Journal of Universal Computer Science, vol. 15, no. 3 (2009), 584-619
submitted: 15/2/07, accepted: 26/1/09, appeared: 1/2/09 © *J.UCS*

Structural Coverage Criteria for Testing SQL Queries

Mª José Suárez-Cabal
(Department of Computer Science, University of Oviedo, Spain
cabal@uniovi.es)

Javier Tuya
(Department of Computer Science, University of Oviedo, Spain
tuya@uniovi.es)

Abstract: Adequacy criteria provide an objective measurement of test quality. Although these criteria are a major research issue in software testing, little work has been specifically targeted towards the testing of database-driven applications. In this paper, two structural coverage criteria are provided for evaluating the adequacy of a test suite for SQL queries that retrieve information from the database. The first deals with the way in which the queries select and join information from different tables and the second with the way in which selected data is further processed. The criteria take into account both the structure and the data loaded in the database, as well as the syntax and semantics of the query. The coverage criteria are subsequently used to develop test inputs of queries drawn from a real-life application. Finally, a number of issues related to the kind of faults that can be detected and the size of the test suite are discussed.

Keywords: database testing, SQL testing, test adequacy criteria, test coverage
Categories: D.2.5

1 Introduction

Ranging from legacy applications in use in the banking, financial or insurance sectors to modern e-commerce applications, there is one component which they all have in common, the database, where sensitive business information is stored and retrieved. Programming languages have experienced a paradigm shift from monolithic programs written in old imperative languages to highly scalable enterprise applications, reusable components and web services written in object-oriented languages. At the same time, database management systems (DBMS) have evolved, increasing their performance, scalability and reliability.

To access information, however, business and data layers are still using the Structured Query Language (SQL) developed in the late 1970s, which has been standardised by ANSI and ISO and has evolved over the years by including new features (1986, 89, 92, 99 and recently in 2003), SQL92 [SQL 1992] being the most commonly used version. The huge number of applications that make use of SQL leads to a real need for helper techniques for its development, in general, and testing, in particular. However, although many testing techniques [Woodward 2001] and adequacy criteria [Zhu et al. 1997] exist, these are not tailored to address certain specific issues that differentiate this kind of non-imperative language from others.

The most frequently used SQL statements in commercial applications are those that retrieve information (SELECT queries) [Pönighaus 1995], that use a common set

of major characteristics, such as the database schema and the core clauses for projecting, joining, selecting and grouping data. However, developing a single statement may be a complicated task [Lu et al. 1993] and queries using GROUP BY, ORDER and HAVING clauses are considered especially difficult by programmers. The number of research papers relating to the topic of test data selection for databases and programs or applications with embedded SQL is limited. The paper written by [Mannila, Räihä 1986] is the first found where test data are generated for queries (represented in relation algebra) but most research has appeared since the year 2000. Test cases are complicated to write because the input is information spread over several tables containing many rows, and the output is likewise a table structure. Queries are closely dependent on the database schema and small changes can entail undesirable side effects in many queries. Moreover, SQL uses a mixture of set-based and logic-based techniques and the logical expressions use a three-valued logic for supporting missing information (null values), which in turn makes the process of writing and testing queries even more difficult [Klein 1994]. A list of frequent SQL errors is given by [Brass, Goldberg 2005].

The goal of this paper is to define coverage criteria for assessing the adequacy of the test suite to exercise various situations that affect the data retrieved by an SQL query. The approach studies queries in an isolated way without considering the imperative code where they will be embedded and the tests can be used as prerequisites for embedding queries in the imperative code.

This paper improves the approach given in [Suárez-Cabal, Tuya 2004], where queries only had FROM and WHERE clauses and conditions were exclusively composed of attributes, constants or NULL. The present paper also considers parameters, GROUP BY and HAVING clauses, aggregate functions, ALL and DISTINCT quantifiers along with UNION operator. Moreover, it shows how to automate the calculation of the coverage and it analyzes different kinds of faults in queries classified in two categories: non SQL-specific (but typical faults in the conditions in imperative programs) and SQL-specific. The approach involves building one or more coverage nodes that are created on the basis of the structure of the query and the database schema. Nodes are arranged in trees for assessing the adequacy of join and selection operations, and in sets for assessing the coverage of the processing performed after selection. Coverage is then evaluated in relation to the load provided by the test database and to the actual parameters dependent on the imperative code. After evaluating coverage, with the information of the non-covered situations in the nodes, the tester has guidelines to follow in the process of completing the test suite by adding or changing information in the test database, creating a new test database and/or calling the query with different parameters.

The paper is organized as follows: [Section 2] includes the description of the relational database model, the SQL specification of the kind of queries used in this approach and the definitions of test suites. [Section 3] describes in detail how the evaluation of the coverage of SQL queries is performed. In [Section 4], the coverage information is used to develop test inputs for a set of queries obtained from a real-life application and [Section 5] discusses a number of issues related to the fault detection ability, the size of the test suite and the complexity of the evaluation algorithm. Finally, in [Section 6], an overview of related work on database testing is given and conclusions are presented in [Section 7].

2 Background

2.1 Relational Database Model

The main concept in the relational data model is a *relation r(R)*, where *R* is the *relation schema*. Given a set of attributes $A_1, \ldots, A_n$ and the domains $D_1, \ldots, D_n$ where $D_i=Dom(A_i)$, *r(R)* is a subset of the Cartesian product of the domains, $r(R) \subseteq D_1 \times \ldots \times D_n$. Each D_i is a finite and homogeneous set of values and the special value called NULL that indicates the absence of valid information and which may be interpreted as undefined, not relevant or unknown.

A *relation schema R* is composed of the list of attributes, domains and constraints of *r(R)*. For referencing an attribute *A* of a relation schema *R* the notation used is *R.A*. A *database DB* is a set of relations $\{r(R_1), \ldots, r(R_m)\}$. A *database schema DBSchema* is composed of a set of relation schemas and the descriptions of the logic relations between them.

Each element of *r(R)* is referred to as *tuple t* and is composed of a list of values $(v_1,\ldots, v_n)$ where $v_i \in D_i$. For referencing the value of the attribute A_i in the tuple *t* the notation used is $v_i=t[A_i]$.
When the relational database model is implemented into a particular database management system (DBMS), relations are referred to as *table*s, tuples as *rows* and attributes as *columns*.

A *primary key PK* in *R* is a subset of its attributes $PK \subseteq \{A_1, \ldots, A_n\}$ where each tuple is uniquely identified by the primary key values. A *PK* must be a minimal set of attributes for which this uniqueness property exists. A *foreign key FK* in R_i is a set of attributes used to reference a *PK* in another relation schema R_j.

2.2 Structured Query Language (SQL)

Structured Query Language (SQL) is the language used to define database schemas and insert, delete, modify and access data stored in databases. The SQL statements considered in this paper are those that retrieve information (SELECT queries). The SELECT clause determines which columns constitute the query output, the FROM clause determines which tables are used and the JOIN determines the criterion for joining rows from different tables (join-conditions). Then the WHERE clause filters the rows based on given criteria (where-conditions). The GROUP BY clause indicates how to combine the selected rows and the HAVING clause performs a final filter based on other criteria (having-conditions).

The subset considered in this paper is that represented in the following BNF grammar:

```
<select query> ::= <select> [UNION [ ALL | DISTINCT ] <select query>]
<select> ::= SELECT [ ALL | DISTINCT ] <select list> <from clause>
    [ <where clause> ]
    [ <group clause> [ <having clause> ] ]
<select list> ::= '*'
    | <column elemnent> [ { ',' <column element> }... ]
<column element> ::= <column>
    | <aggregate function> '(' <column> ')'
```

```
<from clause> ::= FROM <table reference>
    [{','<table reference>}...]
<table reference> ::= <table schema> [[AS ] <correlation name>]
    | <table reference>
      [INNER | LEFT | RIGHT] JOIN <table reference>
      ON <search conditions>
<where clause> ::= WHERE <search conditions>
<group clause> ::= GROUP BY <grouping columns>
<having clause> ::= HAVING <search conditions>
```

The <*search conditions*> term is a logical predicate composed of logical *conditions* concatenated with *AND* and *OR* operators. A *condition* is an expression in the query in the form $X \Re Z$ or $X \Re_N NULL$, where X and Z are sets of values represented by the name of their column, aggregate functions, constants, parameters or NULL, $\Re$ is an operator of {=, !=, <, <=, >, >=} and $\Re_N$ is a predicate of {IS, IS NOT}. An *aggregate function* (*count, sum, max, min* or *avg*) transforms a set of scalars or a set of rows into a scalar.

2.3 Test Suite Definitions

Test cases for programs written in imperative language are composed of the input values (actual parameters) for the set of formal parameters and the desired output, but SQL queries are always executed using a database and actual parameters (if any). Therefore the test inputs for SQL queries must provide both of them.

Given an SQL query Q that contains a set of formal parameters $F=(f_1, \ldots, f_p)$, *a test input TI* for S with regard to a database DB is a pair <DB, P> where $P=(p_1, \ldots, p_f)$ are the actual parameters used for instancing each formal parameter of Q in a program execution. A *test group TG* for Q is a set of test inputs $(TI^1, \ldots, TI^t)$ sharing the same test database. If the query has no formal parameters, its *TG* contains only a test input composed of a test database. Finally, *test suite TS* for Q is defined as the union of all *TG* for Q.

3 SQL Coverage Criteria

An SQL statement, in the form given in [Subsection 2.2], does not have any loops or iterations, although it does have a structure that can be used to explore how the different situations in the test database exercise the processing done by query. For the operations related to data retrieval from relations (SELECT, FROM, JOIN and WHERE clauses) and the operation which filters out grouping information (HAVING clause) the approach consists in defining the coverage for query conditions by building a coverage tree for join and where conditions and another for having conditions (if any). Coverage trees are composed of nodes (*c-nodes*) and each of them has different values (*c-values*) that represent the different condition situations to be exercised and in evaluating the degree to which the c-values are covered when the query is executed using the test inputs. The coverage measurement for conditions incorporates a notion of multiple condition coverage (although other criteria could be

used, it corresponds to more exhaustive testing of each condition that subsumes the rest of test coverage criteria for conditions [Zhu et al. 1997]).

Additionally, SQL performs a further processing after retrieving the data, in which conditions are not explicitly stated (GROUP BY, aggregate functions, DISTINCT and ALL set quantifiers). In these cases, a new type of coverage is defined and it is evaluated on the basis of a set of conditions derived from the general rules that specify the run-time effect of the query. A set of nodes (*r-nodes*) is built for these cases, each having different values (*r-values*). In this case, the r-nodes are not organized in a structure because grouping attributes, attributes in aggregate functions and sets quantifiers do not have hierarchical relations and each one is independently evaluated.

Firstly, a particularized notion of evaluation of conditions of SQL queries is introduced in [Subsection 3.1]. Then the procedure for automatically calculating the coverage is described in detail considering queries which include clauses that retrieve data in [Subsection 3.2] and others that perform its further processing in [Subsection 3.3].

