft keys" (they are used in the real world to identify objects, although in exceptional situations, they might be not unique).

The specification of soft keys would be helpful for discovering errors. (More generally, it would be good to know what are "normal"/"typical" database states. E.g. "violations" of soft keys are possible, but unusual.)

We suggest to print a warning about duplicates whenever the query can produce duplicate answers in a database state that does not contain duplicates in soft keys (i.e. a database state that satisfies them like real keys, i.e. a database state that is not exceptional).

If there is no information about soft keys, one could run the query on an example database state. If it produces a lot of duplicates, we could give a warning. Techniques developed in query optimization for estimating the result size can also be used.

Error 38: DISTINCT that might remove important duplicates.

Conversely, applying `DISTINCT` or `GROUP BY` sometimes removes important duplicates. Consider the following query:

```
SELECT DISTINCT M.ENAME
FROM   EMP M, EMP E
WHERE  E.MGR = M.EMPNO AND E.JOB = 'ANALYST'
```

The purpose of this query seems to find employees who have an analyst in their team. If it should ever happen that two different employees with the same name satisfy this condition, only one name is printed. Since the intention is to find employee objects, this result is at least misleading.

We suggest to give a warning whenever a soft key of a tuple variable appears under `SELECT DISTINCT` or `GROUP BY`, but not the corresponding real key. This could also be seen as a violation of a standard pattern.

6 Possible Runtime Errors

Runtime errors are detected by the DBMS while it executes the query. Often they occur only in some database states, while the same query runs well for other data. A query should be considered as problematic if there is at least one state in which the error occurs.

Error 39: Subquery term that might return more than one tuple.

If one uses a subquery as a term, e.g. in 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.

As usual for runtime errors, the occurrence of the error depends on the evaluation sequence. For instance, consider the following query:

```
SELECT DISTINCT E1.ENAME
FROM   EMP E1, EMP E2
WHERE  'NEW YORK' = (SELECT LOC FROM DEPT D
                     WHERE  D.DEPTNO = E1.DEPTNO
                        OR  D.DEPTNO = E2.DEPTNO)
AND    E1.EMPNO = E2.EMPNO
```

If the underlined condition is evaluated before the subquery, there is no problem. Otherwise the runtime error occurs. The SQL-92 standard does not clarify this point. 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, it seems safest to require that the subquery returns only a single tuple for each assignment of the tuple variables in the outer query, no matter whether that assignment satisfies the conditions of the outer query or not. We will show how to check this in Section 8.

Error 40: SELECT INTO that might return more than one tuple.
If `SELECT ... INTO ...` in embedded SQL should ever return more than one tuple, a runtime error occurs. For instance, the following query will work until it happens that there are two employees with the same name `:N`.

```
SELECT JOB, SAL
INTO   :X, :Y
FROM   EMP
WHERE  ENAME = :N
```

Since the critical situation is so unusual, it might be overlooked during testing. The above query is only safe if `ENAME` is declared as key of `EMP`.

Error 41: No indicator variable for argument 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 (e.g. `SUM` over the empty set is null, not `0`).

Error 42: Difficult type conversion.
Also, the very permissive type system of at least Oracle SQL can pose a problem: Sometimes strings are implicitly converted to numbers, which can generate runtime errors.

Error 43: Runtime error in datatype function.
Datatype operators have the usual problems (e.g. division by zero). It is difficult for the SQL programmer to really avoid this, e.g. this is still unsafe:

```
SELECT ENAME
FROM   EMP
WHERE  COMM <> 0 AND SAL/COMM > 2
```

SQL does not guarantee any specific evaluation sequence. A declarative solution would be to extend SQL's three-valued logic by a truth value "error", and to evaluate e.g. "error and false" to false. We do not know whether this is done in any DBMS. Thus, our test should print a warning whenever in a term A/B appearing under `WHERE`, B can be zero, no matter what other conditions there are. However, if the term appears under `SELECT`, one can assume that the `WHERE`-condition is satisfied.

Updates could also cause runtime errors, e.g. inserting a null value into a `NOT NULL` column. In general, the program logic around an update should avoid constraint violations. But this is beyond the scope of the present paper.

7 Style Checks

In all of the above situations, there are quite strong indications that the query will not (always) behave as intended, or is at least more complex than it needs to be. The following situations are a matter of taste.

