# *Mutating database queries*

Javier Tuya *, Mª José Suárez-Cabal and Claudio de la Riva
Department of Computer Science, University of Oviedo,
Campus of Viesques, s/n, 33204 Gijón (SPAIN)

**Abstract.**
A set of mutation operators for SQL queries that retrieve information from a database is developed and tested against a set of queries drawn from the NIST SQL Conformance Test Suite. The mutation operators cover a wide spectrum of SQL features, including the handling of null values. Additional experiments are performed to explore whether the cost of executing mutants can be reduced using selective mutation or the test suite size can be reduced by using an appropriate ordering of the mutants. The SQL mutation approach can be helpful in assessing the adequacy of database test cases and their development, and as a tool for systematically injecting faults in order to compare different database testing techniques.

Keywords: Software testing, Database testing, SQL testing, Fault-based testing, Mutation testing, Test adequacy criteria

* Corresponding author: Tel. +34 985 182 049, FAX: +34 985 181 986, e-mail: tuya@uniovi.es

## 1. Introduction

Information Technology architecture and its associated infrastructure have changed dramatically over recent decades (hardware, middleware, programming languages, development tools, standards, web technology, etc.). However, the information stored in databases still usually relies on Codd's relational data model supported by relational database management systems and said information is manipulated using the Structured Query Language (SQL) [22], developed in the late 1970's. The huge number of applications that make use of SQL leads to a real need for helper techniques in its development in general and in testing in particular. Yet, although many testing techniques [55] and test adequacy criteria [59] 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) [40], which use a common set of major characteristics, such as the relational schema and core clauses for selecting, joining, combining, grouping and sorting data. On some occasions, however, developing even a single statement may be a complicated task [28]. Test cases are complicated to write because the input consists of information spread over several tables containing many rows, and the output is likewise a table structure. Queries are tightly dependent on the relational schema and small changes can entail undesirable side effects in many queries. Moreover, SQL is non-procedural and 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 [25].

Mutation testing techniques [56,36] help the tester to create test data and evaluate their adequacy by systematically inserting artificial faults in a given program and then evaluating the percentage of faults that are detected by a given test set. Empirical studies comparing the fault detection ability of 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 ability of a test suite [1]. The development of a set of mutants specifically tailored for dealing with the particularities of SQL constitutes the motivation and aim of this work.

In this article we develop a set of mutation operators for SQL queries that retrieve data from the database (SELECT queries) and test the mutants using a set of queries drawn from the NIST SQL

*Final preprint version published in Information and Software Technology, 49(4) 398-417, 2007.*
*© Elsevier, doi:10.1016/j.infsof.2006.06.009*
*Submitted: September 9th, 2005, Accepted: June 12th, 2006*

Conformance Test Suite. Further experiments aimed at reducing the cost of testing are performed using two different approaches: reducing the number of mutants (selective mutation) and reducing the number of test cases (by selecting the order in which mutants are killed). We shall show that in some cases selective mutation behaves slightly differently for the SQL mutants than those developed for imperative programs. However, the number of test cases can be reduced by ordering the mutants from the most difficult to the easiest to be killed.

The article is organised as follows: In Section 2 we provide an overview of related work on database testing, common errors in SQL and mutation testing. The set of mutants for SQL are described in Section 3 and subsequently tested using queries drawn from the NIST SQL Conformance Test Suite in Section 4. Some indications on the feasibility of reducing the cost of testing when using the mutation criterion are given in Section 5. Finally, in Section 6 we discuss all the above issues and present our conclusions in Section 7.

## 2. Background

This section provides some background on research into testing database applications (subsection 2.1), some common sources of errors that programmers commit when writing database queries (subsection 2.2) and a brief overview of mutation testing (subsection 2.3).

### *2.1. Related work on testing database applications*

Even though a great deal of research on databases and on software testing has been carried out in recent years, few studies have been specifically related to the testing of database applications. The studies focus on automatic test case generation (either considering SQL statements, database structure or both), checking test results, web applications, regression testing or developing test adequacy criteria.

Test case generation by means of considering the database schema and other properties or constraints is the approach taken in [12,57,42]. In all cases, neither the SQL statements that are executed nor adequacy criteria are considered. Considering only the SQL query, testing and adequacy measurement is carried out in [9] after translating the SQL query into a procedural language and then using conventional testing techniques. A quite different approach is that of [47], in which the test cases are valid SQL statements (instead of data) that are randomly generated with the goal of evaluating the differences between database management systems.

Many other studies on database test case generation consider both the structure of the data and the query under test. Relational algebra is used in [30,50] and general purpose constraint solvers for test data generation in [58]. The information on the database schema and the SQL queries is completed with heuristics supplied by the tester to automatically generate test cases and checking the test results in the AGENDA tool [11], which has been extended to deal with transactions [15]. The test cases are specified in [54] using pre and post-conditions in the form of intensional rules (using an SQL-like language) and then the database populated for fulfilling the rules.

Other different approaches that involve the testing of applications that use databases focus on the testing of web applications [16,17] and on regression testing [20,52].

Test adequacy criteria for database applications are a more recent field of study. In [48], a multiple condition based coverage criterion is defined for testing select queries taking into account both the database structure and SQL syntax and semantics. The criterion is used to develop and complete test cases that are able to reveal faults caused by errors in the use of joins, conditions or the handling of null values. Data-flow adequacy criteria are defined in [23], which extends the concept of the control flow graph by taking into account database interactions at different levels of granularity (databases, relations, tuples, attributes and values of attributes) starting from the initial test suite database state. An improved approach is presented in [53], which considers transactions (both committed and non-committed) and the define-use pairs for different database states resulting from the execution of previous statements, not only for the initial state. As far as we are aware, the only work dealing with mutants specifically tailored for dealing with the particularities of SQL is [10], that proposes a set of mutants based on features present in a conceptual model of the database schema.

### *2.2. Studies on errors in querying databases*

Since the appearance of the first languages for querying databases, many empirical studies have been conducted into the performance of humans in querying databases using different languages and different underlying data models. Most of the studies covered in Reisner's survey [41] compare different query languages and different skills in the users. These studies provide an initial insight into the problems that the user encounters when writing SQL queries, such as the use of computed variables, correlated variables, group by, composition (nested queries) and quantification.

More recent studies [7] draw attention to the problems with joins and confusion over the handling of where, group by and having clauses. The problems with joins and having are also revealed in [4]. A taxonomy of frequent SQL errors is given in [5,6], which shows many potential problems that are spread across all main SQL clauses, especially with missing and unnecessary joins, inconsistent conditions and with duplicate rows.