3.1 Evaluation of Conditions

A simple example illustrating some basic issues related to the evaluation of single conditions is a database schema composed of two relation schemas named *M* (master) and *D* (detail). *M* has only one attribute named *id* and *D* has two attributes: *id* and *ref*. In both of them, the *id* attributes are *PK*. For this example, a query specification could be to extract a list of all pairs of values of *M.id* and *D.id* in which the value of the attribute *D.ref* coincides with the value of *M.id*. An SQL query that implements correctly this specification is:

```
SELECT M.id, D.id FROM M INNER JOIN D ON M.id=D.ref
```

However, assume that the query is wrongly implemented using a LEFT JOIN instead of an INNER JOIN:

```
SELECT M.id, D.id FROM M LEFT JOIN D ON M.id=D.ref
```

To exercise the condition that joins the information of both relations, a test input is designed in which this condition takes the possible outcomes: true and false. For example, the test input *r(M)*={(1)}, *r(D)*={(11,1), (11,2)} makes the condition true when the tuple in r(*M)* is joined with the first tuple in *r(D)*, and false when the tuple in r(*M)* is joined with the second tuple in *r(D)*. The output of the query is {(1,11)}. However, if the tuples in *r(M)* were {(1), (3)}, then the output of the query would be {(1,11), (3,NULL)}, which is an incorrect output. The first designed test input was unable to reveal this fault, which suggests the need for a different approach to evaluating conditions.

3.1.1 Evaluation of Simple Conditions

One aim of using coverage criteria is to guide the expert to select test inputs which permit the evaluation of different situations in the conditions (named <*search*

condition> terms in [Subsection 2.2]) and then the query processing retrieves or rejects tuples to make up the query output.

In order to evaluate conditions, the approach proposes to analyze join and where conditions in a homogeneous way in a coverage tree using tuples extracted from the test database. On the other hand, the conditions in the HAVING clause are evaluated in an independent coverage tree because tuples used for their evaluation are obtained as result of the processing of GROUP BY clause instead of being obtained directly from the test database.

Due to the fact that the operands of a condition of SQL queries are composed of a set of values, which are normally identified with their attributes, during its evaluation each value of the first operand is evaluated with each value of the second (named *evaluation from left to right*) and vice versa (*evaluation from right to left*) and the result is true or false depending on the pair of values evaluated.

Let C be a condition of an SQL query Q in the form $X \Re Z$ the following valuations are established for such conditions:

- *Fl*: C is *left-falsified* iff there exists at least one tuple t_i belonging to the relation of X where $x=t_i[X]$ such that there does not exist any tuple t_j belonging to the relation of Z where $z=t_j[Z]$ satisfying the condition $x \Re z$.
- *Fr*: C is *right-falsified* iff there exists at least one tuple t_j belonging to the relation of Z where $z=t_j[Z]$ such that there does not exist any tuple t_i belonging to the relation of X where $z=t_i[X]$ satisfying the condition $x \Re z$.
- *T*: C is *verified* iff there exists at least a pair of tuples, t_i belonging to the relation of X where $x=t_i[X]$ and t_j belonging to the relation of Z where $z=t_j[Z]$, such that the condition $x \Re z$ is satisfied. In this definition, there is no distinction between left-verified and right-verified, because if there is a pair of values that verifies the condition when evaluated from left to right, then the same pair of values verifies the condition when evaluated from right to left.

When one or both of values (x or z) in the operands is NULL, the result of evaluation of the condition $x \Re z$ is *indeterminate* (neither true nor false). For that reason, programmers and testers must be very careful to avoid undesirable effects resulting from the incorrect behaviour of conditions having null values ([Vassiliou 1979] and [Imielinski, Lipski 1984]). Therefore, in order to obtain a complete set of test inputs, NULL must be taken into consideration. The following valuations are established for considering null values in attributes:

- *Nl*: *C is left-null* iff there exists at least one tuple t_i belonging to the relation of X where $x=t_i[X]$ such that x is NULL.
- *Nr*: *C is right-null* iff there exists at least one tuple t_j belonging to the relation of Z where $z=t_j[Z]$ such that z is NULL.
- *Nb*: *C is null* iff there exists at least a pair of tuples, t_i belonging to the relation of X where $x=t_i[X]$ and t_j belonging to the relation of Z where $z=t_j[Z]$, such that x and z are NULL simultaneously.

To check whether a value of an attribute or expression is NULL, SQL provides the predicates "IS" and "IS NOT". Given a condition C in the form $X \Re_N NULL$:

- If $\Re_N$="IS" and there exists at least one tuple belonging to the relation of X where $x=t_i[X]$ and x is NULL, or $\Re_N$="IS NOT" and there exists at least one tuple belonging to the relation of X where $x=t_i[X]$ and x is distinct from NULL then C is T.
- If $\Re_N$="IS" and there exists at least one tuple belonging to the relation of X where $x=t_i[X]$ and x is distinct from NULL, or $\Re_N$="IS NOT" and there exists at least one tuple belonging to the relation of X where $x=t_i[X]$ and x is NULL then C is Fl.
- If $\Re_N$="IS" and there are no tuples belonging to the relation of X where $x=t_i[X]$ and x is NULL or $\Re_N$="IS NOT" and all tuples belonging to the relation of X where $x=t_i[X]$ and x is NULL then C is Fr.

Note that for every $\Re_N$, if C is Fr then it is never T, and vice versa. With regard to considering the existence of null values, C is always Nr, and it is Nl and Nb iff there exists at least one tuple belonging to the relation of X where $x=t_i[X]$ and x is NULL.

3.1.2 Condition Coverage

Each of the possible valuations (*Fl*, *Fr*, *T*, *Nl*, *Nr* and *Nb*) is denoted as a *c-value*. When a condition verifies each of them during its evaluation, then this c-value is said to be *covered*. Thus, a first measure of the coverage of a condition may be established as the percentage of c-values being covered when exercising that condition using the test inputs. If there are c-values that have not been covered, the coverage is lower than 100%, which indicates the existence of situations in the condition that are not exercised by the test inputs. So, low percentages indicate test data are not representative enough and, therefore, test inputs may be added in order to increase the coverage and to complete the test suite.

Graphically, a condition C is depicted by a box named *condition node (c-node(C))* with six placeholders, one for each of the c-values. A c-value is labelled as *'N'* if it is not covered, *'Y'* if it is covered, *'I'* (impossible) if it cannot be covered due to some known restriction imposed by the database schema and *'U'* (unreachable) if it cannot be covered because of characteristics or constraints that do not depend on the database schema, such as the condition, their operands, constants or parameters. Note that if database schema is modified affecting any attribute of the condition, the impossible c-values could change with the new constraints.

Initially, each c-value is labelled *'N'* meaning that the c-value has not been covered yet. Moreover, the c-values impossible to cover because of the database schema are automatically labelled as *'I'*:

- *Nb*, *Nl* or *Nr*: when null value is not allowed for the values of their attribute (the attribute has been declared in the database schema as *PK* or NOT NULL).

- *Fl*: when the condition operands are the attributes A_{n1} and A_{n2}, $\Re \in \{=, <=, >=\}$, and A_{n1} is an attribute that is an *FK* referencing the other attribute A_{n2}. In this case, the values of A_{n1} cannot be distinct from the values in A_{n2}.
- *Fr*: because of a symmetric situation when A_{n2} references A_{n1}.

Furthermore, there are unreachable c-values that can be automatically detected and are labelled as *'U'*. These c-values depend on the constant term *K* of a given condition in the form $X \Re K$:

- *T*, *Fl* and *Fr* are *'U'* when *K* is NULL, since the result of evaluation is always indeterminate.
- *Nr* and *Nb* are *'U'* when *K* is distinct from NULL.

During the evaluation of the condition with the designed test inputs, c-values labelled as *'N'* are reached and labelled as *'Y'*. A particular case of the evaluation of conditions is that of some of these operands being a constant term *K*. Given a condition in the form $X \Re K$:

- If *K* is NULL, then *Nl* is always covered and therefore labelled automatically as *'Y'*.
- If *K* is distinct from NULL, then the c-values *T* and *Fr* are never covered simultaneously: if, some of the tuples belonging to the relation of *X* have values in *X* that verify the condition then *T* is covered and labelled as *'Y'* and therefore it is impossible to have all values falsifying it.

A completely symmetric situation in the automatic initializations and evaluations of conditions with a constant term is attained when the condition is in the form $K \Re Z$.

When a query *Q* has conditions which contain formal parameters, each test input supplies actual parameters for instancing each formal one. Once instantiated, these behave in the same way as in the case of constants.

3.2 Coverage for Select and Join Operations

This subsection deals with most common SQL queries, namely those involving one or more relations joined using some of the SQL join operators and/or conditions specified by a WHERE clause. Firstly, the notion of the condition coverage tree will be introduced. Then, the method for automatically evaluating the coverage of the whole query will be described in detail.

3.2.1 Condition Coverage Tree

A *condition coverage tree CT* is a data structure for representing all possible combinations of the results of evaluation of the conditions of an SQL query using the test inputs. Each condition C_k, extracted from *Q* (which includes the conditions in the JOIN and/or WHERE clauses), is represented in a level of the tree in the same order as it appears in the query, upper levels for conditions of JOIN clauses and lower levels for conditions of the WHERE clause.

Given a SELECT query Q, its corresponding condition coverage tree $CT(CS)$ is constructed by considering the ordered set of conditions $CS=(C_1, \ldots C_s)$ of Q. At a level k of the coverage tree [see Fig. 1], $CT(CS^k)$ is defined in a recursive form as:

- An empty structure, if there are no conditions, or
- A hierarchical structure $CT(CS^k)=<c\text{-}node(C_k), CT_T(CS^{k+1}), CT_{Fl}(CS^{k+1}), CT_{Fr}(CS^{k+1})>$, where $CS^{k+1}=CS^k-\{C_k\}=(C_{k+1},\ldots C_s)$, composed of a $c\text{-}node(C_k)$ for the condition C_k and three condition coverage sub-trees for the rest of conditions of CS^{k+1}. Each sub-tree depends on the evaluation of c-values T, Fl, and Fr of $c\text{-}node(C_k)$ respectively.

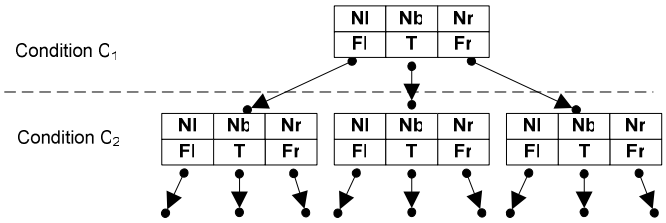

Figure 1: Condition coverage tree

Note that the c-values *Nb*, *Nl* and *Nr* do not generate a sub-tree because they only represent the existence of null values in the operands and, in these cases, the result of the evaluation of the condition is indeterminate.

Before the evaluation of $CT(CS)$, each possible c-value is automatically initialized as has been indicated in [Section 3.1.2].

When a *T*, *Fl* or *Fr* c-value is *'I'* or *'U'*, the entire sub-tree generated from it is also labelled as the same value.

Example 1: Creation of a Condition Coverage Tree

In order to illustrate the creation of a condition coverage tree, an explanation of the process followed by an example query is now given.

