# Detecting Semantic Errors in SQL Queries

Stefan Brass, Christian Goldberg, and Alexander Hinneburg

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
(brass|goldberg|hinneburg)@informatik.uni-halle.de

**Abstract.** We investigate classes of SQL queries which are syntactically correct, but certainly not intended, no matter for which task the query was written. For instance, queries that are contradictory, i.e. always return the empty set, are quite often written in exams of database courses. Current database management systems, e.g. Oracle, execute such queries without any warning. In this paper, we explain serveral classes of such errors, and give algorithms for detecting them. Of course, questions like the satisfiability are in general undecidable, but our algorithm can treat a significant subset of SQL queries. 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.

For instance, consider the following query:

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

This is a legal SQL query, and it is executed e.g. in the Oracle8i DBMS without any warning. However, the condition is actually inconsistent, so the query result will be always empty. Since nobody would use a database in order to get an always empty result, we can state that this query is incorrect without actually knowing what the task of the query was. Such cases do happen, e.g. in one exam exercise that we analyzed, 10 out of 70 students wrote an inconsistent condition.

It is well known that the consistency of formulas is undecidable, and that this applies also to database queries. E.g. one can write a query that checks whether the database contains a solution to a Post's correspondence problem, see [1], Section 6.3. This query does not contain any datatype operations.