The effect of the normalization of database schema and the underlying data models has been extensively studied [3,4,8,27,46]. Although the use of a normalized logical data model facilitates data integrity, it makes the process of query writing more difficult. Other kinds of problems motivated by the semantic distance between the information request (ambiguities), the underlying data representation (incongruence) and the query syntax are analysed in [2]. Queries with greater construct incongruence resulted in more errors for six out of eight classes of SQL clauses. Performance differences were most evident for errors in the where and join conditions, from, and select clauses. Ambiguity is associated with errors in select, where conditions, group by and having clauses.

A survey of the industrial usage of SQL queries is presented in [28]. The overall results give the impression that most SQL features are used quite often in real applications. Among the most outstanding general problems encountered are: the confusing "nested maze" (due to a not well defined semantics for nesting), uncertainty of the query accuracy when there are multiple joins of many tables, and difficulties in detecting logical errors as compared to third generation languages. Problems in the formulation of queries include difficulties in joins, output formatting, the use of many aggregate functions in a single query, the use of incorrect field and name definitions, and variables used with wrong variable types, especially for embedded SQL. Another survey, [40], focuses on the kind of statements that appear in industrial applications. According to this survey, the select clause is the most widely used (constituting up to 68% of the total number of queries), and some of the most frequent features used are the order by, aggregate functions, comparison using equal operators, and the set and like operators.

### *2.3. Mutation testing overview*

Mutation testing is a fault-based testing technique that was originally proposed in [13,19]. 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. Mutants are created by transforming the source code using a set of defined rules (mutation operators) that are developed to induce simple syntax changes based on errors that programmers typically make or to force common testing goals. Each mutant is executed with the test data and when it produces an incorrect output (the output is different to that of the original program), the mutant is said to be killed. A test case is said to be effective if it kills some mutants that have not yet been killed by any of the previously executed test cases. Some mutants always produce the same output as the original program, so no test case can kill them. These mutations are said to be equivalent mutants. After executing a test set over a number of mutants, the mutation score is defined as the percentage of dead mutants divided by the number of non-equivalent mutants.

Mutation testing can be easily integrated in systems that automate mutant generation and execution, as for example, Mothra [24]. The generation of test cases for killing mutants can be performed manually or automatically using a constraint-based test case generator [14]. A great deal of research has been conducted into mutation testing for decades in order to improve the feasibility of the approach (see [36] for a survey). Among some of most recent contributions are tools for different kinds of languages such as object-oriented [29] and mutation systems for different kind of applications like web applications [31] or web services [37].

A mutation approach was used in [50] to perform a partial evaluation of the fault detection capability of database test cases and in [15,17] for seeding manual faults in queries in order to assess the effectiveness of test generation techniques.

## 3. SQL mutation operators

As explained in Section 2.2, usage of and problems in writing queries spread across all syntax and semantics elements of the SQL language. It would therefore seem reasonable to adopt a mutation-based approach covering a wide range of SQL features to assess the adequacy of database test cases.

In this section we describe the SQL mutation operators that have been designed. Operators are organized in the following *categories* identified by two capital letters:

- Mutations for the main SQL clauses (*SC*).
- Mutations for the operators that are present in conditions and expressions (*OR*).
- Mutations related to the handling of NULL values (*NL*).
- Replacement of identifiers: column references, constants and parameters (*IR*).

Each category defines several mutation operators or *mutant types* identified by three capital letters that are described in the subsequent subsections. As most of the operators can be applied in different SQL clauses, each type is further decomposed into *subtypes*, each of which refer to a particular mutant type when it is applied to a given clause (SELECT, JOIN, WHERE, GROUP BY, HAVING and ORDER BY). We conclude the section with a short description of the automation of the mutation process and the way in which views are handled.

### *3.1. SC – SQL clause mutation operators*

The aim of *SC* operators, described below, is to mutate the most distinctive features of SQL as compared to other languages (clauses, aggregate functions, subquery quantifiers, etc.). These operators contribute to detecting a number of faults such as incorrect joins, the wrong usage of the DISTINCT quantifier that can lead to the presence of unwanted duplicate rows or incorrect aggregate calculations, or incorrect orderings in the result set.

*SEL – SELECT Clause*.- Each occurrence of one of the SELECT or SELECT DISTINCT keywords is replaced by the other.

*JOI – JOIN Clause*.- Each occurrence of a join-type keyword (INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, CROSS JOIN) is replaced by each of the others. When a join-type is replaced by CROSS JOIN, the search-conditions under the ON keyword are removed. When CROSS JOIN is replaced by another join-type, an ON clause is added and its corresponding join-condition is created based on the primary keys of the joined tables.

*SUB – Subquery Predicates*.- Subqueries are normally used in *predicates* in the general form of $e\hat{A}p(Q)$, where *e* is the *row value constructor* (usually an attribute or expression), $\Re$ is a *relational operator* {=,<>,<,<=,>,>=}, *p* is a keyword representing the *predicate* and *Q* is the *subquery*. Three types of predicates can be formed depending on the kind of the keyword *p*:

- Type I ($p\in$ {ALL, ANY, SOME}): of the form $e\hat{A}p(Q)$.
- Type II ($p\in$ {IN, NOT IN}): of the form $e\ p(Q)$.
- Type III ($p\in$ {EXISTS, NOT EXISTS}) of the form $p(Q)$.

The mutations are the following: (1) Each occurrence of a keyword in a predicate of any type is replaced by each of the other keywords of the same type except for the replacement of ANY by SOME, as these have the same semantic meaning. (2) Additional replacements are made depending on the keyword type:

- Each type I keyword is (1) replaced by each of the type II keywords and then the relational operator is removed, (2) replaced by each of the type III keywords and then both the relational operator and the row value constructor are removed. Do not replace either =SOME by IN or <>ALL by NOT IN because they have the same meaning.
- Each type II keyword is (1) replaced by all combinations of each type I keywords and relational operators, and (2) replaced by each of the type III keywords and then the row value constructor is removed. Do not replace either IN by =SOME or NOT IN by <>ALL because they have the same meaning

- Each type III keyword is replaced by the other.

*GRU – Groupings*.- Each of the group-by-expressions is removed. If the removed expression in the GROUP BY is present in the select-list or in the order-by-list, this expression must be enclosed in an aggregate function to avoid a syntactically wrong query. In this case, two mutants are generated for each of the expressions, one using the MIN and the other using the MAX aggregate functions. If there is only one group-by-expression, then the whole clause is removed.