This example is extracted from the case study which is detailed in [Section 4]. SQL queries are part of a web-based helpdesk system. The system stores helpdesk *tickets*, which are created for user requests. The person who records a *ticket* is its *receiver* and the person responsible for taking some action on it is its *owner*. If an action is carried out on a ticket, a *history* record is created by a user, its *creator*, indicating annotations, time tracking, attachments and changes in the ticket state and owner.

The following SQL statement extracts the list of creators, the type of invoiceable tickets and the spent time:

```
SELECT H.creatorID,T.typeID, H.spentTime
FROM ticket T INNER JOIN history H ON T.ticketID=H.ticketID
WHERE T.invoiceable=1
```

The condition coverage tree created [see Fig. 2] has two levels, one for each condition. Initially, c-values are automatically labelled as:

- Impossible, *'I'* (depending on the database schema): as T.ticketID is *PK,* it must be not NULL; H.ticketID is *FK* of *ticket* so the *Fr* c-value of condition *T.ticketID=H.tickedID* can not be covered or entire CT_{Fr} sub-tree. Moreover *H.ticketID* and *T.invoiceable* must be not NULL.
- Unreachable, *'U'* (depending on the conditions of the query): the constant '1' can not be NULL, so all *Nr* c-values of its level of the coverage tree are never covered.
- Non-covered, *'N'* (the rest of c-values, only eight of the twenty-four in total).

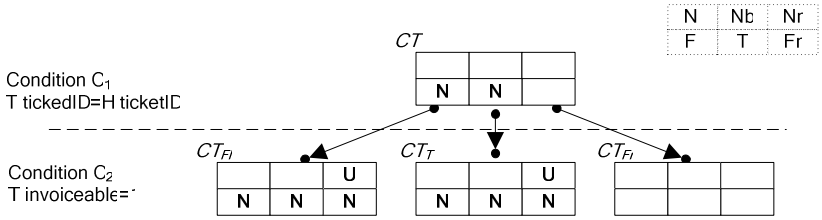

Figure 2: Initial condition coverage tree for a simple query

3.2.2 Evaluation of the Condition Coverage Tree

The condition coverage tree *CT(CS)* for the set of conditions *CS* of a query is automatically evaluated for each test input *TI*. For each, the evaluation algorithm takes the inputs *CT(CS)*, where the formal parameters (if any) have been previously instantiated, and the test database.

The evaluation of the condition coverage tree for each *TI* begins at the root c-node and proceeds recursively by calculating the coverage of each child sub-tree.

The algorithm for evaluating conditions in the form $X \Theta Z$, where *X* and *Z* are attributes of the relations $r(R_X)$ and $r(R_Z)$ respectively, is shown in Appendix I.

At each level *k* of the coverage tree, C_k is evaluated from left to right where, for each tuple t_i of relation $r(R_X)$, the value $x=t_i[X]$ is compared with the value $z=t_j[Z]$ of each tuple t_j of relation $r(R_Z)$. If the condition is satisfied, the c-value *T* is labelled as *'Y'* and then the sub-tree $CT_T(CS^{k+1})$ is evaluated using as input the test database except the tuples of the relations $r(R_X)$ and $r(R_Z)$, which are replaced with t_i and t_j respectively. If the condition was never satisfied for a value *x* and any *z* and then *Fl* is labelled as *'Y'* and $CT_{Fl}(CS^{k+1})$ is evaluated using as input the test database replacing the tuples of $r(R_X)$ with t_i.

After, C_k is evaluated from right to left in a similar way. However, only c-value *Fr* and $CT_{Fr}(CS^{k+1})$ are evaluated because c-value *T* and its child sub-trees have already been evaluated.

Some considerations about the algorithm are summarized below:

- Before evaluating each condition $x \Re z$, the existence of null values is checked and then c-values *Nl*, *Nr* and *Nb* are labelled as *'Y'* depending on *x* and *z*.
- The c-node evaluation finishes when there are no more values in the operands or every c-value is covered (*'Y'*), impossible (*'I'*) or unreachable (*'U'*).
- A c-value and its sub-tree can only be evaluated when the c-value is labelled different from *'I'* and *'U'*.
- When the condition has constants or parameters, there is only one value for evaluating and it is used in the evaluation of sub-trees.
- Note that if the condition is $X \Re_N NULL$, each value *x* is compared with NULL and the sub-trees $CT_T(CS^{k+1})$ and $CT_{Fl}(CS^{k+1})$ are evaluated using as input the test database replacing the tuples of $r(R_X)$ with t_i, and $CT_{Fr}(CS^{k+1})$ is evaluated with the same test database.

After evaluation of *CT(CS)*, when c-values remain without covering (labelled as *'N'*), the tester may examine them and determine if it is necessary to complete test inputs or label them manually as unreachable *'U'*.

Example 2: Evaluation of a Condition Coverage Tree
This example shows the process of the evaluation of a condition coverage tree given an initial set of test inputs.

Considering the previous query and its condition coverage tree (see [Example 1]), test inputs are tickets and their histories. Assume that the initial test inputs are those presented in [Tab. 1], composed of an invoiceable ticket with details (one history row) and another non-invoiceable without details. The columns on the right present the output of the query.

ticket			**History**				**Output**		
ticketID	**invoiceable**	**typeID**	**historyID**	**ticketID**	**creatorID**	**timeSpent**	**creatorID**	**typeID**	**timeSpent**
1	1	81	11	1	91	8.0	91	81	8.0
2	0	89							

Table 1: Initial test inputs for a simple query

After evaluating the join condition *T.ticketID=H.ticketID* (labelled as C_1), test inputs cover the c-values (*T* and *Fl*):

- c-value *T* is covered with a ticket (*ticketID=1*) and its history (*historyID=1*). These rows are used to evaluate c-values in the sub-tree CT_T for the condition *T.invoiceable=1* (labelled as C_2), covering c-value *T*.
- c-value *Fl* is covered with the ticket without details (ticketID=2) which is used to evaluate C_2 in the sub-tree CT_{Fl}, covering c-values *Fl* and *Fr* because it is non-invoiceable.

The resulting condition coverage tree after the evaluation is shown in [Fig. 3]. As can be seen, there are still c-values non-covered.

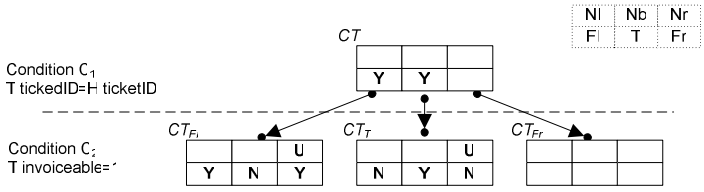

Figure 3: Evaluated condition coverage tree for a simple query

The coverage evaluation informs the tester that it would be necessary to complete test inputs in order that all c-values were distinct from *'N'*. For example:

- To cover the c-values *Fl* and *Fr* for C_2 in the sub-tree CT_T, it would be necessary to have non-invoiceable tickets with details. Then, a new ticket (3,0,89), where *invoiceable=0*, is added along with its detail, row (12,3,99,9.9) in the *history* table.
- Similarly, to cover the c-value *T* for C_2 in the sub-tree CT_{Fl}, an invoiceable ticket without details should be included. Then a new row (4,1,89) is added in the *ticket* table.

Using the final test inputs selected [see Tab. 2], the condition coverage tree is evaluated again and all c-values are covered.

ticket			**History**				**Output**		
ticketID	**invoiceable**	**typeID**	**historyID**	**ticketID**	**creatorID**	**timeSpent**	**creatorID**	**typeID**	**timeSpent**
1	1	81	11	1	91	8.0	91	81	8.0
2	0	89	12	3	99	9.9			
3	0	89							
4	1	89							

Table 2: Test inputs for a simple query

3.3 Coverage of Other SQL Operations

The other kind of core operation carried out by the SQL query after retrieving the data consists in organizing it in groups (GROUP BY clause), to calculate aggregate functions over the data and to discard duplicate tuples (set quantifiers DISTINCT and ALL). In these cases, conditions are determined by taking into account the situations that specify the run-time effect of the query. Each of the conditions produces an *r-value*. The r-values are organized in *r-nodes*. In this case, r-values are labelled as possible but not covered yet (*'N'*), covered (*'Y'*) or unreachable (*'U'*). There are no

impossible r-values because the SQL code of the query could filter the constraints defined in the database schema.

3.3.1 Grouping Columns

Let *Q* be a query with a GROUP BY clause composed of a list of *grouping columns* $A_1..A_c$ (each of them is either the name of a single attribute or an expression over the values of attributes). In this case the select-list is in the form $A_1..A_c$, $F_{c+1}..F_n$, where each F_i is an aggregate function expression over the values of attributes. Firstly, the query retrieves a set of tuples $\{t_1..t_n\}$ and then produces a result set composed of the tuples $\{u_1..u_m\}$ where each u_k is a group of one or more t_i. According to SQL specification [SQL 1992], groups are partitioned "into the minimum number of groups such that for each grouping column of each group, no two values of that grouping column are distinct". As the semantics of the GROUP BY clause interprets null values in the grouping column as belonging to different groups, two conditions are considered, one in which grouping columns are not NULL and another in which they are. For each grouping column A_i two *r-values* are established:

- *G (not null grouping)*: There exists two groups u_k, u_l in the result set, each of them is composed of more than one tuple, such that the values of $u_k[A_i]$ and $u_l[A_i]$ are different and distinct from NULL and all the others $u_k[A_j]$ and $u_l[A_j]$, where $i!=j$, are the same and distinct from NULL.
- *GN (null grouping)*: There exists two groups u_k, u_l in the result set, each of them is composed of more than one tuple, such that the value of $u_k[A_i]$ is NULL, $u_l[A_i]$ is not and all the others $u_k[A_j]$ and $u_l[A_j]$, where $i!=j$, are the same and distinct from NULL.

The evaluation of the coverage is straightforward by using the actual output produced by the query and considering only the tuples having cardinality higher than one. Each grouping column A_i produces an *r-node*, each of which has two placeholders for the r-values *G* and *GN* respectively.

3.3.2 Aggregate Functions

The aggregate functions (SUM, MIN, MAX, COUNT, AVG) perform simple calculations over all values that are included in each group. Additionally, SUM, COUNT and AVG can specify the optional set quantifier DISTINCT, which, if present, excludes the repeated values from the calculation. Two conditions that affect the calculation of the aggregate function are considered: (1) if some values are repeated, then only one value is taken into account if the DISTINCT set quantifier is present and (2) if a value is NULL, then it is not taken into account.

Let *Ag* be an attribute contained in an aggregate function expression, and $v^i=t_i[Ag]$ each of the values of the tuples t_i that are grouped at the tuple u_k in the result set. The following r-values are defined for each:

- *AF (multiple aggregation)*: There is at least one group *k* in which the aggregate functions are evaluated over at least three non-null values v^1, v^2, v^3 distinct from zero such that $v^1=v^2$ and $v^2!=v^3$. As two of the evaluated values

are equal, faults consisting of an incorrect use of the DISTINCT set quantifier (omission or misuse) can be detected.