(1) A tuple variable in the main query should not be "shadowed" by a tuple variable of the same name in a subquery.
(2) A tuple variable from an outer query should not be accessed without its name in a subquery (i.e. only "`A`" instead of "`X.A`").
(3) One should also use "`X.A`" instead of "`A`" to refer to a tuple variable "`X`" of the subquery, if the outer query has a tuple variable "`Y`" with the same attribute "`A`", and "`Y`" is accessed in the subquery.
(4) Some style guides even recommend to use a tuple variable in every attribute reference (when more than one tuple variable is declared), even if the attribute name alone would uniquely determine the tuple variable. This makes the structure of the query understandable without knowing the underlying database schema.
(5) An `IN`-subquery that is correlated, i.e. accesses tuple variables from the outer query, should probably be replaced by an `EXISTS`-subquery.
(6) It might be a matter of taste, but a large number of unnecessary parentheses is also not helpful for reading the query.
(7) Language defaults and optional keywords should be used consistently. E.g. it would be strange to declare one tuple variable with "`EMP AS E`",

and another tuple variable simply with "`DEPT D`".

(8) Identifiers should be written with consistent case although SQL is case-insensitive.

(9) In SQL queries that are embedded into programs, it might be good to avoid `SELECT *` (except in `EXISTS`-subqueries): New columns can be added to tables (with `ALTER TABLE`), and then this query would no longer match the result variables in the program.

(10) Using `SELECT DISTINCT` in `IN`-subqueries is not necessary: It is formally equivalent to `SELECT`. However, some programmers think that in some systems this might improve performance. Therefore, it might be a matter of taste.

(11) Of course, SQL queries should be portable between different DBMS. However, there is a tradeoff between portability on the one hand, and conciseness and efficiency on the other hand.

Sometimes, the distinction between an indication for a semantic error and a style recommendation is not completely clear. For instance, Error 13 (unnecessarily complicated `SELECT` in `EXISTS`-subquery) and Error 17 (unnecessary argument of `COUNT`) could be seen as a matter of taste.

Conversely, inconsistent use of defaults might also indicate a possible misconception about the language: For instance, consider

```
ORDER BY A, B ASC
```

If the user really wants to make the default order explicit, he/she should write "`ORDER BY A ASC, B ASC`". It is also unusual to declare an explicit tuple variable, but then reference it only implicitly via unique attribute names.

8 Algorithms for Selected Semantic Errors

In the error descriptions above, we have already sketched algorithms for those errors that can be discovered with simple syntactic checks, and given references to the literature where algorithms for some errors can be found.

However, there is a number of errors that need a check for duplicate solutions in one way or another. In this section, we want to look at two representative cases: Error 2 (unnecessary `DISTINCT`) and Error 39 (subquery term that might return more than one tuple).

Suppose the following query is given, and we want to check whether `DISTINCT` is indeed necessary:

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

In this section, we exclude aggregation functions. Furthermore, we assume that all attribute references include a tuple variable (this can easily be reached).

Now the question whether DISTINCT is necessary in the above query can be reduced to a consistency check by duplicating the tuple variables in order to find two different variable assignments that both satisfy the WHERE-condition, and yield the same values for the SELECT-terms:

```
SELECT *
FROM   R1 X1, ..., Rn Xn, R1 X1', ..., Rn Xn'
WHERE  φ AND φ'
AND    (t1 = t1' OR (t1 IS NULL AND t1' IS NULL))
...    ...
AND    (tm = tm' OR (tm IS NULL AND tm' IS NULL))
AND    (X1 ≠ X1' OR ··· OR Xn ≠ Xn')
```

Here φ' results from φ by replacing every occurence of X_i by X_i'. The same for the t_j. 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 duplicates (if the condition φ is consistent).

The DISTINCT is necessary in the original query if and only if this modified query is consistent (i.e. has a nonempty result in at least one database state that respects all constraints). The consistency is not decidable in general, but the reduction is possible in both directions, so nothing is lost. But one can utilize now the rich literature on consistency tests, e.g., [4,5,20,24,16,28,21,3] (and of course all literature on automated theorem proving, e.g. [11]).