*AGR – Aggregate functions*.- Each occurrence of one of the aggregate functions (MAX, MIN, AVG, AVG(DISTINCT), SUM, SUM(DISTINCT), COUNT, COUNT(DISTINCT)) in a select-list or having-list is replaced by each of the others. Replacement must take into account the data type of the function argument. If the data type is character, then AVG and SUM are excluded from the replacement. Also, if the data type is character and the aggregate function belongs to a HAVING clause, then the COUNT function is not mutated if it participates in a comparison with a numeric expression.

*UNI – Query concatenation*.- (1) Each occurrence of one of the union keywords (UNION, UNION ALL) is replaced by the other keyword. (2) Each of the queries that participate in the union is removed.

*ORD – Ordering of the result set*.- For each occurrence of an order-by-expression (1) the direction of ordering is changed by replacing each of the keywords (ASC, DESC) by the other. If neither of these keywords is present, DESC is added. (2) Remove each of the order-by-expressions (if there is only one order-by-expression, then the entire clause is removed), and (3) exchange each pair of adjacent order-by-expressions.

### *3.2. OR – Operator replacement mutation operators*

The *OR* operators adapt and extend the expression modification operators described in [24]. The aim of these mutants is to detect logical errors in the WHERE and HAVING clauses.

*ROR – Relational operator replacement*.- Each occurrence of one of the relational operators {=,<>,<,<=,>,>=} is replaced (1) by each of the other operators, (2) by *falseop* (always returns false) and (3) by *trueop* (always returns true).

*LCR – Logical connector operator*.- Each occurrence of one of the logical operators (AND, OR) is replaced (1) by each of the other operators, (2) by *falseop*, (3) by *trueop*, (4) by *leftop* (returns the left operand), and (5) by *rightop* (returns the right operand).

*UOI – Unary Operator Insertion*.- Each arithmetic expression or reference to a number *e* is replaced by *-e*, *e+1* and *e-1*. References to numbers are not mutated either inside of GROUP BY and ORDER BY clauses or in the select-list of an EXISTS subquery.

*ABS – Absolute Value Insertion*.- Each arithmetic expression or reference to a number *e* is replaced by *ABS(e)* and *–ABS(e)*. The same exceptions as for *UOI* operators are applicable here.

*AOR – Arithmetic operator replacement*.- Each arithmetic operator {+,-,*,/,%} is (1) replaced by each of the others, (2) operators *leftop* and *rightop* are applied to the arithmetic expression.

*BTW – Between predicate*.- Each condition in the form *a BETWEEN x AND y* is replaced (1) by *a>x AND a<=y* and (2) by *a>=x AND a<y*. If the condition is NOT BETWEEN, the mutants are negated.

*LKE – Like predicate*.- The possible combinations of string conditions in the form *a* LIKE *s* are infinite, since *s* is a search pattern. Therefore the mutations will be restricted to exercising the behaviour of the wildcards {%,_} (the percent symbol means for any character string and the underscore means for an individual character). Each occurrence of a wildcard is mutated by (1) removing the wildcard, (2) replacing the wildcard by the other, (3) removing the character just before the wildcard if it is not at the beginning of *s* and (4) removing the character just after the wildcard if it is not at the end of *s*. (5) If no wildcard is present at the beginning of *s*, then add each of the wildcards at the beginning, and (6) if no wildcard is present at the end of *s* then add each of the wildcards at the end.

### *3.3. NL – NULL mutation operators*

In SQL, the domain explicitly defined for each attribute is extended to include a distinguished symbol (NULL) that denotes the absence of any data value and which may be interpreted as undefined, not relevant or unknown. Programmers and testers must be very careful to avoid undesirable effects resulting from the incorrect behaviour of conditions having null values [51,21].

Therefore, conditions are evaluated using a tri-valued logic and then logical expressions can return true, false and undefined. For instance, the evaluation of an AND logical expression of the form *a AND b*, will return unknown when either *a* or *b* are unknown; the evaluation of an OR expression of the form *a OR b* will return unknown if *a* is false and *b* is unknown, and true if *a* is true and *b* is unknown. When the search condition in a WHERE clause is evaluated to unknown, the row that would result if it were evaluated to true is excluded from the result set (it behaves similarly to false).

Incorrect treatment of NULL values can lead to unpredictable results such as failing to include rows in the result set that should be present, or returning NULL values in the result set that could cause incorrect behaviours when they are stored in variables and then processed by the program[1]. Hence, the mutations related to null values must be considered in order to detect test cases for considering this kind of situation.

*NLF – Null check predicates*.- Each occurrence of one of the predicates IS NULL or IS NOT NULL is replaced by the other.

*NLS – Null in select list*.- A good test case must result in each output variable having values that cover its domain as much as possible. In the case of an SQL query, we want the result set of the query to return at least one NULL value when possible. The *NLS* operator will transform each item in the select list (column names or expressions) by generating mutants that will be killed when this value is NULL, but not killed when it is not NULL. This operator replaces each column reference *c* in the select-list by a function *ifnull(c,r)*[2], that substitutes a value *c* by *r* (*r* is a replacement value outside of the domain of *c*) when a null value is encountered in *c*. The column reference is not mutated if all the attributes involved in it are declared NOT NULL by the database schema.

*NLI/NLO – Nulls in the input data*.- Operators *ROR* and *LCR* seek to produce a change in the outcome of a condition for some values in the input domain. However, since SQL uses tri-valued logic, a condition can have three possible outcomes: true, false and undefined, and this issue must be taken into account.

Let us consider the first four rows and first two columns of Table 1. Each column represents an original condition *cond(A)* over some attribute *A* when the outcome of its evaluation is true and false respectively. The first row represents this condition and the next three rows represent the mutation operators that achieve the different combinations of outcomes for each different evaluation of the condition. In the first two cases, if the attribute is null, then the outcome of the condition is undefined (last column). The set of mutation operators to be designed for exercising null values in the inputs must be such that for every different combination of outcomes of the condition under non null values, whenever a null value is present, the outcome is changed (therefore, the mutant can be killed). The last four rows achieve this goal and will be the basis for the *NLI* and *NLO* mutants that are described below:

*NLI – Include nulls*.- This operator forces a true value of the condition when there is a null value. For each attribute *a* in a condition *C* of the form *aÂ b* or *bÂ a*, the condition is replaced by *C OR a IS NULL*. If *a* is present in more than one condition, then every occurrence of *a* is replaced simultaneously.