- *AFn (aggregation with null)*: There is at least one group k in which the aggregate functions are evaluated over at least two values v^1, v^2 such that v^1 is NULL and v^2 is not.

Before the evaluation, an intermediate query is constructed by removing the GROUP BY, and by replacing each reference to aggregate functions by their arguments. Then the result set produced by this query is explored to evaluate each r-value.

The coverage of the aggregate functions consists of a set of r-nodes, one for each attribute *Ag*, each of these having two placeholders, one for each of the r-values *AF* and *AFn*.

3.3.3 Other Set Quantifiers

The clauses SELECT and UNION (combination of queries) may be evaluated using the ALL and DISTINCT set quantifiers. DISTINCT produces an output in which duplicate tuples are removed and ALL keeps duplicate tuples if any. If no quantifier is specified, then the SELECT clause uses ALL by default and UNION operator uses DISTINCT.

Ensuring that duplicate tuples appear only when specified is a very important issue to avoid duplicate processing that could produce unexpected results. The condition used to evaluate the coverage is the presence or absence of duplicate tuples before quantifying. The following *r-values* are defined:

- *S*: For each SELECT clause, when quantified using ALL, the result set contains at least two tuples that are equal. This r-value is automatically labelled as unreachable if: (1) the select-list contains all the attributes that are primary keys of the relations being joined and (2) the query has a GROUP BY clause, this r-value is not evaluated.
- *U*: For each UNION clause, when quantified using ALL, the result set of each of the combined queries has at least two tuples that are equal.

In this case, an r-node consisting of a single r-value *S* or *U* is created for each of the SELECT and UNION clauses respectively.

Example 3: Creation and Evaluation of r-nodes

In order to illustrate the creation and evaluation of r-nodes, another query extracted from the case study [Section 4], has been selected as it includes the most common SQL clauses: JOIN and WHERE with two conditions, a GROUP BY based on two attributes and two aggregate functions.

This query calculates the sum and average of the spent time in the actions carried out on tickets for each creator and type of the invoiceable ones:

```
SELECT H.creatorID,T.typeID,sum(H.timeSpent),avg(H.timeSpent)
FROM ticket T INNER JOIN history H ON T.ticketID=H.ticketID
WHERE T.invoiceable=1
```

```
GROUP BY H.creatorID, T.typeID
```

There is a set of four r-nodes for evaluating groupings, aggregate functions and quantifiers: two r-nodes for grouping columns, the attributes *creatorID* and *typeID*, one for the attribute *timeSpent* in the aggregate functions (in this case, there are two functions sharing the same attribute) and the last one for the SELECT clause (automatically labelled as *'U'* because of GROUP BY clause).

Assume the initial test inputs presented in [Tab. 3] for evaluating the r-nodes of GROUP BY clause and aggregate functions. Note that, although the query used in the Examples 1 and 2 has the same FROM and WHERE clauses, their test inputs can not be used because groups can not form with the single row selected as output [see Tab. 2].

ticket			**history**				**JOIN/WHERE Output**		
ticketID	**invoiceable**	**typeID**	**historyID**	**ticketID**	**creatorID**	**timeSpent**	**creatorID**	**typeID**	**timeSpent**
1	1	81	11	1	91	8.0	91	81	8.0
2	0	89	12	3	99	9.9	91	81	8.0
3	0	89	13	1	91	8.0	91	82	3.0
4	1	89	14	5	91	3.0	91	82	NULL
5	1	82	15	6	91	NULL			
6	1	82							

Table 3: Initial test inputs for a simple query with a GROUP BY clause and aggregate functions

After the evaluation of grouping and the aggregate function attribute, the initial test inputs cover:

- The r-value *G* of *typeID* because the output of JOIN and WHERE clauses produce two groups of more than one row in which *typeID* is different.
- The aggregation with null r-value (*AFn*) with a grouping formed by a null value and a non-null value (3.0).

[Fig. 4] includes the r-nodes after the evaluation where the covered situations are labelled as *'Y'*.

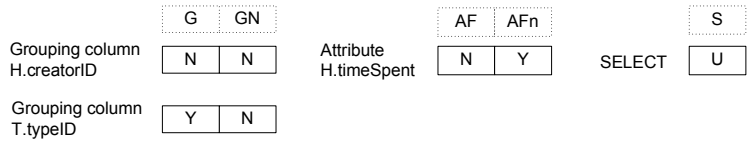

Figure 4: r-nodes after initial evaluation for a simple query with a GROUP BY clause and aggregate functions

The evaluation of the r-nodes indicates that there are situations not covered with the initial test inputs or unreachable. The tester should examine the r-values and determine each case. The r-values *GN* are unreachable for both *creatorID* and *typeID* (because they never are NULL). Then they are labelled manually as *'U'*. The rest of r-values may be covered if test inputs are completed with the aim of reaching them. For example:

- For r-value *G* of the grouping column *creatorID*, the output should include a group of rows where *creatorID* ≠91 and *typeID*=81 or *typeID*=82. Then, a new detail (16,5,92,NULL), for the ticket indexed by 5, and a new ticket (7,1,82) with its detail (17,7,92,NULL) are added and a new grouping is produced.
- For the multiple aggregation r-value (*AF)*, there should be at least three values (two of them being equal). As each grouping is composed of no more than two rows, another *history* (18,8,91,4.0) is added along with its referenced *ticket* (8,1,81). In this detail, the value 4.0 is assigned to the *timeSpent* that is different from the rest of time spent of this group.

With the final database [see Tab. 4], the r-nodes are evaluated and all r-values are covered.

ticket			**history**				**GROUP BY Output**			
ticketID	**Invoiceable**	**typeID**	**historyID**	**ticketID**	**creatorID**	**timeSpent**	**creatorID**	**typeID**	**sum**	**avg**
1	1	81	11	1	91	8.0	91	81	20.0	6.66
2	0	89	12	3	99	9.9	91	82	3.0	3.0
3	0	89	13	1	91	8.0	92	82	NULL	NULL
4	1	89	14	5	91	3.0				
5	1	82	15	6	91	NULL				
6	1	82	16	5	92	NULL				
7	1	82	17	7	92	NULL				
8	1	81	18	8	91	4.0				

Table 4: Test inputs for a simple query with GROUP BY clause and aggregate functions

3.3.4 Selecting after Grouping

The output produced by a query with a GROUP BY clause can be further filtered by a HAVING clause (which behaves similarly to the WHERE clause). The HAVING clause specifies an expression over a set of conditions $CH=(H_1, \ldots H_h)$ so as to restrict the output to those tuples that verify these conditions.

In order to evaluate the coverage, a new condition coverage tree over the having conditions *CT(CH)* is constructed and evaluated as has been described in [Subsection 3.2]. For conditions in the HAVING clause, the condition coverage tree is an independent structure. The particularities of its evaluation are two: (1) the test inputs are the tuples retrieved after grouping and, therefore, they are considered as a single

relation and (2) the c-values can not labelled as impossible since constraints defined in database schema are not valid due to the joins and GROUP BY clause of the query.

Example 4: Creation and Evaluation of a Condition Coverage Tree for a HAVING clause

In this example, the coverage of conditions of a HAVING clause is determined. For this, a condition coverage tree is created and evaluated as shown below.

The following query selects the average of the spent time in the actions carried out on tickets for each creator and type of the invoiceable ones when the sum of the time is less than an 8-hour working day:

```
SELECT H.creatorID,T.typeID,sum(H.timeSpent),avg(H.timeSpent)
FROM ticket T INNER JOIN history H ON T.ticketID=H.ticketID
WHERE T.invoiceable=1
GROUP BY H.creatorID, T.typeID
HAVING sum(H.timeSpent<8.0)
```

For the HAVING clause, the condition coverage tree is composed of a single c-node for the condition *sum(H.timeSpent)<8.0* [see Fig. 5]. The c-values initially labelled as unreachable are *Nr* and *Nb* because the constant '8.0' cannot be NULL.

As all clauses of the query, except HAVING, are the same as used in Example 3, then condition coverage tree for HAVING clause is evaluated using the test inputs at [Tab. 4]. The algorithm will use the set of rows retrieved by groupings (the last columns presented in this table).

After the evaluation, the c-values covered are [see Fig. 5]:

- *Nl*, because there is a row where *sum(H.timeSpent)* is NULL.
- *T* and *Fl* because there are rows where *sum(H.timeSpent)* is lower and higher than '8.0' respectively.

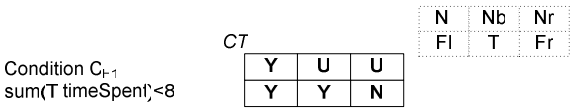

Figure 5: Condition coverage tree for HAVING clause

The tester may complete the test inputs for reaching the non-covered c-value *Fr*: it is necessary that, in all rows of the output retrieved by GROUP BY clause, the *sum(H.timeSpent)* value is higher than '8' what it is impossible using the same test database. In this case, a new test database should be created and the rows inserted for *ticket* and *history* tables should satisfy this condition, for example the rows presented in [Tab. 5].

ticket			**history**				**GROUP BY Output**			
ticketID	**invoiceable**	**typeID**	**historyID**	**ticketID**	**creatorID**	**timeSpent**	**creatorID**	**typeID**	**sum**	**Avg**
1	1	81	11	1	91	1.0	91	81	6.5	2.17
8	1	81	13	1	91	1.5				
			18	8	91	4.0				

Table 5: New test database to complete the test suites for a query with HAVING clause

3.4 Putting It All Together

The main goal of the evaluation of the coverage of a query is to obtain a metric to be used as an adequacy criterion of the test suite designed to exercise a query. Depending on the structure of the query under test, one or more condition coverage trees and sets of r-nodes must be evaluated:

- If the query has a SELECT with JOIN and/or WHERE, then a condition coverage tree *CT(CS)* has to be evaluated.
- If the query includes HAVING, another condition coverage tree *CT(CH)* has to be evaluated.
- If there is more than one query being combined by UNION, a separate condition coverage tree has to be evaluated for each query.
- Furthermore, a set of r-nodes has to be evaluated to consider the groupings and aggregate functions for each query and set quantifiers for SELECT and UNION clauses.

If a test suite is composed of several test groups, the evaluation of all c-nodes and r-nodes is repeated for each test input and then combined as follows: If a value has been covered by at least one test input, then it is labelled as *'Y'* and if it has not been covered by any case as *'N'*. Impossible values (*'I'*) and unreachable values (*'U'*) are the same for each test input, since the former depend solely on the database schema and the latter depend on the conditions of the query.

After evaluating all condition coverage trees over all test inputs, some of the values are covered and some others remain uncovered. The *theoretical condition coverage* is calculated as the percentage of covered c-values. As this measure does not consider impossible c-values, it is necessary to define another that takes them into account, the *schema condition coverage (c-coverage)*:

$$c\text{-}coverage = \frac{sum(covered\ c\text{-}values)}{sum(total\ c\text{-}values) - sum(impossible\ c\text{-}values)} \times 100$$

The *r-coverage* is calculated in the same way by taking into account the sum of r-values. Moreover, in this case there are not impossible r-values in the set of r-nodes as has been explained in [Subsection 3.3]:

$$r\text{-}coverage = \frac{sum(covered\ r\text{-}values)}{sum(total\ r\text{-}values)} \times 100$$

Due to the fact that the number of c-values grows exponentially by conditions and the number of r-values grows lineally, both coverage measures are maintained separate, as there are likely to be many more c-values than r-values and hence integrating them in a single value would distort the interpretation of the results.