Such a simple reduction to a consistency test is possible for nearly all non-syntactic errors above. E.g., Errors 19, 20, 37, 39, 40 can obviously be solved with very similar techniques, and the method for Error 8 was already given in its description above. This means that all advances in consistency checking would immediately extend the applicability of our other tests. The consistency of SQL queries is decidable at least for all queries without aggregations and without datatype functions (such as +), and that have only a single level of subqueries. This corresponds to the quantifier prefix $\exists^*\forall^*$, for which it is well known that the consistency of first order logic with equality is decidable (it was proven 1928 by Bernays and Schönfinkel). In [3] we proposed an algorithm that can also handle null values and often deeper nested subqueries.

Let us consider a variation of the idea to show its general applicability. Suppose

the following subquery is used as a term, so we must make sure that Error 39 cannot occur, i.e. that it always returns only one tuple:

$$\begin{array}{l}\texttt{SELECT } t\\ \texttt{FROM } \quad R_1\,X_1\texttt{, } \ldots\texttt{, } R_n\,X_n\\ \texttt{WHERE } \quad \varphi\end{array}$$

The interesting point is that φ might access tuple variables $S_1\,Y_1, \ldots, S_m\,Y_m$ from the outer query. However, we explained above that in order to be really sure that the runtime error does not occur, nothing can be assumed about their values. Thus, we must check the following query for consistency:

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

Error 39 can occur if and only if this query is consistent.

Of course, doing a lot of consistency tests during the semantic check of a single query might be considered as too expensive. One can also easily develop efficient algorithms that are only sufficient or only necessary. As an example, let us consider again the question whether a query might produce duplicates (Error 2).

Sufficient test for error 2: Unnecessary DISTINCT.

Let the given query be:

$$\begin{array}{l}\texttt{SELECT DISTINCT } t_1\texttt{, } \ldots\texttt{, } t_m\\ \texttt{FROM } \quad R_1\,X_1\texttt{, } \ldots\texttt{, } R_n\,X_n\\ \texttt{WHERE } \quad \varphi\end{array}$$

(1) Let $\psi_1 \wedge \cdots \wedge \psi_k$ be the conjunctive normal form of φ.

(2) Let $\mathcal{K}$ be the set of those t_i that are an attribute reference (not a composed term or a constant). I.e. $\mathcal{K}$ is initialized with the attributes that appear as output column under `SELECT`. The idea is that $\mathcal{K}$ contains those attributes that have a unique value for every given output row.

(3) Add to $\mathcal{K}$ all attributes A such that $A = c$ or $c = A$ with a constant c appears among the ψ_i.

(4) While $\mathcal{K}$ differs from the last step do:
- Add to $\mathcal{K}$ attributes A such that $A = B$ or $B = A$ appears among the ψ_i and $B \in \mathcal{K}$.
- If $\mathcal{K}$ contains a key of a tuple variable X_i, add all other attributes of X_i to $\mathcal{K}$.

(5) If $\mathcal{K}$ contains a key of every tuple variable X_i, `DISTINCT` is superfluous. (If the query contains `GROUP BY`, one checks instead whether all `GROUP BY` columns are contained in $\mathcal{K}$.)

9 Some Statistics about Error Occurrence

The above list of error types is based on our experience from grading a large number of exams and homeworks. After this error taxonomy was finished, we analyzed the solutions of the SQL exercises in four exams of the course "Databases I" at the University of Halle (One exam in winter term 2002/03, a "second chance" exam in summer term 2003 for students who wanted to improve their grade, and midterm and final exam in winter term 2003/2004. The course was taught by two different professors.) The results are shown in Table 1. The exercises are numbered with letters, e.g. the first exam contained four exercises (A, B, C, D). Course material and exam exercises are available from the project web page (see Conclusions).