*NLO – Other nulls*.- This operator completes the other combinations presented in Table 1. For each attribute *a* in *C*, the condition is replaced by (1) *NOT C OR a IS NULL*, (2) *a IS NULL*, and (3) *a IS NOT NULL*.

### *3.4. IR – Identifier replacement mutation operators*

*IR* operators are an adaptation of the Replacement-of-operand operators described in [24], taking into account the fact that arrays do not exist in SQL, though column references do. These mutants replace the column identifiers, constants and references to query parameters and so, they are able to

[1] The value obtained when an application program retrieves a null column is highly dependent on both the programming language and the platform. For instance, in the J2SE platform using a `ResultSet`, a null integer is retrieved either as zero (when stored in an `int` variable), or as `null` (when stored in an `Object`). In the .NET platform, a `DataSet` produces a run-time exception when retrieving a null value.

[2] The syntax is different depending on the DBMS vendor. In SQL92, for instance, this is a special case of the COALESCE function, in MySQL it is implemented by *IFNULL(c,r)*, in Oracle by *NVL(c,r)* and in SQL Server by *ISNULL(c,r)*.

detect mistakes such as the use of incorrect fields. The replacement of column references in queries having subqueries, group by or unions is restricted only to those replacing columns that are within the scope of the clause containing the column to be replaced.

*IRC – Column replacement*.- Each column reference is replaced by each of the other column references, constants and parameters that are present in the query and are type compatible.

*IRT – Constant replacement*.- Each constant is replaced by each of the other constants, columns and parameters that are present in the query and are type compatible.

*IRP – Parameter replacement*.- Each query parameter reference is replaced by each of the other parameters, columns and constants that are present in the query and are type compatible.

*IRH – Hidden column replacement*.- The aim of this operator is to detect potential errors produced when many similar columns appear in the same table and test cases have not enough diversity in their values to detect the use of a wrong column name. Each column attribute reference is replaced by each of the other columns that are defined in its table provided that they have not been the replacement in any of the other *IR* operators and are type compatible.

### *3.5. Execution of the mutants*

The generation and execution of the mutants has been completely automated in an SQL mutation tool[3]. The query to mutate along with relevant information about the database schema, the loading of the test databases, commands for changing the data and query parameter setup are written into an XML script that is further processed for mutant generation and running the test cases. The query is parsed into an XML internal representation and then each of its elements is processed: For each one, both the query and the database schema are explored and mutants are generated by applying the aforementioned rules. Finally, the mutants are executed.

Each execution of all mutants of a query is enclosed in a single database transaction which finishes with a rollback to ensure a clean execution for all queries. Inside the transaction, the test data specified in the script is loaded into the test tables and then some SQL queries can be optionally executed to modify it. Subsequently, query parameters are instantiated with their actual values for both the original query and each mutant. Each of these is executed and finally their outputs are compared to determine the mutants that have been killed. When the execution of a mutant causes a run-time error, it is considered as if it were killed. The process is repeated for each test case by executing only the mutants that have not yet been killed.

Many queries use database views to encapsulate complex queries. In the case of a query using views, each view is also mutated. In this case the execution of mutants consists in first executing each mutated version of the query with the original views and then executing the original query with each mutant of each view. If there is more than one view, all views are kept as the original except the one that is being mutated. Views are mutated in the same way as queries, except that views do not have any parameter and each column in the select-list of the view is not mutated in any way if this column is not referenced in the query using that view (the converse would generate equivalent mutants).

In the following, the unit of execution of the test cases will be denoted as *T-Query*, which, in the case of a query without views, is the same as the query. In the case of a query using views, it refers to the query together with its views. Therefore, in this latter case the mutants of a T-Query will be all the mutants of both the main query and each view used by it.

## 4. Testing

In this section we describe the results of the execution of a set of SQL queries against the mutants generated as defined in the previous section. We first describe the SQL suite to be used and then provide the results of the execution of the mutants.

### *4.1. Description of the SQL test suite*

The SQL Conformance Test Suite was originally developed by the Information Technology Laboratory of the National Institute of Standards and Technology (NIST) and is used to validate

[3] A Web interface to generate the mutants of SQL queries is available at http://in2test.lsi.uniovi.es/sqlmutation

commercial SQL products for conformance with ISO, ANSI, and FIPS SQL standards [49]. The test suite is organised into programs (which we shall call *modules*), each one having several tests. Each test exercises one or more queries. The software for the SQL Test Suite can be downloaded from the NIST Conformance Test Suite Software Web pages [33].

For the purpose of mutation testing, we selected a set of tests that exercises the way in which SQL retrieves data from the database: *Data Manipulation Language* (modules whose name begins with *dml*). The content of each module is inserted in the XML script and tagged for automating mutant generation and test execution. Original queries are designed to be executed without any parameters and always make use of constants in the conditions. To be able to exercise a query under different parameters, we transform these constants into parameters.

The test suite contains tests and procedures to evaluate conformance to various levels of the standards or profiles. We have organized these in two groups of tests:

- Entry SQL: This is the most basic feature set. All major commercial database vendors conform to this level of the standard.
- Transitional SQL and Intermediate SQL: These represent more advanced levels of conformance. Major database vendors have many features of these levels, although they do not usually attain full conformance.

The two groups differ mainly in the number of different tables that are used and their more or less intensive use of views and joins. Table 2 displays the main characteristics of each group and the SQL features being tested are enumerated in Appendix I. We shall henceforth refer to the set of queries being used as the *SQL suite*.

### *4.2. Executing the entry SQL suite*

We shall execute the SQL suite against all mutants. Since the test data, as specified in the NIST suite, will achieve a lower than 100% mutation score, then we shall complete the test data in several steps, the results of which are summarized in Table 3 grouped by mutant category and mutant type.

#### *4.2.1. Step 1: Executing the NIST test cases*

The first step will consist in taking the entry level SQL suite and executing it against all the mutants. The test cases are exactly as defined by the original NIST test suite. From column 1 in Table 3, we can see that this test set has achieved 69.6% mutation coverage ranging from 51.4% for *NL* mutants to 81.0% for *IR* mutants.