The procedure for calculating coverage has been described above and is automated. Taking a query and a populated database, the query and the filter views are executed and every c-value and r-value is produced as output with an indication as to whether it has been covered. By exploring the restrictions in the database schema and the query, the impossible and some unreachable values are also calculated. It is up to the tester to determine whether there are non-covered values that are unreachable.

When unreachable values exist, the 100% of c-coverage and r-coverage are not reached but they have the superior limits:

$$maximum\ c\text{-}coverage = (1 - \frac{sum(unreachable\ c\text{-}values)}{sum(total\ c\text{-}values) - sum(impossible\ c\text{-}values)}) \times 100$$

$$maximum\ r\text{-}coverage = (1 - \frac{sum(unreachable\ r\text{-}values)}{sum(total\ r\text{-}values)}) \times 100.$$

Besides defining an adequacy criterion for the test suite designed, another goal is to use coverage as a test input selection criterion. The procedure for test input design consists in loading an initial set of data and query parameters (test inputs). For all test inputs, the coverage is evaluated automatically and c-values and r-values are labelled as *'Y'* whenever they have been covered by at least one test input. The user subsequently inspects the c-values and r-values that have not yet been covered. Based on the values previously covered, the tester may add new test inputs by trying to cover more values (usually by selecting different query parameters or by adding tuples to the database). On some occasions, inserting new information in the test database can result in covered values no longer being covered. When this situation occurs, although it is preferable to have only one database, the new tuples must be added in a different test database with a few tuples specific for this situation (this case has been illustrated in Example 4).

Note that the evaluation of coverage is performed automatically, although the process of completing the test inputs in order to increase the coverage depends on the user, so different test suites may be developed by the tester. At the end of the process, the same coverage must be obtained.

Example 5: Coverage Calculation
Continuing with the queries used in examples of previous sections, the coverage measures are calculated obtaining for:

- Examples 2 and 4: $maximum\ c\text{-}coverage=(1-\frac{2+2}{(24-14)+(6-0)})\times 100 = 75.0\%$. and $c\text{-}coverage = \frac{8+4}{(24-14)+(6-0)}\times 100 = 75.0\%$.
- Example 3: $maximum\ r\text{-}coverage= (1-\frac{1+1+0+1}{2+2+2+1})\times 100 = 57.1\%$ *and* $r\text{-}coverage=\frac{1+1+2+0}{2+2+2+1}\times 100 = 57.1\%$

As can be observed, both c-coverage and r-coverage equals the maximum coverage, respectively. So, selected test suites are complete according to the criteria established.

4 A Case Study

In order to illustrate the use of SQL coverage as a test selection criterion, it was used to develop the test inputs for the queries that are part of the business logic controlling the security access to user requests managed by a web-based helpdesk system that is currently in use at the University of Oviedo. This system manages general requests for end-user technical assistance, along with software change requests for corporate applications and time tracking. To date, it has been used by more than 200 users and has managed around 25,000 requests. Security control is an extremely important issue, since the system is used by many people with different profiles and levels of responsibility, the stored information is often highly confidential and the application is open to the web.

4.1 Brief Description of The System Under Test

The main information stored in the system is the *helpdesk ticket*, which is created for each user request. Each *ticket* has a *receiver* (the person who records it) and an *owner* (the person responsible for carrying out some action on it). Whenever an action is performed on a ticket, a *history* record is created including annotations, time tracking information, attachments and changes in the ticket state and owner.

The implementation of the security controls is centralized in a function that receives the ticket identifier, the type of object to grant or deny access to (ticket, history or others), the type of transaction (read, update or insertion), as well as the identifier of the user who performs the transaction. Before starting each transaction, the security function executes the SQL queries that include the database searches needed for deciding whether to grant or deny access. These queries are embedded in the procedural code and their specification along with their SQL implementation is displayed in [Tab. 6] and [Tab. 7] respectively. Nine queries and a view are used to control access to tickets and history records. Two more queries related to time tracking and invoicing are also included for exercising the grouping operations.

Query	Specification
Q11	A user having access type “OR” has read access to those tickets in which he/she is either the receiver or the current owner.
Q12	A user having access type “OO” has read access to those tickets in which he/she is either the receiver or the current owner or he/she has been the owner of the ticket sometime in the past.
Q13	A user having access type “OU” has read access to those tickets in which some of the following conditions are met: (1) he/she belongs to the same organizational unit as that of the receiver of the ticket, (2) he/she belongs to the same organizational unit as that of the current owner of the ticket, (3) he/she belongs to the same organizational unit as that of at least some user who has been the owner of the ticket sometime in the past.
Q21	A user has update access to those tickets in which either he/she is the receiver or he/she has privilege over the owner as indicated by the specification of the view V (userPermissions).
Q22	A user has update access to those tickets that belong to a type for which privileges have been explicitly stated for such a user in the typePermissions table.
Q31	A user has update access to those history records that have been created by him/her.
Q32	A user having access type “OR” can read the history and insert new history records if he/she is either the receiver or the current owner of the ticket to which the history record belongs.
Q41	A user having an access type different to “OR” can read the history and insert new history records if either he/she is the receiver of the ticket to which the history record belongs or he/she has privilege over the current owner of that ticket as indicated by the specification of the view V (userPermissions).
Q42	A user having an access type different to “OR” can read the history and insert new history records to those tickets that belong to some type for which privileges have been explicitly stated for that user in the typePermissions table.
V	This view determines which users have special privileges over others: A user, U1, has special privileges over another, U2, if both the following conditions are met: (1) either both users belong to the same organizational unit or user U1 belongs to the privileged organizational unit whose id is 310, (2) User U1 is flagged as responsible for his/her unit or both users are the same or user U1 represents the organizational unit to which U1 belongs.
G1	List the users, type of the ticket, the total and average time spent on all history records in invoiceable tickets for each user and type.
G2	List the number of different tickets that have been processed and total time spent for each month and organizational unit. Invoiceable and non-invoiceable tickets must be separated in different records.

Table 6: Specifications of queries under test

Query	SQL implementation
Q11	SELECT ticketID FROM ticket WHERE (ticketID=@1) AND ((receiverID=@2) OR (ownerID=@2))
Q12	SELECT T.ticketID FROM ticket T LEFT JOIN history H ON T.ticketID=H.ticketID WHERE (T.ticketID=@1) AND ((T.receiverID=@2) OR (T.ownerID=@2) OR (H.previousOwner=@2))
Q13	SELECT T.ticketID FROM ticket T LEFT JOIN user U0 ON T.receiverID=U0.userID WHERE (T.ticketID = @2) AND (U0.areaID=@1) UNION SELECT T.ticketID FROM ticket T LEFT JOIN user U1 ON T.ownerID = U1.userID WHERE (T.ticketID = @2) AND (U1.areaID = @1) UNION SELECT T.ticketID FROM ticket T LEFT JOIN history H ON T.ticketID = H.ticketID LEFT JOIN user U2 ON H.previousOwner = U2.userID WHERE (T.ticketID = @2) AND (U2.areaID = @1)
Q21	SELECT receiverID , ownerID FROM ticket T LEFT JOIN userPermissions P ON T.ownerID = P.IDover WHERE (T.ticketID=@1) AND ((T.receiverID=@2) OR ((P.ID=@2)))
Q22	SELECT T.ticketID , T.typeID , TP.userID FROM ticket T LEFT JOIN typePermissions TP ON T.typeID = TP.typeID WHERE (T.ticketID=@1) AND (TP.userID=@2)
Q31	SELECT historyID FROM history WHERE (historyID = @1) AND (creatorID = @2)
Q32	SELECT H.historyID FROM history H LEFT JOIN ticket T ON H.ticketID = T.ticketID WHERE (H.historyID=@1) AND ((T.receiverID=@2) OR (T.ownerID=@2))
Q41	SELECT T.receiverID, T.ownerID FROM ticket T LEFT JOIN history H ON T.ticketID = H.ticketID LEFT JOIN userPermissions P ON T.ownerID = P.IDover WHERE (H.historyID=@1)AND((T.receiverID=@2) OR ((P.ID=@2)))
Q42	SELECT H.historyID , T.typeID , TP.userID FROM ticket T LEFT JOIN history H ON T.ticketID = H.ticketID LEFT JOIN typePermissions TP ON T.typeID = TP.typeID WHERE (H.historyID = @1) AND (TP.userID = @2)
V	CREATE VIEW userPermissions AS SELECT L.ID AS ID , L1.ID AS IDover FROM user L1, user L WHERE ((L.areaID = L1.areaID) OR (L.areaID = 310)) AND ((L.responsible <> 0) OR (L.ID = L1.ID) OR (L.areaID = L1.ID))
G1	SELECT H.creatorID,T.typeID,SUM(H.timeSpent),AVG(H.timeSpent) FROM ticket T INNER JOIN history H ON T.ticketID = H.ticketID WHERE T.invoiceable = 1 GROUP BY H.creatorID , T.typeID
G2	SELECT convert(varchar (6),H.date,112) , U.areaID , T.invoiceable , COUNT(DISTINCT T.ticketID), SUM(H.timeSpent) FROM history H LEFT JOIN user U ON H.creatorID = U.userID LEFT JOIN ticket T ON H.ticketID = T.ticketID GROUP BY U.areaID , convert(varchar (6),H.date,112) , T.invoiceable

Table 7: SQL implementation of queries under test

4.2 Using the Coverage to Select Test Inputs

[Tab. 8] summarizes all the results obtained after using coverage as a criterion for selecting the test inputs for all queries until attaining the maximum coverage of *c-coverage* and *r-coverage*.

		Q11	Q12	Q13	Q21	Q22	Q31	Q32	Q41	Q42	V	G1	G2
Metrics for the queries under test	**Parameters**	2	2	2	2	2	2	2	2	2	0	0	0
	Joined tables	1	2	2+2 +3	2	2	1	2	2	3	2	2	3
	Conditions	3	5	3+3 +4	4+5	3	2	4	5+5	4	5	2	2
c-values	**Total**	78	726	78+78 + 240	240 +726	78	24	240	726+ 726	240	726	24	24
	I	26	296	32+32 + 144	55+ 92	27	8	134	354+ 92	136	92	14	20
	U	9	126	0+0+ 0	97+ 510	0	0	18	194+ 510	0	510	2	0
	Y	43	304	46+46 + 96	88+ 124	51	16	88	178+ 124	104	124	8	4
r-values	**Total**	1	1	1+1+ 1+2	1+1	1	1	1	1+1	1	1	7	11
	U	1	0	1+1+ 0+1	0+1	1	1	1	0+1	1	1	3	4
	Y	0	1	0+0+ 1+1	1+0	0	0	0	1+0	0	0	4	7
(a) using the same DB	**Test inputs**	11	27	8	9	10	7	15	12	9	2	1	1
	Rows *	16	16	16	16	16	16	16	16	16	8	16	16
(b) using separate DBs	**Test inputs**	11	25	7	7	10	7	14	9	11	2	1	1
	Rows *	2	7	3	1	2	1	4	2	4	8	8	5
%coverage	**c-coverage**	82.7	70.7	100	25.9	100	100	83.0	30.0	100	19.6	80.0	100
	r-coverage	-	100	40	50	-	-	-	50	-	-	57.1	63.6

Table 8: Characteristics of queries and test inputs for attaining the maximum coverage. (*)The number of rows refers to the ticket table with the exception of the view (query V), where it refers to the users table. In the latter case, the number of test inputs is two, one for each test database instance

The first group of rows (Metrics for the queries under test) provides information about the number of parameters, joined tables and conditions in each query. Query Q13 is a UNION of three SELECT statements, and the information is provided separately for each condition coverage tree. Moreover, queries Q21 and Q41 join a table and a view (V), therefore the information provided includes the query and the view values, respectively.