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

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

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

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

```
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`. Although the tuple variable declared in the subquery has no attribute `EMPNO`, no error is generated, and the tuple variable in the outer query is accessed. However, a test for missing join conditions would have revealed the error.

Also in some contexts, SQL queries or subqueries must return not more than one row. Otherwise a runtime error is generated, and the application is terminated. Certainly it would be good to prove that all queries in an application program can never violate this condition.

## 2 Related Work

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

Actually, Oracle's precompiler for Embedded SQL (Pro*C/C++) has an option for semantic checking, but this means only that it checks whether tables and columns exist and that the types match. Also "Trouble Checker" from `http://www.msqlproducts.com` claims semantic checking, by it concentrates on procedures and triggers, e.g. it finds loops in triggers. These checks do not cover semantic errors in the most important declarative part of SQL.

The need of semantically rich error and warning messages for SQL statements in a learning context has been investigated in [11, 12]. However, the SQL Tutor system proposed there has knowledge about the task that has to be solved (in form of a correct query). In contrast, our approach assumes no such knowledge, which makes it applicable also for software development, not only for teaching.

Of course, for the special problem of detecting inconsistent conditions, a large body of work exists in the literature. In general, all work in automated theorem proving can be applied (see, e.g., [6]). The problem whether there exists a contradiction in a conjunction of inequalities is very relevant for many database problems and has been intensively studied in the literature. Klug's classic paper [10] checks for such inconsistencies but does not treat subqueries and assumes dense domains for the attributes. The algorithm in [8] can handle recursion, but only negations of EDB predicates, not general `NOT EXISTS` subqueries. A very efficient method has been proposed by Guo, Sun, Weiss [7]. We use it here as a subroutine. Our main contribution is the way we treat subqueries. Although this uses ideas known from Skolemnization, the way we apply it combined with an algorithm like [7] seems new. We also can handle integrity constraints and null values.

Consistency checking in databases has also been applied for testing whether a set of constraints is satisfiable. A classic paper about this problem is [2] (see also [3]). They give an algorithm which terminates if the constraints are finitely satisfiable or if they are unsatisfiable, which is the best one can do. However, the approach presented here can immediately tell whether it can handle the given query and constraints. If it cannot, one can start the [2] algorithm, but otherwise our algorithm is probably faster and termination is guaranteed.

The use of semantic knowledge about SQL statements has also been studied in the context of semantic query optimization. E.g. in the work of [4, 5], integrity constraints were used to transform queries into a form that can be answered more efficiently. The approach by Hsu and Knoblock [9] learns semantic rules from user queries and uses these rules for further query optimization. There is a strong connection of semantic query optimization to detecting semantic errors, but the goals are different. As far as we know, DB2 contains some semantic query optimzation, but prints no warning message if the optimizations are "too good to be true". Also the effort for query optimization must be amortized when the query is executed, whereas for error detection, we would be willing to spend more time. Finally, soft constraints (that can have exceptions) can be used for generating warnings about possible errors. but not for query optimization.

## 3 Inconsistent Conditions

In this section, we present an algorithm for detecting inconsistent conditions in SQL queries. Since the problem is in general undecidable, we can handle only a subset of all queries. However, our algorithm is reasonably powerful and can decide the consistency of surprisingly many queries. To be precise, consistency in databases means that there is a finite model, i.e. a relational database state (sometimes called a database instance), such that the query result is not empty.

In this paper, we assume that the given SQL query contains no datatype operations, i.e. all atomic formulas are of the form $t_1 \, \theta \, t_2$ where $\theta$ is a comparison operator (`=`, `<>`, `<`, `<=`, `>`, `>=`), and $t_1, t_2$ are attributes (possibly qualified with a tuple variable) or constants (literals). It should be quite easy to extend it at least to linear equations. Null values and `IS NULL` are treated in Section 3.5, before that, they are excluded. Aggregations are not treated in this paper, they are subject of our future research.

### 3.1 Conditions Without Subqueries

If that the query contains no subqueries, the consistency can be decided with methods known in the literature, especially the algorithms of Guo, Sun and Weiss [7].

The condition then consists of the above atomic formulas connected with `AND`, `OR`, `NOT`. We first push negation down to the atomic formulas, where it simply "turns around" the comparison operator, so it is eliminated from the formula. Then, we translate the formula in disjunctive normal form: $\varphi_1 \vee \cdots \vee \varphi_n$ is consistent iff at least one of the $\varphi_i$ is consistent.