#### *4.2.2. Steps 2 to 4: Completing test cases automatically*

To complete the test cases in order to increase mutation coverage and avoid the expensive task of manually designing test cases, we perform steps 2 to 4 as explained below:

- In step 2 we keep the same database as that of step 1, but the queries will be executed using different values for the parameters. For each parameter, we construct a vector of possible values consisting of every value of the attribute represented by the parameter as extracted from the test tables. Additional values are added in order to exercise different conditions, such as ensure that there is at least one value surrounded by two values immediately before and after (for checking boundary values) high, negative, zero and null values. For each query having parameters, a test case is created for each combination of parameters. After executing the new cases generated, the percent mutation score increases up to 79.7% (column 2 in Table 3).
- In step 3 we take a copy of the original database and apply changes to it so as to obtain duplicate rows, high and negative values in the attributes as well as incomplete relations to other tables (master without details and details without master). Query parameters are selected as in step 2. The percentage score now increases up to 83.3% (column 3 in Table 3).
- Finally, we can see that the category with the lowest score is *NL*. Thus, in step 4 we take another copy of the original database and modify it by including null values when possible and query parameters selected in the same way as in step 2. Now the mutation coverage increases up to 85.6% (column 4 in Table 3), with an increase of 25.4% for the *NL* category.

#### *4.2.3. Step 5: Completing test cases manually and detecting equivalent mutants*

After obtaining a reasonably diverse set of test cases consisting of several database instances and sets of parameters, we proceed in this step to manually complete the test cases in order to kill the

remaining mutants. During this process, some mutants will be killed by the new tests and others will be determined as being equivalent. Before starting this manual process, we determine whether some of the equivalent mutants can be determined automatically and implement these criteria in the SQL mutation tool.

- *SEL* mutants are those that obtain the lowest score after step 4. Many of the remaining mutants may be automatically determined as equivalent when some of the following rules are fulfilled: (1) If the SELECT is inside a subquery predicate (if the SELECT is a scalar subquery, then SELECT DISTINCT is equivalent to SELECT, but not conversely). (2) If the SELECT participates in a UNION (without ALL), because the UNION will remove duplicated rows. (3) If all primary keys of all tables being joined in the SELECT are included in the select-list and there is not a GROUP BY clause. Keys that are not in the select list are considered as if they were if they participate in one or a series of AND's join conditions in the form $k_1=k_2$, where $k_1$ is in the select-list and $k_2$ is not. (4) If every column in the select-list is composed of aggregate functions, as these will always return only one row. (5) When there is a GROUP BY clause and all the group-by columns are in the select-list.
- *NLS* mutants are equivalent when the attribute or expression *c* to be mutated participates in a condition composed by AND's of single conditions in the form $c=x$, because if *c* has a null value, then the result of the condition will be undefined and hence the row will never be selected at the output.
- *IR* mutants: Replacing a column reference $c_1$ by another $c_2$ generates an equivalent mutant if the replacement is made in a SELECT in which all tables are joined using INNER JOIN and there is a single or a series of AND's join conditions in the form $c_1=c_2$.

The percentage of coverage achieved after manually completing the test cases (94.2%) is given in Table 3, column 5. The remaining percentage (up to 100%) corresponds to equivalent mutants. The number and percentages of equivalent mutants (both automatically and manually detected) are given in the subsequent columns. It should be noted that there are not many equivalent mutants (5.8%) and half of these correspond to automatically detected equivalent mutants.

The time spent manually creating the new test cases was 24 hours (to kill all the non-equivalent mutants remaining alive after step 4 and detect the manual equivalent mutants). The machine time needed to generate the mutants, select and execute all test cases and store the information in the database is 16 minutes for all steps on a single dedicated computer (single Pentium 4 3GHz processor). The total number of test cases selected was 4,662, of which 778 were effective in killing the mutants.

### *4.3. Executing the Transitional and Intermediate SQL suite*

Transitional and intermediate suite includes another set of different queries, using different tables and involving more views and join clauses than in the entry level, as can be seen in Table 2. Table 4 shows the results obtained after reproducing exactly the same procedure as described before for this suite. The percentages of scores are similar to the entry SQL (Table 3). The manual time for creating the new test cases was 15 hours and the machine time 22.5 minutes. The total number of test cases selected was 1,579, of which 575 were effective in killing the mutants.

The last row in Table 4 presents the grand total over all queries (entry, transitional and intermediate). It should be noted that the percentage of equivalent mutants (6.0%) is similar, though slightly lower than that obtained in other experiments for mutation in imperative programs (e.g. [35], which achieves 6.75%), although in the case of SQL mutants, many of these (2.5%) are automatically detected as being equivalent.

## 5. Reducing the cost

Having elaborated the test cases for achieving a 100% mutation score, we now wish to check whether the cost of testing could be reduced in some way. Two different approaches will be explored in the following subsections. The first (subsection 5.1) consists in reducing the number of mutants being tested and the second (subsection 5.2) in selecting an adequate ordering for mutants when test cases are developed in order to reduce the number of test cases.

### *5.1. Reducing the number of mutants by selective mutation*

This approach was first suggested in [32] and further developed in [35]. The basic idea of selective mutation consists in selecting a reduced number of mutant operators such as those mutants being truly different from the others. If operators that generate the largest number of mutants can be removed, then the reduction of the cost of running mutants will also be large.

The procedure consists in developing a set of effective test cases for killing all mutants excluding some operators. The test cases are then run over the whole set of mutants. If the score obtained after the run is very close to 100%, this implies that the operators that were excluded when generating the test cases may be removed from the set of mutants because they are not useful in detecting new faults.

#### *5.1.1. Selective mutation by mutant category*

First at all, we need to have a large enough set of test cases. A random test case development would not be appropriate here, since it is likely to ignore test cases for killing the most complicated situations. We therefore decided to use the test databases developed previously in Section 4 and to generate different random sequences of the test cases as explained here: For each T-Query and each test case to be generated, we randomly select each of the groups of test cases from steps 2 to 5 (test cases from step 1 are not used because they are included in step 2). Once a group has been selected, we randomly select one of the test cases included in it. We thus balance the use of different test databases along with the use of different parameter instantiations when calling the queries. The pool consists of 6,241 test cases for all queries. All the values that will be presented below correspond to the means of 10 random series of test cases.

The first two columns in Table 5 display for each mutant category (row) the mutation score obtained by selecting test cases for killing all mutants with the exception of said category (column 1) and the score after executing the selected cases over all mutants (column 2). Comparing the results with those presented in [35], we obtain a similar percentage when removing *IR* (99.67%) as that obtained when excluding the replacement-of-operand operators (99.54%). Lower percentages are obtained in all the other cases that can be compared: removing *OR* (94.50%) compared with the removal of the expression modification operators (97.31%) and removing *SC* (99.23%) compared with the removal of the statement modification operators (99.97%). The *SC* and *NL* operators are very specific to SQL features and it does not seem appropriate to remove them in order to avoid the risk of forgetting important features to be tested. The only operators that could be considered for their exclusion are the *IR* operators, which also generate many mutants.