We did sometimes count several unrelated semantic errors in the same exercise, but that did not occur often. The number of exam solutions that contained at least one semantic error is the sum of the entries "Semantics" and "Both". Of course we counted only semantic errors from our above list, i.e. that are detectable without knowing the task of the query. "Wrong task" lists the number of exams that can only be detected as incorrect if the goal of the query is known. In syntactically relatively simple exercises (e.g., midterm exam in winter term 2003/04), there are many more detectable semantic errors than syntax errors. "Not Counted" are exams that did not try the particular exercise, or that contained so severe syntax errors that a detailed analysis was not possible. In the exams that were analyzed, the ten most often occurring semantic errors are (percentages are relative to all detected semantic errors):

1.	21,3 %	Error 27: Missing join condition
2.	11,4 %	Error 1: Inconsistent condition
3.	10,8 %	Error 37: Many duplicates
4.	8,4 %	Error 6: Unnecessary join
5.	5,6 %	Error 5: Unused tuple variable
6.	5,4 %	Error 8: Implied, tautological or inconsistent subcondition
7.	4,4 %	Error 19: Singleton groups
8.	3.7 %	Error 2: Unnecessary DISTINCT
9.	3.2 %	Error 3: Constant output column
9.	3.2 %	Error 7: Tuple variables are always identical

10 Related Work

It seems that the general question of detecting semantic errors in SQL queries (as defined above) is new. However, it is strongly related to the two fields of semantic query optimization and cooperative answering.

	2002/03				2003			Mid 03/04			Fin 03/04			
	67 participants				18 part.			153 part.			148 part.			
Error	A	B	C	D	E	F	G	H	I	J	K	L	M	$\sum$
1	-	-	2	1	8	-	-	3	4	14	1	14	2	49
2	4	-	-	-	-	-	1	2	-	5	2	2	-	16
3	8	-	-	4	-	1	-	-	-	-	-	1	-	14
4	-	-	-	-	-	-	-	-	-	2	-	-	-	2
5	1	-	-	-	-	-	-	-	-	2	11	9	1	24
6	-	-	-	-	-	2	-	-	-	25	2	5	2	36
7	-	-	-	-	-	-	-	5	-	8	-	-	1	14
8	-	1	3	-	3	-	-	2	2	6	4	1	1	23
9	-	-	-	-	1	-	-	-	6	-	-	-	-	7
11	-	-	1	-	-	-	-	-	-	-	1	-	3	5
12	-	-	-	-	1	-	-	2	3	2	-	-	-	8
14	-	-	-	-	-	-	-	-	-	-	2	2	1	5
15	-	-	2	2	-	-	-	-	-	-	-	-	-	4
19	-	-	-	-	-	-	-	-	-	-	1	2	16	19
20	-	-	-	-	-	-	-	-	-	-	1	-	-	1
21	-	1	1	-	-	-	-	-	-	-	1	1	1	5
22	1	-	1	-	-	-	-	-	-	-	-	1	-	3
23	-	-	-	-	-	-	-	1	-	-	-	-	-	1
25	-	-	1	1	-	-	-	-	-	-	3	-	1	6
27	14	17	14	6	4	1	2	1	3	29	2	5	-	91
28	-	-	-	5	1	-	-	-	-	-	-	6	-	12
30	1	-	-	2	-	-	-	-	-	-	1	3	-	7
31	-	-	-	1	-	-	-	-	5	1	-	-	-	7
34	-	-	-	-	-	-	-	11	-	-	-	-	-	11
37	-	-	-	-	-	-	-	-	46	-	-	-	-	46
39	6	2	3	1	-	-	-	-	-	-	-	-	-	12
Correct	22	2	10	-	-	6	5	104	37	30	18	45	54	27%
Syntax	16	20	18	20	5	6	6	7	17	9	86	30	57	24%
Semantics	19	5	7	6	4	3	1	20	50	50	8	31	18	18%
Both	9	14	14	12	9	1	2	6	11	23	17	15	7	11%
Wrong Task	1	7	3	15	-	1	2	12	30	9	6	23	8	10%
Not Counted	-	19	15	14	-	1	2	4	8	32	13	4	4	10%

Table 1. Error Statistics for Four Exams

Semantic query optimization (see e.g. [9,18,8]), also tries to find unnecessary complications in the query, but otherwise the goals are different. As far as we know, no system prints a warning 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., [15,12,14]). 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 described in [22,23] discovers semantic errors, too, but it 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.

Software testing techniques for database queries are developed e.g. in [6,28]. Further studies about errors in database queries, especially psychological aspects, are [27,26,19,13,10].

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.

We are currently developing a tool "`sqllint`" for finding semantic errors in SQL queries. The list of error types contained in this paper can serve as a specification for this tool. The current state of the project is reported at:

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

The current version of the prototype and the evaluated exams are posted on this page.