Now a conjunction of the above atomic formulas can be tested for satisfiability with the method of [7]. They basically create a directed graph in which nodes are labelled with "Tuplevariable.Attribute" (maybe a representative for an equivalence class with respect to $=$) and edges are labelled with $<$ or $\leq$. Then they compute an interval of possible values for each node. Note that SQL data types like `NUMERIC(1)` also restrict the interval of possible values.

Unfortunately, if there are only finitely many values that can be assigned to nodes, inequality conditions ($t_1$ `<>` $t_2$) between the nodes become important and can encode graph-coloring problems. Therefore, we cannot expect an efficient algorithm if there are many `<>`-conditions. Otherwise, the method of [7] is fast. (However, the DNF computation that we apply before [7] can lead to an exponential increase in size.)

### 3.2 Subqueries

For simplicity, we treat only `EXISTS` subqueries. Other kinds of subqueries (`IN`, `>=ALL`, etc.) can be reduced to the `EXISTS` case. E.g. Oracle performs such a query rewriting before the optimizer works on the query.

**Definition 1.** *Given a query $Q$, let us call a tuple variable in $Q$ existential if it is declared in a subquery that is nested inside an even number of* `NOT`*s, and universal otherwise. For instance, the tuple variables in the outermost (main) query are existential.*

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

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

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

In automated theorem proving (see, e.g., [6]), it is a well-known technique to eliminate existential quantifiers by introducing Skolem constants and Skolem functions. This simply means that a name is given to the tuples that are required to exist. For tuple variables that are not contained in the scope of a universal quantifier (such as `L` in the example), a single tuple is required in the database state. However, for an existential tuple variable like `E` that is declared within the scope of a universal tuple variable (`D`) a different tuple might be required for every value for `D`. Therefore, a function $f_{\mathtt{E}}$ is introduced that takes a value for `D` as a parameter and returns a value for `E`. Such a function is called a Skolem function. There is also a Skolem function $f_{\mathtt{L}}$ for `L`, but this function has no parameters (it is a Skolem constant).

Let us make precise what parameters $Y$ the Skolem function $f_X$ for a tuple variable $X$ must have:

**Definition 2.** *An existential tuple variable $X$ depends on a universal tuple variable $Y$ iff*

1. *the declaration of $X$ appears inside the scope of $Y$, and*
2. *$Y$ appears in the subquery in which $X$ is declared (including possibly nested subqueries).*

The second part of the condition is not really required, but it reduces the number of parameters which will help us to handle more queries ("arity reduction" for Skolem functions is also a known technique in automated theorem proving).

In contrast to the classical case of automated theorem proving, we use a "sorted" logic: Each tuple variable can range only over a specific relation. Therefore our Skolem functions have parameter and result types. E.g. the function $f_{\mathtt{E}}$ in the example assumes that a tuple from the relation `DEPT` is given, and returns a tuple from the relation `EMP`.

**Definition 3.** *Given a query $Q$, a set of sorted Skolem constants and functions $\mathcal{S}_Q$ is constructed as follows: For each existential tuple variable $X$ ranging* *over relation $R$, a skolem constant/function $f_X$ of sort $R$ is introduced. Let $Y_1, \ldots, Y_n$ be all universal tuple variables, on which $X$ depends, and let $Y_i$ range over relation $S_i$. Then $f_X$ has $n$ parameters of sort $S_1, \ldots, S_n$.*

In the example, there is one Skolem constant and one Skolem function:

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

**Definition 4.** *Given a query $Q$, and a relation $R$, let $\mathcal{T}_Q(R)$ be the set of all terms of sort $R$ that can be built from the constants and function symbols in $\mathcal{S}_Q$ respecting the sorts. Let $\mathcal{T}_Q$ be the union of the $\mathcal{T}_Q(R)$ for all relation symbols $R$ appearing in $Q$.*

In Example 1, $\mathcal{T}_Q = \{f_{\texttt{L}},\ f_{\texttt{E}}(f_{\texttt{L}})\}$.

Of course, in general it is possible that infinitely many terms can be constructed. Then we cannot predict how large a model (database state/instance) must be and our method is not applicable. However, this requires at least a nested `NOT EXISTS` subquery (otherwise only Skolem constants are produced, no real functions). And as the example shows, even heavily nested subqueries do not necessarily produce an infinite set of Skolem terms. We treat this problem further in Section 3.4.

Once we know how many tuples each relation must have, we can easily reduce the general case (with subqueries) to a consistency test for a simple formula as treated in [7] (see Section 3.1):

**Definition 5.** *Let a query $Q$ be given, and let $\mathcal{T}_Q$ be finite. The flat form of the* `WHERE`*-clause is constructed as follows:*

1. *Replace each tuple variable $X$ of the main query by the corresponding Skolem constant $f_X$.*
2. *Next, treat subqueries nested inside an even number of* `NOT`*: Replace the subquery*

   `EXISTS (SELECT` ... `FROM` $R_1\,X_1$`,` ...`,` $R_n\,X_n$ `WHERE` $\varphi$`)`

   *by $\sigma(\varphi)$ with a substitution $\sigma$ that replaces the existential tuple variable $X_i$ by $f_{X_i}(Y_{i,1}, \ldots, Y_{i,m_i})$, where $Y_{i,1}, \ldots, Y_{i,m_i}$ are all universal tuple variables on which $X_i$ depends.*
3. *Finally treat subqueries that appear within an odd number of negations as follows: Replace the subquery*

   `EXISTS (SELECT` ... `FROM` $R_1\,X_1$`,` ...`,` $R_n\,X_n$ `WHERE` $\varphi$`)`

   *by* `(`$\sigma_1(\varphi)$ `OR` ... `OR` $\sigma_k(\varphi)$`)`*, where $\sigma_i$ are all substitutions that map the variables $X_j$ to a term in $\mathcal{T}_Q(R_j)$. Note that $k = 0$ is possible, in which case the empty disjunction can e.g. be written* `1=0` *(falsity).*

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