However, this reasoning cannot be generalised without looking at the individual queries. Table 5, column 3 displays the number of T-Queries that do not achieve a 100% score under selective mutation. Columns 4 and 5 display the same information as columns 1 and 2, though considering only the T-Queries that have not achieved a 100% score under selective mutation. Similarly, columns 6 and 7 present the same information, though considering only the T-Query that has achieved the lowest score.

When excluding the *IR* operators, we check that unkilled *IR* mutants are concentrated in 10.4% of T-Queries (25 out of the total of 241). The score for these queries is 97.30% and the worst case lowers this percentage to 91.43%. Compared with [35], in which the program with the lowest score when removing the replacement-of-operators operands achieves a 98.45% score, in the case of the SQL mutants developed in the present study and for some queries we do not conclude that the *IR* operators should be eliminated. Another argument in favour of not removing these operators is that the use of a wrong column name is an easily made error when writing queries, especially if there are many columns with similar names and/or meanings.

#### *5.1.2. Selective mutation by mutant type*

If we assume that we do not remove an entire mutant category, a further experiment is carried out performing the selective mutation by excluding each mutant type instead of all types in a category. Table 6 displays the same information as in the previous table, though in this case each row displays the scores when removing only the mutants that belong to a type (a single mutation operator). Following a similar reasoning to that put forward in [35], we could draw a line at 99% and try to remove those mutant types that achieve more than 99% when they are excluded form the mutant set. We check the scores over all mutants when we consider the queries that do not achieve a 100% score

(column 5) and we can consider *AOR*, *LCR*, *NLO* and *UNI*. If we consider only the worst case (column 7), we can only consider *AOR* and *NLO*. However, arguments could be found for not removing any of them: The *AOR* and *LCR* are the operators corresponding to the 'sufficient mutant operators' for imperative programs; *LCR* does not generate many mutants and therefore the saving would be minimal and *AOR* has not generated many mutants, as the SQL set does not have many arithmetic operations. The *UNI* operator mutates a very important SQL clause, and allows duplicate rows in unions to be detected. *NLO*, which tries to complete the *ROR* and *LCR* to include nulls in conditions is the only clear candidate to be removed (compare its score with *NLI*, which aims for the same goal).

Since the goal of selective mutation is to significantly reduce the number of mutants without loss of effectiveness and as we must take a conservative approach, the elimination of some of the above operators would not give a significant improvement and hence we decided not to remove any of them.

Nonetheless, the information given in Table 6 provides an indication that some mutants are more easily killed than others. Therefore, the most difficult mutants could be used first when developing test cases, in the hope that they will also kill other 'easier to kill' mutants. This issue will be explored in the next subsection.

### *5.2. Reducing the size of the test set by ordering the mutants*

Another approach to reducing the cost is to somehow reduce the number of test cases that must be developed. Test suite prioritization techniques seek to order test cases previously created according to some criterion in order to reduce the number of test cases needed to attain a certain goal such as achieving a given coverage criterion as fast as possible or the rate of fault detection [43,45,18]. Test suite reduction techniques seek to select a subset of the test suite so that its coverage is the same as the original test suite [44,38]. These approaches have the common goal of reducing the cost of regression testing.

Fault-based testing researchers have established a fault class hierarchy that orders some kinds of mutations [26] that can be used to skip a test case from an 'easier to detect' class in the hierarchy, provided that we detect a corresponding fault from a 'harder to detect' class [39]. The question is whether this would be applicable to the SQL mutants. In our case, we wish to determine an ordering of the mutants such that if we design test cases to kill the mutants in said order, we achieve a reduction in the number of test cases while attaining the same fault detection effectiveness (100% mutation score). This would be useful not only for regression but also for the development of new test cases.

#### *5.2.1. The effect of ordering on the total number of test cases*

After running the test set as explained in the previous section, the number of effective test cases needed to kill all mutants over all queries was 1,353 (unordered sequence). An initial indication of the influence of ordering on the number of test cases that are needed is that if we run the test cases using this unordered sequence in inverse order (from step 5 down to 2), the number of test cases is 1,160 (14.3% lower than the former). We shall now determine the number of test cases needed to kill all mutants under different sequences obtained by ordering the mutants in different ways.

The procedure will consist in first selecting a set of groups of mutants ordered according to some criterion. For each group of mutants in the established order, we select the set of effective test cases to kill all mutants within this group and then run the test cases over the whole set of mutants.

Three main types of orderings are considered:

- *Category ordering*: Each group is formed by all mutants in the same category. The orderings are denoted as *Cxyzt*, where *x*, *y*, *z* and *t* represent the first character of the corresponding category. For instance, *Cinos* first uses the whole set of test cases in the pool, keeping only the test cases that are effective for killing all mutants that belong to the *IR* category and executes these test cases over all mutants, then the procedure is repeated for *NL*, then for *OR* and finally for *SC*.
- *Global Ordering*: According to the previous results on fault class hierarchies, we first order some groups of mutants according to their mortality. Given a group of mutants, mortality is the percentage of test cases of the whole test pool that kill some mutants in that group and gives an indication about whether the mutant is easier to detect (high mortality) or harder to detect (low mortality). Two different groups of mutants are considered (one for each type and another for each subtype) and two orderings (ascending and descending). These orderings are denoted by *GXy*, where *X* refers to the kind of grouping (*T*: by type, *S*: by subtype) and *y* refers to the ordering in

mortality (*a*: ascending, *d*: descending). For instance, *GTa* includes in each group all mutants that belong to the same type and orders them by mortality in ascending order.

- *Local Ordering*: The ordering is determined as in global ordering, with only one difference: Mortality is calculated by considering only the mutants and test cases in the pool related to the query being tested. Therefore, we have a different local ordering for each T-Query, instead of a unique global ordering for all T-Queries. The orderings are denoted in the same way as before, but beginning with *L*.

Figure 1 depicts the box plots of the total number of test cases for each of the above orderings. Each box includes the values of ten series of randomly selected test cases from the pool of test cases developed as explained in the previous subsection. The first box corresponds to test cases selected in a random sequence (labelled as *RA*).