In future work, we especially want to investigate patterns for SQL queries in greater detail (by analyzing SQL queries from real projects).

Furthermore it is an interesting idea that for view definitions, the programmer might be willing to specify additional information like keys and foreign keys. Our tool could then try to verify that these constraints on the query result are indeed implied by the query and the constraints on the base table. In contrast, the present paper has assumed that the only input to the semantic check is the query itself and the database schema.

The authors would be thankful for reports about any further semantic errors or violations of good style that are not treated in this paper.

Acknowledgements

We would like to thank the following persons, who all made important contributions to this work. Sergei Haller and Ravishankar Balike told us about Error 29. Joachim Biskup contributed the idea that query size estimation techniques could be used. We had a very interesting and inspiring discussion with Ralph Acker about runtime errors. Elvis Samson developed the prototype of the consistency test. The discussions with Alexander Hinneburg have been very 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. Wolf Zimmermann helped us with compiler construction questions. Last but not least, without the students in our database courses, this work would have been impossible.

References

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

[2] A. V. Aho, Y. Sagiv, J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems*, 4:435–454, 1979.

[3] Stefan Brass, Christian Goldberg. *Detecting Logical Errors in SQL Queries*. Technical Report, University of Halle, 2004.

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

[5] François Bry, Hendrik Decker, and Rainer Manthey: A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *Proceedings of the International Conference on Extending Database Technology*, 488–505, 1988.

[6] M. Y. Chan and S. C. Cheung. Testing database applications with SQL semantics. *Proceedings of 2nd International Symp. on Cooperative Database Systems for Advanced Applications (CODAS'99)*, 364–375, 1999.

[7] Joe Celko: *Joe Celko's SQL for Smarties: Advanced SQL Programming, 2nd Ed.* Morgan Kaufmann, 1999.

[8] Qi Cheng, Jarek Gryz, Fred Koo, Cliff Leung, Linqi Liu, Xiaoyan Qian, and Bernhard Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. *Proceedings of the 25th VLDB Conference*, 687-698, 1999.

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

[10] Hock C. Chan. The relationship between user query accuracy and lines of code. *Int. Journ. Human Computer Studies 51*, 851-864, 1999.

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

[12] Wesley W. Chu, M.A. Merzbacher, and L. Berkovich. The design and implementation of CoBase. In *Proc. of ACM SIGMOD*, 517-522, 1993.

[13] Hock C. Chan, Bernard C.Y. Tan, and Kwok-Kee Wei. Three important determinants of user performance for database retrieval. *Int. Journ. Human-Computer Studies 51*, 895-918, 1999.

[14] 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.

[15] Terry Gaasterland, Parke Godfrey, and Jack Minker. An Overview of Cooperative Answering. Journal of Intelligent Information Systems 21:2, 123–157, 1992.

[16] 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.

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

[18] 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.

[19] W.J. Kenny Jih, David A. Bradbard, Charles A. Snyder, and Nancy G.A. Thompson. The effects of relational and entity-relationship data models on query performance of end users. *Int. Journ. Man-Machine Studies*, 31:257–267, 1989.

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

[21] Michael J. Minock. Knowledge Representation under the Schema Tuple Query Assumption. In *10th International Workshop on Knowledge Representation meets Databases (KRDB 2003)*.

[22] Antonija Mitrovic. A knowledge-based teaching system for SQL. In *ED-MEDIA 98*, pages 1027–1032, 1998.

[23] 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.

[24] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. Generating Consistent Test Data: Restricting the Search Space by a Generator Formula. *VLDB Journal*, 2:173–213, 1993.

[25] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, Val Tannen: A chase to far? In *Proc. of ACM SIGMOD*, 273-284, 2000.

[26] Antonio Rizzo, Sebastiano Bagnara, and Michele Visciola. Human error detecting processes. *Int. Journ. Man-Machine Studies 27*, 555-570, 1987.

[27] Charles Welty. Correcting user errors in SQL. *International Journal of Man-Machine Studies 22:4*, 463-477, 1985.

[28] Jian Zhang, Chen Xu and S.-C. Cheung, Automatic Generation of Database Instances for White-box Testing. In *Proc. of the 25th International Computer Software and Applications Conference (COMPSAC'01)*, Oct. 2001, IEEE Press, pp. 161-165.