```
NOT(f_L.LOC = f_L.LOC                       /* D.LOC = L.LOC       */
    AND NOT(f_E(f_L).DEPTNO = f_L.DEPTNO    /* E.DEPTNO = D.DEPTNO */
            AND f_E(f_L).JOB = 'SALESMAN')) /* E.JOB = 'SALESMAN'  */
```

This is logically equivalent to

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

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

As in this example, it is always possible to construct a database state that produces an answer to the query from a model of the flat form of the query. The database state/instance will have one tuple in relation $R$ for each term in $\mathcal{T}_Q(R)$ (and no other tuples). It is possible that two of the constructed tuples are completely identical (i.e. there can be fewer tuples than elements in $\mathcal{T}_Q(R)$).

In the opposite direction, note that NOT EXISTS ($\forall$) conditions are only more difficult to satisfy if the database state/instance contains more tuples. E.g. with a single level NOT EXISTS subquery, we need one tuple for each of the tuple variables in the outer query, but we would introduce no additional tuples for the relations listed under NOT EXISTS. It is the basic idea of Skolemnization that we can give names to the tuples that the formula requires to exist, and then reduce the given model to all the named elements.

**Theorem 1.** *Let a query $Q$ be given such that $\mathcal{T}_Q$ is finite. $Q$ is consistent iff the flat form of $Q$ is consistent.*

## 3.3 Integrity Constraints

Consider the following query:

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

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

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

The original query is consistent relative to the constraints iff this extended query is consistent without constraints.

Note that pure "for all" constraints like keys or `CHECK`-constraints need only a single level of `NOT EXISTS` and therefore never endanger the termination of the method. Foreign keys, however, require the existence of certain tuples, and therefore might sometimes result in an infinite set $\mathcal{T}_Q$. This is subject of the next section.

## 3.4 Restrictions and Possible Solutions

As mentioned above, the main restriction of our method is that the set $\mathcal{T}_Q$ must be finite, i.e. no tuple variable over a relation $R$ may depend directly or indirectly on a tuple variable over the same relation $R$. This is certainly satisfied if there is only a single level of subqueries.

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

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

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

Because of the undecidability, this problem can in general not be eliminated. However, it turns out that if we introduce Skolem constants and functions not for tuple variables, but for attributes of tuple variables (domain variables), and do a stricter arity reduction, e.g. the cyclic foreign key can actually still be handled. This foreign key only requires that for every value in the `MGR` column of `EMP`, there is the corresponding value in `EMPNO`, i.e. there is a Skolem function $f_{\mathtt{M,EMPNO}} : \mathtt{EMP.MGR} \to \mathtt{EMP.EMPNO}$. However, there is no restriction for the `MGR`-column of the tuples constructed for `M`: They can all have the same value. Thus $f_{\mathtt{M,MGR}}$ is a Skolem constant (of type `EMP.MGR`).

If even this should not work, one could at least heuristically try to construct a model by assuming that e.g. 2 tuples in the critical relation suffice. Then $\mathcal{T}_Q(R)$

would consist of two constants and one would replace each subquery declaring a tuple variable over $R$ by a disjunction with these two constants. For relations not in the cycle, the original method could still be used. If the algorithm of Section 3.1 constructs a model, the query is of course consistent. If no model is found, the system can print a warning that it cannot verify the consistency. At user option, it would also be possible to repeat the step with more constants.

### 3.5 Null Values

Null values are handled in SQL with a three-valued logic.

*Example 2.* The following query is inconsistent in two-valued logic (without null values):