The first issue worth noting is that we need a higher number of test cases when using random order (*RA*) than when using any ordering, and the number of test cases is always lower using any order than when using both the unordered and the random sequences. Let us conduct an ANOVA analysis to confirm the above indications using the total number of test cases as the dependent variable and a single factor with a level for each ordering. The subjects will be each of the ten series of randomly selected test cases. Using alpha=0.05, we first check the prerequisites: Levene's test for checking the homogeneity of variances gives p=0.743, and Shapiro-Wilk's test for checking normality gives p>0.05 for all orderings with the exception of *Csoin* (p=0.048), *Csoni* (p=0.036), *LSa* (p=0.008) and *LTa* (p=0.013). As ANOVA is robust enough with respect to deviations from normality, we can conduct the test, which gives F=92.471, p<0.001. As expected, this result allows us to reject the null hypothesis that the means are equal for the different orderings.

A post-hoc multiple comparison procedure will allow us to identify homogeneous subsets. We use Tukey's Honestly Significant Difference test (HSD) for multiple comparisons, the results of which are given in Table 7. For each ordering, an *x* means the range in which it is classified, range 1 being that which achieves the lowest number of cases.

Comparing the different orderings by category (*Cxxxx*), we find that all of them are not significantly different, with the exception of those in which the first category of mutants to be killed is *OR* (orderings *Coxxx*). This gives a first rule for selecting the order of mutants: do not select the *OR* mutants as the first ones to be killed. Comparing the global and local orderings, we find that using ascending orderings always produces a lower number of test cases than using descending orderings, in keeping with the hypothesis that the most difficult mutants must be the first to be killed to reduce the number of test cases. Local ascending orderings achieve the lowest number of test cases, although there is not much difference between local and global (the difference is, however, significant). Additionally, it is slightly better (although not completely significant) to use the orderings by subtype instead of by type.

If we wish to establish one ordering as a criterion for testing all the queries, we would select *GSa* (*LSa* ordering is distinct for each query and must thus be discarded as a general criterion). Table 8 summarizes the results, showing the mean values of some of the orderings compared to the unordered and random sequences. The table shows that by using *GSa* ordering we reduce the number of test cases needed 14.7% when compared to a random sequence. The improvement is much higher (a 30.4% reduction) if compared with the unordered sequence.

*5.2.2. Validity of the ordering*

The previous analysis has indicated that *GSa* is the most adequate ordering to use in order to obtain a lower number of test cases. However, this ordering was elaborated using the results (the mortality of the mutants) of the same queries that are to be tested using it. An important question here is whether this ordering is sufficiently valid for testing queries that are different from those that have participated in the elaboration of the ordering.

To check this, we arrange all the T-Queries in quartiles. Each quartile is determined on the basis of the mutant size (the number of mutants in the T-Query). Therefore, the first quartile will group queries having the lowest number of mutants and the last one will group queries having the highest number of mutants. We use the orderings *RA* and *GSa* as before, plus a new ordering (*QGSa*) obtained as follows: For all T-Queries that belong to a quartile $Q_i$, the ordering *QGSa* is determined using the same criteria as for the *GSa* ordering, though considering only the T-Queries that belong to all the

other quartiles $Q_j$, $j?i$. The ordering *QGSa* used to test each query was determined in this way using different queries with different mutant sizes.

We conduct a univariate ANOVA with two factors: quartile (0 to 3) and ordering (*RA*, *GSa* and *QGSa*) and the number of test cases as the dependent variable. Although samples are normal across all cells (the Shapiro-Wilk test of normality gives p>0.25), Levene's test of the homogeneity of variances is just barely not satisfied (p=0.031). The reason is that samples under the ordering *RA* give higher variances than the others. Even so, we proceed with the analysis, which gives significant differences in both factors (p<0.001).

Figure 2 depicts the marginal means for each quartile and ordering. Queries in higher quartiles need more test cases, as would be expected, since they are more complex. In each quartile, the ordering RA needs more test cases than the others. Orderings *GSa* and *QGSa* are apparently very close to each other. A further post-hoc Tukey's HSD test confirms what was indicated by the figure: the differences in the number of test cases is significantly different across all quartiles (they appear classified in four different groups), *RA* ordering is significantly different to all the others, but *GSa* and *QGSa* are not significantly different to each other (they appear in the same group).

All the above analyses considered the whole set of queries as if they were a single large program. The analyses were performed using ten random sequences of test cases as the experimental subjects. However, the effects of the ordering in each individual query should also be considered. We therefore consider each T-Query as an independent subject and the average of test cases needed across all the ten series of cases as the dependent variable. A proper analysis for this would be a repeated measures ANOVA, which requires assuring that the variance-covariance matrices of the dependent variable are circular. This is checked using Mauchly's test of sphericity, which gives p<0.001. As sphericity is violated so severely that it cannot be adjusted (e<0.6), we switch to a non-parametric test. A test that does not require all these assumptions to be satisfied is Friedman's test, which checks whether a set of variables that are measured on the same subject are equal by ranking the variables and stating the null hypothesis that the rankings of the variables are equal.

Friedman's test gives p<0.001 when comparing all the orderings and when comparing *RA* against *GSa* and *QGSa* in turn. However, when comparing *GSa* against *QGSa*, the test gives p=0.067. Therefore, we cannot reject the null hypothesis that both orderings are equal.

The above result provides a useful indication to the tester when using SQL mutants as criteria for developing test cases: The ordering *GSa* obtained after analyzing the mortality of the mutants over a set of queries can be efficiently used in the testing of a different set of queries. This provides the user with a fairly stable criterion about the order in which mutants must be killed in order to obtain a lower number of test cases than if no ordering were used.

## 6. Discussion

This paper has focused on the development of a set of mutants for queries that retrieve data from the database (SELECT statements), which are the most complex in terms of the variety of SQL features being exercised and the most widely used, both for selecting the data to be processed in transactions and for reporting. As indicated in [40] "once you have a reasonable understanding of SELECT, the other statements are fairly straightforward". The mutation of the other SQL main statements that modify the database state could thus be easily adapted. The UPDATE statement is composed of several assignments of values to columns along with a WHERE clause to select the rows that will be updated. Then most of the operators used for SELECT statements are applicable. Similarly the INSERT statement also performs assignments of values to columns and optionally, uses a SELECT clause as the source of the values to be assigned, and the DELETE statement also uses a WHERE clause to select the rows to be deleted. The most significant difference would be the way a mutant is determined to be killed: in this case comparison must be made by comparing the final state of the database after executing the query instead of comparing the result set, that would result in an additional overhead when running the mutants.