The second and third groups of rows (c-values and r-values) quantify the number of c-values and r-values respectively (total, impossible, unreachable and covered).

In the fourth and fifth groups of rows (using the same DB, using separate DBs), information is provided about the test inputs that have been selected until the maximum coverage is attained. Two different approaches have been followed:

1. The same database for all queries: After generating the test inputs for a query, the following cases are generated using the same database as was used for the previous query by adding rows when needed. The size of the database (rows) is the same for all queries, with the only exception of the view, which needs two different databases (different in only one value of a row) and refers to the *users* table.
2. A separate database for each query: Test inputs for each query are developed from scratch, by adding only the rows that are needed for testing that query.

The former approach gives a single, larger database (16 rows in the tickets table) and the latter many small databases. However, the number of tests is similar in both cases.

The last group of rows (%coverage) provides the percentage of coverage (*c-coverage* and *r-coverage*) reached for each query. The values obtained for these measures are the maximum coverages regardless of the approach used to select test suites.

5 Fault Detection Capability

In this section a number of issues related to the kind of faults (both SQL-specific and non SQL-specific) that may be detected by using the coverage criterion described in this paper are discussed and certain considerations with respect to the test size are provided.

In software testing, one way of characterizing and comparing test selection and adequacy criteria is their effectiveness in detecting faults that may appear in a given program under test. In the absence of records concerning real faults that have occurred during the development and operational life of an application, artificially injected faults can be used. In order to do so in a systematic way, a mutation testing approach [DeMillo et al. 1978] [Hamlet 1977] can be used. The mutation analysis consists in generating a large number of alternative programs called mutants, each one having a simple fault that consists of a single syntactic change in the original program. Each mutant is executed with the test suite and when it produces an incorrect output (the output is different to that of the original program), the mutant is said to be killed. Some mutants always produce the same output as the original program, so no test input can kill them. These mutants are said to be equivalent mutants. After executing the test suite over a number of mutants, the mutation score is defined as the percentage of dead mutants divided by the number of non-equivalent mutants.

5.1 Mutation Analysis

For each of the queries in the case study [Section 4], a set of mutated queries has been developed using (1) some of the mutation operators that are described by [King, Offutt 1991] applied to the query conditions and (2) others defined in [Tuya et al. 2007] in order to check other kinds of faults specifically related to the SQL language.

For these queries, the used mutation operators are:

- OR (*Operator Replacement mutation operators*) adapt the *Expression modification operators* known as the "sufficient mutation operators" [Offutt et al. 1996]:
  - LCR (*Logical Connector Replacement*): Each occurrence of {AND, OR} is replaced by the other, by *falseop* (always returns false), by *trueop* (always returns true), by *leftop* (returns the left operand), and by *rightop* (returns the right operand).
  - ROR (*Relational Operator Replacement*): Each occurrence of {=, !=, <, <=, >, >=} is replaced by each of the other operators, by *falseop* and by *trueop*.
  - UOI (*Unary Operator Insertion*). Each reference to a parameter or column *e* is replaced by *-e*, *e+1* and *e-1*.
- IR (*Identifier Replacement mutation operators*) adapt the *Replacement-of-operand* operators. Every column (IRC), constant (IRT) and parameter (IRP) reference is replaced by its counterpart column, constant or parameter of the query with type compatible.
- SC (*SQL clause mutation operators)* are defined to mutate the main SQL clauses:
  - SEL (*SELECT clause*): Each occurrence of {SELECT, SELECT DISTINCT} is replaced by the other.
  - JOI (*Join clause)*. Each occurrence of {INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, CROSS JOIN} is replaced by each of the others.
  - GRU (*Grouping*): Each of the grouping columns is removed.
  - AGR (*Aggregate functions*): Each occurrence of an aggregate function (including functions quantified with DISTINCT) is replaced by each of the others.
  - UNI (*Query concatenations*): Each occurrence of {UNION, UNION ALL} is replaced by the other.
- NL (*NULL mutation operators)* generate mutations related to the handling of null values:
  - NLS (*NULL in select list*): each item *c* in select list (if can be NULL) is replaced by a function *f(c,r)* that substitutes the null value of *c* by *r*.
  - NLI (*Include nulls*): for each attribute *a* of a condition *C* in the form *a ℜ b* or *b ℜ a*, the condition is replaced by the expression *C OR a IS NULL*.

The mutation operators are applied to the case study queries. The corresponding mutated queries are generated in an automated way using SQLMutation tool [Tuya et al. 2006] obtaining the results summarized in [Tab. 9]. This table displays the number of mutants generated automatically (998 non-equivalent mutants in total) running these mutation operators, the number of dead mutants with the test suite and the mutation scores for each query and each of the above operators.

Mutation coverage for the LCR operators, replacement operators (IR) and specific operators to SQL (SC and NL) attain a score of 100%, UOI score is 99% (only one live mutant corresponding to the replacement of a=k by a+1=k) and ROR

score is 93% (16 non-killed ROR mutants consisting of the replacement of = by ≥or ≤). The total score is 98,3%, only 17 of 981 mutants are alive. These mutants are related to cover boundary values because this criterion has not been considered in the approach of this work.

		Q11	Q12	Q13	Q21	Q22	Q31	Q32	Q41	Q42	V	G1	G2	Num Mut.	Dead. Mut.	% score
OR	LCR	10	14	15	10	5	5	10	10	5	19			103	103	100
	ROR	21	28	42	21	14	14	21	21	14	31	3		230	214	93.0
	UOI	9	12	18	9	6	6	9	9	6	21			105	104	99.0
IR	IRC	10	27	42	29	22	4	20	41	32	27	20	30	304	304	100
	IRP	9	16	12	12	6	4	9	12	6				86	86	100
	IRT										9	3	14	26	26	100
SC	SEL		1		1			1	1	1	1		1	7	7	100
	JOI		4	16	4	4		4	8	8	4	4	8	64	64	100
	GRU											4	8	12	12	100
	AGR											14	14	28	28	100
	UNI			5										5	5	
NL	NLS											2	2	4	4	100
	NLI		4	12							8			24	24	100
Num. of mutants		59	106	162	86	57	33	74	102	72	120	50	77	998		
Dead mutants		56	106	157	82	55	32	73	101	72	120	50	77		981	
%score		94,9	100	96,9	95,3	96,5	97,0	98,6	99,0	100	100	100	100			98,3%

Table 9: Mutation scores for queries under test

5.2 Limitations of the Analysis

Two main potential threats to the validity of the approach may limit the ability to generalize the mutation analysis results.

The first concern is related to the queries used in the case study. All queries of the example are taken from a real application currently in use, and they use a relatively wide variety of SQL clauses. However, they constitute a limited set of the kind and complexity of queries that can be found in real applications.

The second is whether the set of SQL mutants used to evaluate the fault detection ability are representative of real-life faults. Some of the mutation operators are the "sufficient mutation operators" that have been widely used in the literature with this aim [Offutt, Untch 2000], although used to seed faults in procedural programs. Other mutant operators introduce changes in SQL clauses that are likely to represent single faults introduced by a programmer. If it is assumed that SQL mutants behave like mutants for imperative code, empirical studies comparing test suites on hand-seeded, automatically generated (mutation) and real-world faults suggest that the generated mutants provide a good indication of the fault detection capability of a test suite [Andrews et al. 2005]. Also, the mutation approach has been used in [Tsai et al. 1990], to perform a partial evaluation of the fault detection capability of database test cases, in [Deng et al. 2005] and [Elbaum et al. 2005] for seeding manual faults in queries in order to assess the effectiveness of test generation techniques, and in [Chan et al. 2005] for SQL queries based on the semantics of the conceptual model.

The effectiveness of the criteria developed in the present paper was evaluated using a controlled experiment in [Tuya et al. 2008] where the tester is guided using the criteria to select the database test cases. Authors compare them with other techniques and the results show that the use of the present criteria allows the tester to develop more effective test cases and the effectiveness is higher when considering the kind of faults that are more specifically related to SQL than others.

5.3 Other Detected Faults

Apart from detecting the injected faults using mutation operators as shown above, test inputs developed to attain the maximum c-coverage and r-coverage can detect a number of common faults in conditions and query clauses, which are illustrated with examples below.

5.3.1 Conditions

Here the simplest query (Q11) is considered, whose goal is to grant permission to a *ticketID* (given by the parameter @1) whenever either *ownerID* or *receiverID* is the user who accedes to it (given by the parameter @2):

```
SELECT ticketID FROM ticket
WHERE ticketID=@1 AND (receiverID=@2 OR ownerID=@2)
```

An alternate implementation of the WHERE conditions could be:

```
(ticketID=@1 AND receiverID=@2) OR (ticketID=@1 AND ownerID=@2)
```

And if the second reference to *ticketID* is removed, it could result in the following faulty query:

```
SELECT ticketID FROM ticket
WHERE (ticketID=@1 AND receiverID=@2) OR (ownerID=@2)
```

This fault could also be obtained by a change in the bracket grouping or by a removal of the brackets (because AND has precedence over OR).

Eleven test inputs were selected for this query until the maximum coverage was reached, along with a simple database with two rows in the tickets table. Eight of these cases include the possible combinations of the truth table for the conditions including *ticketID*, *ownerID* and *receiverID*, and three more include null values in the parameters, the 9^{th} test input being the one that detects the aforementioned faulty query. However, only four are needed to kill the above mutants, none of them is able to detect the faulty query.

5.3.2 Null Values

A common mistake that is made when writing queries is the misuse or misunderstanding of the effect of the three-valued logic used by SQL.

For example, consider the query Q11 again with an addition to its specification consisting in also granting access to users who are not the receivers of a ticket,

provided that the owner has not yet been assigned. In this case, the database schema would permit null values in the *ownerID* attribute. While selecting test inputs until reaching the maximum coverage, a row in the test database would be inserted having a null value in that field. The query would be demonstrated as faulty, since a row having a null value in *ownerID* would only be selected when both *ticketID* and *receiverID* match the parameters, the correct one being as follows:

```
SELECT ticketID FROM ticket
WHERE ticketID=@1 AND
  (receiverID IS NULL OR receiverID=@2 OR ownerID=@2)
```

5.3.3 JOIN Clauses and Conditions

The wrong use of join clauses and predicates for selecting rows is another common source of faults.