```
SELECT X.A
FROM   R X
WHERE  NOT EXISTS (SELECT *
                   FROM   R Y
                   WHERE  Y.B = Y.B)
```

However, this query is satisfiable in SQL if the attribute `B` can be null: It has a model in which `R` contains e.g. one tuple `t` with `t.A=1` and `t.B` is null. □

We can handle null values by introducing new logic operators `NTF` ("null to false") and `NTT` ("null to true") with the following truth tables:

| $p$ | NTF($p$) | NTT($p$) |
|---|---|---|
| FALSE | FALSE | FALSE |
| NULL | FALSE | TRUE |
| TRUE | TRUE | TRUE |

In SQL, a query or subquery generates a result only when the `WHERE`-condition evaluates to `TRUE`. Thus, when `EXISTS` subqueries are eliminated in Definition 5, we add the operator `NTF`:

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

In Example 2, a Skolem constant $f_{\texttt{X}}$ is introduced for the tuple variable `X`, and the elimination of the subquery gives the following formula:

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

As usual, `NOT` is first pushed down to the atomic formulas and is there eliminated by inverting the comparison operator. This needs the following equivalences (which can easily be checked with the truth tables):

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

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

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

Next, the formula is as usual converted to DNF. After that, we must check the satisfiablility of conjunctions of atomic formulas that have possibly the operator `NTF` or `NTT` applied to them. An attribute can be set to null iff it appears only in atomic formulas inside `NTT` and the formula is not `IS NOT NULL`. Then it should be set to null, because these atomic formulas are already satisfied without any restrictions on the remaining attributes. Otherwise the attribute cannot be set to null. `IS NULL` and `IS NOT NULL` conditions can now be evaluated. After that, we apply the algorithm from Section 3.1 to the remaining atomic formulas (that are not already satisfied because of the null values).

### 3.6 Program Variables in SQL Statements

In order to be practical, such a tool must also be able to check SQL queries in application programs. E.g. in embedded SQL, these queries can contain program variables. In general, the program variables can be treated like attributes of a new relation. However, the user should at least be warned if a variable can have only a single value in a consistent state, or if two variables must always have the same value.

## 4 Unnecessary Complications

Another reason why queries can be considered as "not good" independent of the application is when they are unnecessarily complicated. Suppose the user wrote a query $Q$, and there is an equivalent query $Q'$ that is significantly simpler, and basically can be derived from $Q$ by deleting certain parts. There might be the following reasons why the user did not write $Q'$:

- The user knew that $Q'$ is not a correct formulation of the task at hand. In this case, $Q$ is of course also not correct, but the error might be hidden in the more complicated query, so that the user did not realize this. A warning would certainly be helpful in this case.
- The user did not know that $Q'$ is equivalent. Since $Q'$ is not a completely different query, but results from $Q$ by deleting certain parts, this shows that the user does not yet master SQL. Again, a warning would be helpful. Often, the simpler query will actually run faster (e.g. the Oracle query optimizer does not remove unnecessary joins).

- The user knew that $Q'$ is equivalent, but he or she believed that $Q$ would run faster. Since SQL is a declarative language this should only be the last resort. With modern optimizers, this should not happen often in practice. If it is necessary, there probably should be some comment, and this could also be used to shut off the warning.
- The user knew that $Q'$ is equivalent, but he or she thought that $Q$ would be clearer for the human reader and easier to maintain. One must be careful to define the possible transformations from $Q$ to $Q'$ such that this does not happen.

