Semantic Errors in SQL Queries: A Quite Complete List

Stefan Brass Christian Goldberg
Martin-Luther-Universität Halle-Wittenberg, D-06099 Halle (Saale), Germany
(brass|goldberg)@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 obviously not intended. However, current database management systems 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 database management systems will perform such checks and that the generated warnings will help to develop code with fewer bugs in less time.

1. Introduction

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

A semantic error means that a legal SQL query was entered, but the query does not 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. 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. 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 no 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.

The main contribution of this paper is a list of semantic errors that represents years of experience while cor-

recting 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.

The paper is structured by reasons why SQL queries can be considered 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. Section 8 contains some statistics how often errors appeared in exams. Related work is discussed in Section 9.

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

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

This is a simplified version of an example schema that comes with the Oracle DBMS.

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 (e.g. the Oracle 9i query optimizer does not seem to 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 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. 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. 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. Nobody would pose a query if he or she knew beforehand that the query result is empty, no matter what the database state is. In general, one could also construct other queries that have a constant result for all database states (maybe under the assumption that relations are not empty).

2.2. SELECT Clause

Error 2: Unnecessary DISTINCT. One should use an explicit duplicate elimination only when necessary. Because of keys it sometimes can be proven that a query cannot return duplicates. Then DISTINCT should not be used, because the query then will run slower (at least the Oracle 9i and DB2 V8.1 optimizers do not remove the unnecessary duplicate elimination). Always writing "DISTINCT" furthermore shadows possible errors: When a query does produce duplicates, it is often helpful to understand why.

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.

2.3. FROM Clause: Unnecessary Tuple Vars

Error 5: Unused tuple variables. (See the discussion about equivalence at the beginning of this section.)

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.

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.

2.4. 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.

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: 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 >=. Also writing IN here is quite confusing.

Error 11: 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 10, but it is so common that it should be treated separately.

Error 12: Unnecessarily complicated SELECT lists 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).

Error 13: 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')
```

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 14: Unnecessary DISTINCT in aggregations. 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 30.

Error 15: 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.

2.6. GROUP BY Clause

Error 16: 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 17: 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.

Error 18: 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.

Error 19: 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. In addition, conditions that are possible under WHERE are better written there (see Error 22 below).

2.8. UNION/UNION ALL

Error 20: 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.

2.9. ORDER BY Clause

Error 21: 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, 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, 11, 13, 14, 16, 17, 20). In the following two cases the query does not get shorter by choosing the more efficient formulation.

Error 22: 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`.

Error 23: Inefficient UNION. `UNION` should be replaced by `UNION ALL` if one can prove that 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 24: Missing join conditions. 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 very 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.

Error 25: 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. This is a very unusual behaviour. Actually, uncorrelated `EXISTS`-subqueries are simply missing join conditions (possibly for anti-joins).

Error 26: 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. In this particular example, also missing join condition should have been detected, but that is not always the case.

Error 27: Conditions 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 28: 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.

Error 29: Strange HAVING. Using `HAVING` without a `GROUP BY` clause is strange: Such a query can have only one result or none at all.

Error 30: DISTINCT in SUM and AVG. For the aggregation functions `SUM` and `AVG`, duplicates are most likely significant.

Error 31: 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.

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.

Error 32: Many duplicates. Consider this 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
```

But 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. Thus, we need a declaration of such "soft keys". Then we check whether there could possibly be duplicates under the assumption that these soft keys were real keys.

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.

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 33: Subqueries terms 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 (or pushed into the subquery), there is no problem. Otherwise the runtime error occurs. The SQL standard does not clarify this point, but 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.

Error 34: SELECT INTO that might return more than one tuple. If a `SELECT ... INTO ...` command in embedded SQL should ever return more than one tuple, a runtime error occurs.

Error 35: 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 36: Difficult type conversions. 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 37: Runtime errors in datatype functions. 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.

7. Style Checks

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. 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 might be a matter of taste, but a large number of unnecessary parentheses is also not helpful for reading the query.
5. 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.

8. Some Numbers about Error Occurrence

The above list is based on our experience from grading a large number of exams and homeworks. After this error taxonomy was finished, we analyzed the SQL exercises in midterm and final exam of the course "Databases I" at the University of Halle, winter term 2003/04. The results are shown in Figure 1. The midterm exam had 153 participants (exercises A, B, C), and the final exam hat 148 participants (exercises D, E, F). 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 exams that contained at least one semantic error is the sum of the entries "Only 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. As can be seen, in the syntactically relatively simple exercises of the midterm exam, there are many more exams with detectable semantic errors than with syntax errors. This demonstrates that the tool we are developing will be useful. "Not Counted" lists exams that did not try the particular exercise, or that contained so severe syntax errors that looking at semantic errors in detail was not possible. In the exams that were analyzed with this error taxonomy, the most often occurring semantic errors are (percentages are relative to all detected semantic errors):

15,2 %	32. Many Duplicates
13,2 %	24. Missing Join Conditions
12,5 %	1. Inconsistent Conditions
11,2 %	6. Unnecessary Joins
7,6 %	5. Unused Tuple Vars
6,3 %	16. Singleton Groups
5,3 %	8. Implied etc. Subconditions
4,6 %	7. Always Identical Tuple Vars

Error	A	B	C	D	E	F	$\sum$
1	3	4	14	1	14	2	38
2	2	-	5	2	2	-	11
3	-	-	-	-	1	-	1
4	-	-	2	-	-	-	2
5	-	-	2	11	9	1	23
6	-	-	25	2	5	2	34
7	5	-	8	-	-	1	14
8	2	2	6	4	1	1	16
9	-	6	-	-	-	-	6
10	-	-	-	1	-	3	4
11	2	3	2	-	-	-	7
13	-	-	-	2	2	1	5
16	-	-	-	1	2	16	19
17	-	-	-	1	-	-	1
18	-	-	-	1	1	1	3
19	-	-	-	-	1	-	1
20	1	-	-	-	-	-	1
22	-	-	-	3	-	1	4
24	1	3	29	2	5	-	40
25	-	-	-	-	6	-	6
27	-	-	-	1	3	-	4
28	-	5	1	-	-	-	6
31	11	-	-	-	-	-	11
32	-	46	-	-	-	-	46
Correct	104	37	30	18	45	54	32%
Only Syntax	7	17	9	86	30	57	23%
Only Semantics	20	50	50	8	31	18	19%
Both	6	11	23	17	15	7	9%
Wrong Task	12	30	9	6	23	8	7%
Not Counted	4	8	32	13	4	4	10%

Figure 1. Error Statistics for Two Exams

9. 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.

Semantic query optimization (see e.g. [3, 9, 2]), 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., [8, 5, 7]). 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 [11, 12] 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.

Further studies about errors in database queries, especially psychological aspects, are [14, 13, 10, 6, 4].

10. 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 of the task of this tool. We have algorithms for all of the error types (for a suitable SQL subset), but for space reasons, we could not present them here (see, however, the technical report [1]). The error checks can often be reduced to a consistency test. While the undecidability in the general case remains, we can at least use the mature methods of automated theorem proving. We also have simpler, direct sufficient conditions for some of the errors. The current state of the project is reported at:

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

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

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 26. 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.

References

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

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