The query number Q22 grants permission to a ticket (@1) for a user (@2) when the type of the ticket and the user match some of the rows (*userID*,*typeID*) that are stored in the *typePermission* table.

```
SELECT T.ticketID, T.typeID, P.userID
FROM ticket T LEFT JOIN typePermission P ON T.typeID=P.typeID
WHERE T.ticketID=@1 AND P.userID=@2
```

The selected test inputs include database rows having tickets with and without related rows in the *typePermission* table. A faulty query permitting the selection of a ticket that has no related row in that table would be detected. An example of the WHERE condition for this situation would be the following:

```
T.ticketID=@1 AND (P.userID=@2 OR P.userID IS NULL)
```

Note that in this case, in which the joined attributes are used in the WHERE under the logical operator AND, the inner, left and right joins are equivalent. This is not applicable when the operator is OR. An example of this is query number Q12, which grants access to a ticket (@1) when the user (@2) matches either the *ownerID* or the *receiverID* or the *previousOwner* (this attribute is used to check whether he/she has been the owner sometime in the past):

```
SELECT T.ticketID
FROM ticket T
  LEFT JOIN history H ON T.ticketID = H.ticketID
WHERE T.ticketID=@1 AND
  (T.receiverID=@2 OR T.ownerID=@2 OR H.previousOwner=@2)
```

As in the previous query, the selected test inputs will include tickets without history that will make it possible to detect that a query using inner or right joins would be faulty.

5.3.4 Aggregate Functions

Another possible fault to be considered is that related to aggregate functions that would be detected by the test inputs selected for query G1 [see Tab. 4].

The average aggregate function *avg(a)*, where *a* is the *timeSpent* attribute, can be replaced by *sum(a)/count(a)*. As neither *sum(a)* nor *count(a)* will take into account the null values, the evaluation of the aggregate function for the second grouping (91,82,3.0,3.0) originated by grouping two rows (one of them having a null value) is 3.0 under both implementations. However, if *count(a)* is replaced by *count(*)*, then the count would be 2, as *count(*)* includes every row. Hence, the average would be 1.5 and therefore the fault would be detected.

5.4 Considerations about the Size of the Test Suite

At first glance, one of the main drawbacks of the approach used for the evaluation of condition coverage is the size of the test suite. Multiple condition coverage is expensive, because in order to cover *n* conditions, a set of 2^n inputs are needed. In this case, in which each condition is evaluated in relation to more than two outcomes (the c-values), the theoretical size is much larger. If there are *n* conditions, then the number of c-nodes is calculated as $1+3^1+3^2+\ldots 3^{n-1}$ and the total number of c-values is six times this value: $total\ c\text{-}values=(3^n-1)\times 3$.

In practice, however, the number of cases is likely to be significantly lower than this theoretical limit, since each test input may cover many c-values. Moreover, test inputs are not required to cover the impossible and unreachable c-values.

Consider the data presented in [Tab. 8]. The queries presenting the most conditions are Q12, Q41 and V (five conditions each). Query Q12 is the one with the largest number of test inputs (25 cases in the last group of rows, far from the total number of n-values), whereas query Q41, with a similar structure (two joined tables), needs 9 test inputs and query V only 2. The number of test inputs is significantly different depending on the kind of query: queries Q12 and Q41 have very small databases and references to the parameters in the conditions and V has a larger database and no parameters. The reason is that the execution of Q12 and Q41 selects a reduced number of rows from the database and therefore the number of possible situations being exercised for each test case is low, whereas query V selects many rows from the database and therefore the number of possible situations being exercised is high.

The size of the test suite is not only limited as a result of a test input exercising many c-values, but also because many c-values are known to be impossible (*'I'*), making the entire sub-tree impossible. Moreover, if a c-value is detected as unreachable (*'U'*) and then the entire sub-tree likewise becomes unreachable. This issue may be checked in [Tab. 8] by calculating the percentage of impossible and unreachable c-values with respect to the total, 36% and 46% respectively. The query presenting the most unreachable c-values is V, accounting for 70% of the unreachable values, as the query is a recursive join that imposes some restrictions that make many nodes unreachable.

Nevertheless, it might still be possible to reduce the breadth expansion of the trees. A first approach would consist in using two separate trees, one for the JOIN condition and another for the WHERE. A second approach would consist in limiting

the creation of sub-trees solely to the branches generated by the *T* c-value and cutting off the sub-trees for the *Fl* and *Fr* c-values. The former approach would maintain a multiple coverage criterion if considered independently for JOIN and WHERE, whilst the latter would not. However, this could mask certain kinds of faults, as is illustrated below with an example:

Considering again query G1, though with the join incorrectly stated as LEFT JOIN instead of INNER JOIN:

```
SELECT H.creatorID,T.typeID,SUM(H.timeSpent),AVG(H.timeSpent)
FROM ticket T LEFT JOIN history H ON T.ticketID = H.ticketID
WHERE T.invoiceable=1
GROUP BY H.creatorID, T.typeID
```

To cover the JOIN condition, the test inputs selected consist of the rows of *ticket* indexed by 1 and 4 and the *history* indexed by 11 (see the test inputs in [Tab. 4]). If the evaluation of the WHERE condition is made separately, then the above cases will cover the *T* c-value and it is only necessary to add a row to cover *Fl*, for instance the ticket indexed by 2. The output produced is {(91,81,8.0,8.0)}, which is correct. If the query is subsequently executed after adding the ticket indexed by 2 to the database, then the output is {(NULL,89,0,0), (91,81,8.0,8.0)}, which is not correct, as it refers to a non-existent user. In this case the approach of considering the JOIN and WHERE conditions separately has masked this fault.

5.5 Considerations about the Complexity of the Algorithm of Evaluation

The algorithm of evaluation of the condition coverage trees evaluates all possible combinations of c-values using every test input. Its complexity depends on two variables: the number of conditions (n) and the number of tuples (t) of each relation joined in the query to be evaluated. It has the worst-case complexity $O(t^n)$ when it is necessary to cover all *Fr* or *Fl* c-values since the root c-node.

However, considering the queries Q12, Q41 and V (each with five conditions) and using the same database for all queries [see Tab. 8] the time consumed by the algorithm in the evaluation was less than 1 second. In another experiment, for the evaluation of the query Q41, a different test database was used. In this new example, where the *user* table had 49 rows and *userPermission* view returned 1652 rows, the time consumed was 13 seconds. This time is low compared with the time necessary to manually evaluate queries and calculate their coverage, or to determine if test inputs are complete for a query.

In the common scenarios, the algorithm will be used for helping to develop the test suite for a set of queries and test databases will have a limited number of tuples. In these cases, the time consumed for evaluating of the coverage is low.

6 Related Work

Even though a great deal of research in software testing has been carried out in recent years, few studies have been specifically related to the testing of database applications, whether for test input selection criteria or test input adequacy criteria.

An initial way of classifying the related work is in relation to the information used to meet the criteria: only the database, only the queries, and both of them.

The selection of test inputs by means of considering the database schema and constraints, though not the application code is the approach taken by [Davies et al. 2000]. In order to automatically load the initial database, a set of valid and invalid data is generated from a database schema considering primary keys, null values and established ranges, but not referential integrity. Besides basing on database schema, [Wu et al. 2003] select the test inputs using a set of non-deterministic rules like associations, correlations and patterns, and statistics of the current live data in the production database. In both cases, neither the SQL statements that are executed nor adequacy criteria are considered. In this paper, coverage criterion is considered and it is calculated taking into account parts of the database schema where the relations of the query are defined as well as null values, primary keys and foreign keys.

[Chan, Cheung 1999] take into consideration only SQL queries. The queries are translated to C and then testing can be carried out by using conventional white-box testing techniques. [Gardikiotis, Malevris 2006] present two approaches where control flow graphs are applied to generate test cases. In the first, "gray-box" testing method is used considering SQL statements as black-box and analyzing the imperative code as white-box. In the second approach, SQL statements are translated to the imperative code and then the control flow graph is generated. Instead of translating the query to be tested into C or into another programming language, the approach of this paper represents the conditions of the query by the coverage tree and incorporates a notion of a white-box technique (multiple condition coverage) for evaluating the tree.

The structure of the data and the query under test is considered in most studies on database testing. [Mannila, Räihä 1986] present a theoretical approach using relational algebra and a notion of adequacy related to the concept of an Armstrong database. Queries, with select, project and join operations, expressed in relational algebra are represented as query graphs to be evaluated and a test database is generated for each given query after evaluating functional dependencies obtained from the database schema and the query. In the present paper, apart from clauses included in [Mannila, Räihä 1986], parameters, grouping operations, aggregate functions and set quantifiers are considered, and selected test inputs for a query are evaluated according to a defined adequacy criterion.

Algebra relational is also used by [Tsai et al. 1990] to specify queries. A set of predicates are obtained and translated into sets of systems of linear inequalities from which the authors derive the test inputs. The effectiveness of the test inputs, like in the present paper, is compared with mutated queries.

The use of general purpose constraint solvers for test data generation is the approach taken by [Zhang et al. 2001]. The query is translated into a set of constraints which are solved to generate the test database.

A fairly elaborate and elegant approach for generating test inputs by means of considering the database schema together with SQL queries is performed by [Chays et al. 2004] in the AGENDA tool. The goal is to facilitate the testing of applications, composed of a single query, by automating the generation of the test database and test inputs for input parameters. Both are selected taking into account the database schema, information obtained from SQL statements and some heuristics and information provided by the tester, such as category-partitions, boundary values,

duplicate values and null values. Their adequacy criterion relies implicitly on these heuristics, although it is not defined explicitly. Test input selection is automated by performing this task in a semi-automatic way where tester intervention is required and illustrated using examples with one or two joins. Its performance was discussed in several small case studies but the effectiveness in detecting faults of the selected test database is not detailed. Our approach, in contrast to AGENDA tool, does not generate test database but defines coverage criteria that automatically evaluate test inputs for SQL queries.

Other authors also include AGENDA for update statements and extend the tool to deal with database transactions with multiples queries [Deng et al. 2005]. The main feature introduced in their approach is to check whether transactions are consistent with their requirements. Moreover, as in this paper, results of example applications with seeded faults are studied.

Adequacy criteria were defined to assess the quality of the tests manually designed by [Kapfhammer, Soffa 2003]. Their criteria make use of control flow and data flow techniques associated with database entities and queries. Adequate tests are those that exercise all database associations (relations, tuples, attributes and values of attributes) for all entities. An improved approach is given by [Willmor, Embury 2005]. Test adequacy criteria considers transactions (both committed and non-committed) and the define-use pairs for different database states resulting from the execution of previous statements. These works are others of the most similar to this paper because they establish explicitly defined adequacy criteria, although SQL semantics are not taken into consideration. [Halfond, Orso 2006] also define a test adequacy criterion based on the coverage of all the SQL statements dynamically-generated that an application can issue to a database.

The approaches presented in [Deng et al. 2005], [Chays et al. 2004], [Kapfhammer, Soffa 2003] and [Willmor, Embury 2005] are complementary to our approach and, in all cases, a common feature is that validation is performed using non-trivial systems or real-life applications, which inspires confidence in the capabilities of each method.