Not only students write queries that are unnecessarily complicated, but also certain tools that generate SQL queries sometimes construct queries that are more difficult than needed. E.g. unnecessary joins are a typical example. Writing a simpler, but equivalent query often helps to improve the performance of query evaluation.

Note that inconsistent conditions can be seen as an extreme case of an unnecessary complication: If one does not want any results, the entire query is superfluous.

For space reasons, we can give here only a list of possible errors of this type:

1. **Unnecessary Joins:** If only the key attributes of a tuple variable are accessed, it might be possible to use the foreign key in another tuple variable instead. E.g. the following query prints all employees in department 20, but the tuple variable `D` could be eliminated:

   ```
   SELECT E.ENPNO, E.ENAME
   FROM   EMP E, DEPT D
   WHERE  E.DEPTNO = D.DEPTNO AND D.DEPTNO = 20
   ```

   Some such unnecessary joins are eliminated in the DB2 query optimizer [5].
2. **Tuple Variables that are Required to be Equal:** If two tuple variables are declared over the same relation, and their key attributes are equated, they must always point to the same tuple.
3. **Conditions that can be replaced by TRUE or FALSE:** Implied, tautological, or inconsistent subconditions in a larger condition are unnecessary. E.g. it happens sometimes that conditions are tested in a query that is actually a constraint on the relation. Also if one uses views, it is unnecessary to repeat a condition that is already contained in the view definition. In general, if one replaces a subcondition by `TRUE` or `FALSE`, the resulting query should not be equivalent to the original one.
4. **Unncessarily General Comparison Operator:** Consider the query:

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

   In this case, one could write `=` instead of `>=`. We have also seen students writing `IN` here, which is again quite confusing.

5. **Non-Correlated EXISTS Subqueries:** For a given database state, such subqueries are either true for all assignments of the tuple variables in the outer query, or false for all such assignments. Most likely a join condition is missing. Also subqueries that have a constant truth value under the heuristic assumption that relations are non-empty or that attributes contain at least two different values are suspicious.
6. **Unnecessary DISTINCT**: `DISTINCT` normally requires an expensive duplicate elimination. However, if, e.g., the `SELECT` list contains a key, the `DISTINCT` is superfluous, and a good database programmer should not write it. One might argue that it is the job of the query optimizer to eliminate an unnecessary `DISTINCT`, but at least the Oracle 8i optimizer does not do it, even in really obvious cases, and in the general case, it is again undecidable. The DB2 query optimizer discovers an unnecessary DISTINCT in some cases [5]. `DISTINCT` is always superfluous inside the aggregations `MIN`, `MAX`.
7. **Strange DISTINCT**: `DISTINCT` inside `SUM` or `AVG` is most likely an error.
8. **Unefficient UNION:** Also a `UNION` should be replaced by a `UNION ALL` if one can prove that the results of the two queries are always disjoint.
9. **GROUP BY with singleton groups:** If it can be proven that each group consists only of a single row, the entire aggregation is unnecessary.
10. **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`).
11. **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.
12. **Inefficient HAVING:** If a condition uses only `GROUP BY` attributes and no aggregation function, it can be written under `WHERE` or under `HAVING`. It is much cheaper to check it already under `WHERE`. E.g. in one homework, a join was done under `HAVING`, and it was syntactically correct SQL, because the student added the join attributes under `GROUP BY`.
13. **Strange HAVING:** `HAVING` without `GROUP BY` is strange: Such a query can have only one result or none at all. In special situations this may be a useful trick, but more often it is probably an error.

We do not have space to give algorithms for all of these problems. However, most can be reduced to a consistency test. As an example, let us consider the test whether `DISTINCT` is necessary. It is quite typical. Let the following general query be given:

```
SELECT DISTINCT $t_1$, ..., $t_k$
FROM   $R_1\,X_1$, ..., $R_n\,X_n$
WHERE  $\varphi$
```

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

```
SELECT *
FROM   $R_1\,X_1$, ..., $R_n\,X_n$, $R_1\,X_1'$, ..., $R_n\,X_n'$
WHERE  $\varphi$ AND $\varphi'$
AND    ($t_1$ = $t_1'$ OR $t_1$ IS NULL AND $t_1'$ IS NULL)
AND    ...
AND    ($t_k$ = $t_k'$ OR $t_k$ IS NULL AND $t_k'$ IS NULL)
AND    ($X_1 \neq X_1'$ OR $\cdots$ OR $X_n \neq X_n'$)
```

This query is tested for consistency with the method of Chapter 3. If it is inconsistent, the `DISTINCT` is superfluous: The original query can never produce duplicates.

We use $X_i \neq X_i'$ as an abbreviation for requiring that the primary key values of the two tuple variables are different (we assume that primary keys are always `NOT NULL`). If one of the relations $R_i$ has no declared key, duplicate result tuples are always possible and the `DISTINCT` is not superfluous. The formula $\varphi'$ results from $\varphi$ by replacing each $X_i$ by $X_i'$.

## 5 Results That Are Too Large

Of course, it is in general difficult to detect that a query is incorrect if one does not know for which task the query was written. If the condition is inconsistent, it is clear that the query result is too small (it is always empty). Conversely, it is often also possible to conclude with reasonable likelihood that the query result is larger than intended.

A general approach might be to consider queries as problematic if there is another query that produces the same information in a smaller query result. We need to investigate this idea further in future research, however, it seems to cover all of the cases listed in this section. In general, it is bad if the query result is larger than necessary: It has to be sent from the server to the client over the network and a program or a human must somehow process the query result. Note that a query with an inconsistent condition gives no information from the database: Its result is fixed, no matter what the database state is.

Again, for space reason, we only list possible errors of this type:

1. **Missing Join Conditions**: A very common error is to forget a join condition. In one exam exercise, 11 out of 70 students had an error of this type.
2. **Many Duplicate Answers**: Another case where the query result is probably too large is when it contains (many) duplicates. E.g. consider the following query:

   ```
   SELECT JOB
   FROM   EMP
   ```

   Normally, there will be many employees with the same job. Then it would have been better either to add `DISTINCT` or to use `GROUP BY` and count for every job how many employees there are with this job. Of course, when duplicates are unlikely or at least very few, one would consider a query without "`DISTINCT`" acceptable:

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

Since `ENAME` is not a key, duplicate answers are possible, but unlikely. Formally, one can use "soft constraints", i.e. conditions that are in general true, but can have exceptions. For the purpose of detecting duplicates one does the above test, but assumes that all columns with no or only few duplicates form a key. The reason is that "`ENAME`" is used in real life to identify "`EMP`" objects, although we know that it might be not always unique. Detecting queries that can generate duplicate answers is also useful because this is often a consequence of another error, such as a missing join condition.

3. **Constant Result Columns**: A query result can also be "too wide". E.g. if the condition contains $A = c$, it makes little sense to put $A$ into the `SELECT` list. Yet, beginners often make this mistake.
4. **Identical Result Columns**: Sometimes a query has two result columns that must always contain the same value. This is not useful.

Techniques developed in query optimization for estimating the result size can also be used: If the estimated size is extremely large, one should warn the user before the query is really executed.

## 6 Possible Runtime Errors

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

1. If one uses a condition of the form `A = (SELECT ...)`, it is important that the subquery returns only a single value. If this condition should ever be violated, the DBMS will generate a run-time error. This can be tested with a method very similar to the test for an unnecessary `DISTINCT` shown above, one only replaces the `SELECT`-list by "`SELECT 'yes'`.
2. The same problem happens if SQL is embedded in a programming language, and one uses the `SELECT ... INTO ...` syntax.
3. 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.
4. Also, the very permissive type system of at least Oracle SQL can pose a problem: Sometimes strings are implicitly convered to numbers, which can generate runtime errors. In general, if one knows domains for the attributes, one could warn the user for comparisons between attributes of different domains. If there is no domain information, one could analyze an example database state for columns that are nearly disjoint.
5. In addition, datatype operators have the usual problems (e.g. division by zero).

## 7 Conclusions

There is a large class of SQL queries that are syntactically correct, but nevertheless certainly not intended, no matter what the task of the query might be. One could expect that a good DBMS prints a warning for such queries, but, as far as we know, no DBMS does this yet.

In this paper we have shown various kinds of semantic errors that can be detected without knowing the task of the query. We have tried to be complete, the authors would be thankful for reports about other such errors. All errors did actually occur in exam or homework exercises. We have algorithms for detecting all these error types, however some algorithms still need improvements. In future work, we aim at a rule-based system that permits to define new types of errors easily (or a proof that we really detect all such errors).

A prototype of the consistency test is available from

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

Extended versions will be made available under the same address.

## Acknowledgements

Of course, without the students in my database courses, this work would have been impossible. We would also like to thank Jan Van den Bussche for suggesting relevant literature and Joachim Biskup for contributing the idea that query size estimation techniques could be used. We would like to thank Sergei Haller and Ravishankar Balike for telling us about the type of error illustrated in the example in the Introduction. Elvis Samson developed the prototype of the consistency test, and made several suggestions for improving the paper, which is both gratefully acknowledged.

## References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1994.
2. François Bry, Rainer Manthey: Checking Consistency of Database Constraints: a Logical Basis. In *Proceedings of the 12th International Conference on Very Large Data Bases*, 13–20, 1986.
3. François Bry, Hendrik Decker, Rainer Manthey: A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *Proceedings of the International Conference on Extending Database Technology*, 488–505, 1988.
4. Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15:162–207, 1990.
5. Qi Cheng et al.: Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. *Proceedings of the 25th VLDB Conference*, 687–698, 1999.

6. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.
7. 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.
8. A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in Datalog extensions. *Journal of the ACM*, 48:971–1012, 2001.
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. Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35:146–160, 1988.
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.