7 Conclusions

In this paper, two coverage criteria for SQL queries that retrieve information from a database have been defined. The criteria take into account both the database and the query structure to determine a set of situations (c-values and r-values) that must be exercised by the query when executed against the test suite. A quite complete set of the SQL syntax and semantics is supported, including joins, unions, groupings, aggregate functions, set quantifiers and non-existing information (NULL), as well as WHERE and HAVING clauses, and parameterised queries.

The SQL coverage criteria can then be used as a test selection criterion and guide the tester in completing the test inputs by seeking out non-covered situations. The effectiveness of the test inputs designed using the criterion were illustrated using a real-life example and were found to be adequate for detecting a number of kinds of frequent SQL faults (such as problems with null values, joins and aggregate functions).

The aim of SQL coverage has been to consider the coverage for the distinctive features and processing of the SQL language compared to procedural languages. Consequently, a number of issues, such as boundary value analysis of conditions, have not been considered, although this may be included as heuristic rules when developing test inputs. It can, moreover, be easily integrated in the testing process, since coverage is automatically calculated by using the query under test and live data from the database, along with the database schema. This allows the test input design and the adequacy evaluation to be performed in an interactive way and serves as a basis for automatic test data generation tools.

Acknowledgements

This research work was funded by the Department of Science and Innovation (Spain) and ERDF Funds, projects Test4SOA (TIN2007-67843-C06-01) and RePRIS (TIN2007-30391-E), and the Government of Castilla La-Mancha, project PRALIN (PCI-08-121-1374).

References

[Andrews et al. 2005] Andrews, J., Briand, L., Labiche, Y.: "Is mutation an appropriate tool for testing experiments?"; Proc. 27th Int. Conf. on Software Engineering, ACM Press, New York (2005), 402–411.

[Brass, Goldberg 2005] Brass, S., Goldberg, C.: "Semantic Errors in SQL Queries: A Quite Complete List (Extended version)"; Journal of Systems and Software 79, 5 (2005), 630-644.

[Chan, Cheung 1999] Chan, M.Y., Cheung, S.C.: "Testing Database Applications with SQL Semantics"; Proc. 2nd Int. Symp. on Cooperative Database Systems for Advanced Applications, March 1999, Springer, Singapore (1999), 363–374.

[Chan et al. 2005] Chan, W.K., Cheung, S.C., Tse, T.H.: "Fault-Based Testing of Database Application Programs with Conceptual Data Model"; Proc. the 5th Int. Conf. on Quality Software, September 2005, IEEE Computer Society (2005), 187-196.

[Chays et al. 2004] Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., Weyuker, E.J.: "An AGENDA for Testing Relational Database Applications"; Software Testing, Verification and Reliability 14, 1 (2004), 17-44.

[Davies et al. 2000] Davies, R.A., Beynon, R.J.A., Jones, B.F.: "Automating the Testing of Databases"; Proc. the First Int. Workshop on Automated Program Analysis, Testing and Verification, June 2000, IEEE Computer Society (2000).

[DeMillo et al. 1978] DeMillo, R.A., Lipton, R.J., Sayward, F.G.: "Hints on Test Data Selection: Help for the Practicing Programmer"; IEEE Computer 11, 4 (1978), 34-43.

[Deng et al. 2005] Deng, Y., Frankl, P., Chays, D.: "Testing Database Transactions with AGENDA"; Proc. the 27th Int. Conf. on Software Engineering, May 2005, ACM Press, New York (2005), 78-87.

[Elbaum et al. 2005] Elbaum, S., Rothermel, G., Karre, S., Fisher, I.I.M.: "Leveraging User-Session Data to Support Web Application Testing"; IEEE Transactions on Software Engineering, 31, 3 (2005), 187-202.

[Gardikiotis, Malevris 2006] Gardikiotis, S.K., Malevris, N.: "Program Analysis and Testing of Database Applications"; Proc. the 5th Int. Conf. on Computer and Information Science and 1st IEEE/ACIS Int. Workshop on Component-Based Software Engineering, Software Architecture and Reuse, July 2006, IEEE Computer Society (2006).

[Halfond, Orso 2006] Halfond, W.G.J., Orso, A.: "Command-Form Coverage for Testing Database Applications"; Proc. the 2nd IEEE/ACM Int. Conf. on Automated Software Engineering, 2006, 69-80.

[Hamlet 1977] Hamlet, R.G.: "Testing Programs with the Aid of a Compiler"; IEEE Transactions on Software Engineering, 3, 4 (1977), 270-290.

[Imielinski, Lipski 1984] Imielinski, T., Lipski Jr, W.: "Incomplete Information in Relational Databases"; Journal of the Association for Computing Machinery, 31, 4 (1984), 761-79.

[Kapfhammer, Soffa 2003] Kapfhammer, G.M., Soffa, M.L.: "A Family of Test Adequacy Criteria for Database-Driven Applications"; Proc. the 9th European Software Engineering Conf. and the 11th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering, September 2003, ACM Press, New York (2003), 98–107.

[King, Offutt 1991] King, K.N., Offutt, A.J.: "A Fortran Language System for Mutation-Based Software Testing"; Software Practice and Experience, 21, 7 (1991), 686-718.

[Klein 1994] Klein, H.J.: "How to Modify SQL Queries in Order to Guarantee Sure Answers"; ACM SIGMOD Record, 23, 3 (1994), 14-20.

[Lu et al. 1993] Lu, H., Chan, H.C., Wei, K.K.: "A Survey on Usage of SQL"; ACM SIGMOD Record, 22. 4 (1993), 60-65.

[Mannila, Räihä 1986] Mannila, H., Räihä, K.J.: "Test Data for Relational Queries"; Proc. the 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, March 1986, ACM Press, New York (1985), 217-223.

[Offutt, Untch 2000] Offutt, A.J., Untch, R.H.: "Mutation 2000: Uniting the Orthogonal"; Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, October 2000, 45-55.

[Offutt et al. 1996] Offutt, A.J., Lee, A., Rottermel, G., Untch, R.H., Zapf, C.: "An Experimental Determination of Sufficient Mutant Operators"; ACM Transactions on Software Engineering and Methodology, 5, 2 (1996), 99-118.

[Pönighaus 1995] Pönighaus, R.: "'Favourite' SQL-Statements – An Empirical Analysis of SQL-Usage in Commercial Applications"; Proc. the 6th Int. Conf. on Information Systems and Management of Data (Lecture Notes in Computer Science, vol. 1006), November 1995, Springer, Berlín (1995), 75-91.

[SQL 1992] International Organization for Standardization. Information technology -- Database languages – SQL, ISO/IEC 9075:1992, third edition.

[Suárez-Cabal, Tuya 2004] Suárez-Cabal, M.J., Tuya, J.: "Using an SQL Coverage Measurement for Testing Database Applications"; Proc. the ACM SIGSOFT Symp. on the Foundations of Software Engineering, October 2004, ACM Press, New York (2004), 253-262.

[Tsai et al. 1990] Tsai, W.T., Volovik, D., Keefe, T.F.: "Automated Test Input Generation for Programs Specified by Relational Algebra Queries"; IEEE Transactions on Software Engineering, 16, 3 (1990), 316-324.

[Tuya et al. 2008] Tuya, J., Dolado, J., Suárez-Cabal, M.J., De la Riva, C.: "A Controlled Experiment on White-box Database Testing"; ACM SIGSOFT Software Engineering Notes, 33, 1 (2008).

[Tuya et al. 2007] Tuya, J., Suárez-Cabal, M.J., De la Riva, C.: "Mutating Database Queries"; Information and Software Technology, 49, 4 (2007), 398-417.

[Tuya et al. 2006] Tuya, J., Suárez-Cabal, M.J., De la Riva, C.: "SQLMutation: a Tool to Generate Mutants of SQL Database Queries"; 2nd Workshop on Mutation Analysis (Mutation 2006), 2006.

[Vassiliou 1979] Vassiliou, Y.: "Null values in database management. A denotational semantics approach"; Proc. the 1979 ACM SIGMOD Int. Conf. on Management of Data, May 1979, ACM Press, New York (1979), 162-169.

[Willmor, Embury 2005] Willmor, D., Embury, S.M.: "Exploring Test Adequacy for Database Systems"; Proc. the 3rd UK Software Testing Research Workshop, September 2005.

[Woodward 2001] Woodward, M.: "Insights into Software Testing"; Software Focus 2, 3 (2001), 93-103.

[Wu et al. 2003] Wu, X., Wang, Y., Zheng, Y.: "Privacy Preserving Database Application Testing". Proc. the ACM Workshop on Privacy in Electronic Society, October 2003, ACM Press, New York (2003), 118-128.

[Zhang et al. 2001] Zhang, J., Xu, C., Cheung, S.C.: "Automatic Generation of Database Instances for White-Box Testing"; Proc. the 25th Annual Int. Computer Software and Applications Conf., October 2001, IEEE Computer Society Press, Los Alamitos (2001), 161–165.

[Zhu et al. 1997] Zhu, H., Hall, P.A.V., May, J.H.R.: "Software Unit Test Coverage and Adequacy"; ACM Computing Surveys 29, 4 (1997), 366-427.

Appendix I: Algorithm for evaluating conditions

Algorithm Eval_CCT(CT, TestDB) {

```
Let CT be the condition coverage tree for CSk
Let TestDB be tuples of test database for the evaluation of CT(CSk)

Let Ck be the condition in c-node(Ck) to evaluate in the form XℜZ

For each tuple, ti∈ r(Rx) where x=ti[X]
  For each tuple, tj∈ r(Rz) where z=tj[Z]
    If x is NULL
      Label Nl as 'Y'
    If z is NULL
      Label Nr as 'Y'
    If x is NULL and z is NULL
      Label Nb as 'Y'
    If c-value T is labelled as 'N' or 'Y'
      If xℜz is satisfied
        Label T as 'Y'
```

```
        Call Eval_CCT(CTT(CSk+1),TestDB-{r(Rx),r(Rz)}∪{ti∈r(Rx),tj∈r(Rz)}
        If all c-values of CTT(CSk+1) are different from 'N'
                exit Foreach
    End Foreach
    If c-value Fl is labelled as 'N' or 'Y'
      If (Ck was not satisfied during every iteration with x)
        Label Fl as 'Y'
        Call Eval_CCT(CTFl(CSk+1),TestDB-{r(Rx)}∪{ti∈r(Rx)}
        If all c-values of CTFl(CSk+1) are labelled different from 'N'
          exit Foreach
  End Foreach
  For each tuple, tj∈r(Rz) where z=tj[Z]
    For each tuple, ti∈r(Rx) where x=ti[X]
      If xℜz is satisfied
        exit Foreach
    End Foreach
    If c-value Fr is labelled as 'N' or 'Y'
      If (Ck was not satisfied during every iteration with z)
        Label Fl as 'Y'
        Call Eval_CCT(CTFr(CSk+1),TestDB-{r(Rz)}∪{tj∈r(Rz)}
        If all c-values of CTFr(CSk+1) are labelled different from 'N'
          exit Foreach
  End Foreach
}
```