A first issue about the way in which mutants are killed by the test cases is suggested by the results presented in Section 4 (Tables 3 and 4). The columns presenting the mutation scores after the first step present a relatively high score obtained using (in almost all cases) only one test case for each T-Query. An initial explanation might be that this first test case is the same as the one used in the NIST SQL test

suite and is hence far from being random. It was elaborated to exercise a particular SQL feature, so we expect a relatively high effectiveness. Perhaps a more general explanation is that when testing SQL queries it is usual to have a large input test space (many rows for each table), as in the case of the test cases used here. Therefore, a single test case is able to exercise many different situations in the data and then detect many of the faults represented by the mutants.

Although many mutants are killed by a single test case, all mutants seem important, and there are some mutants in some queries that are difficult to kill, as shown in subsection 5.1. We can take advantage of this fact by using a specific ordering of mutants to reduce the number of test cases (subsection 5.2). However, there are certain potential threats to the validity of this conclusion. The first concern is with regard to the SQL suite that has been used and its representativeness compared to real-life SQL queries. We may lay claims to its representativeness in terms of the set of SQL features covered by it, since it was designed precisely to test SQL conformance. Nevertheless it includes many simple queries (complex queries are concentrated in the last quartile), and it is unsure whether each query is representative in terms of the combination of features that can be found in a real-life query.

A second major concern is related to the way in which the test cases have been constructed. The procedure for selecting the initial test cases does not vary much from the usual procedure: use a small set of database loads to test the most common features (as used in steps 1 to 4) and then complete with specific test cases to cover the rest (step 5). However, for the experiments related to selective mutation and the ordering of mutants (Section 5), the procedure for constructing a pool of test cases is conditioned by the initial test cases and therefore could introduce bias in the conclusions. A huge effort would be required to develop more test cases manually (which would not necessarily reduce the bias), and selecting random database loads would run the risk of omitting test cases for the most difficult mutants.

A third relevant issue is whether the set of SQL mutants are representative of real-life faults. If so, the effectiveness of a set of test cases for killing mutants is similar to the effectiveness in detecting real-life faults. Ostensibly, if we assume that mutants for SQL behave like mutants for imperative code, we could borrow conclusions from previous studies on mutation testing (e.g. [1]). The mutants used in this study are 1-order mutants (each mutant is generated by applying only one mutation operator). Previous studies on the coupling effect in mutation testing for imperative programs have shown that test data developed to kill 1-order mutants are very successful at killing 2-order mutants [34]. However, it should not be forgotten that testing SQL queries is somewhat different to testing imperative programs because of the high input space of test cases and also because a single query can be considered as a small program that performs many complex operations. An example of the differences compared to mutants in imperative programs is the behaviour for some queries of the *IR* mutants under selective mutation, as shown in subsection 5.1.

Nonetheless, the set of queries was found to be useful in validating the tool that automates the creation and execution of mutants, showing that selective mutation may not be generally applicable for some queries. The conclusions about the reduction of test cases by means of selecting an adequate order of mutants also agree with previous studies on fault-based testing.

## 7. Conclusions

We have developed a quite complete set of mutant operators for SQL queries that retrieve information from a database that exercises the main syntax and semantic features of SQL. Mutants perform small changes in condition operators (*OR*), the replacement of variables, constants and parameters (*IR*) and take into account several very specifically SQL-related features that exercise the way in which the query selects, joins, combines, groups and orders the selected data (*SC*). Another specific category of mutants deals with the processing of unknown information (null values) and the evaluation of conditions using tri-valued logic (*NL*). The SQL mutation adequacy criterion is used as both a guide to complete database unit test cases as well as a measure of the completeness of a test set.

The generation and execution of mutants is fully automated and these were tested against a reasonably large set of SQL statements drawn from the NIST SQL Conformance Test suite. We found slight differences compared with the mutants developed for imperative programs, as in the case of the study on selective mutation. However, the approach of using the most difficult mutants to develop the first test cases of a suite with the aim of reducing the overall number of test cases that are needed is

validated with the test set that we have used. This may be useful not only for regression testing, but also for significantly reducing the effort involved in developing the test cases and especially in reducing the effort spent on the construction of the test oracles.

Despite the growing number of applications whose core information is stored in a database, there is a lack of test adequacy criteria and test case design techniques specifically tailored for database programs. The mutation approach for SQL queries may be used as a complementary aid to the tester for developing database test cases or as a foundation for test automation tools. Similar to the use of mutation in imperative programs, the use of the SQL mutants could also become a valuable tool for systematically injecting faults in queries and then using these faults to evaluate the effectiveness of test cases and for comparing different assessment and test case generation techniques for database applications.

## Acknowledgments

This study was funded by the Department of Education and Science (Spain) under the National Program for Research, Development and Innovation, Projects IN2TEST (TIN2004-06689-C03-02) and REPRIS (TIN2005-24792-E).

## References

[1] J. Andrews, L. Briand, Y. Labiche, Is Mutation an Appropriate Tool for Testing Experiments?, Proceedings of the 27th International Conference on Software Engineering. ACM Press, New York, NY, USA, 2005, pp. 402-411.

[2] A.F. Borthick, P.L. Bowen, D.R. Jones, M.H.K. Tse, The effects of information request ambiguity and construct incongruence on query development, Decision Support Systems 32(1) (2001) 3–25.

[3] A.F. Borthick, P.L. Bowen, S.T. Liew, F.H. Rohde, The effects of normalization on end-user query errors. An experimental evaluation, International Journal of Accounting Information Systems 2(4) (2001) 195–221.

[4] P.L. Bowen, F.H. Rohde, Further evidence of the effects of normalization on end-user query errors: an experimental evaluation, International Journal of Accounting Information Systems 3(4) (2002) 255–290.

[5] S. Brass, C. Goldberg, Proving the Safety of SQL Queries, Proceedings of the fifth International Conference on Quality Software, IEEE Computer Society Press, Los Alamitos, California, 2005, pp. 197-205.

[6] S. Brass, C. Goldberg, Semantic Errors in SQL Queries: A Quite Complete List. Journal of Systems and Software 79(5) (2006) 630-644.

[7] H.C. Chan, K.K. Wei, User-Database Interface: The Effect of Abstraction Levels on Query Performance, MIS Quarterly 17(4) (1993) 441-464.

[8] H.C. Chan, L. Xiang, Evaluation of the Impacts of Data Model and Query Language on Query Performance, Proceedings of the Second Annual Workshop on HCI Research in MIS, Seattle, WA, 2003, pp. 12-13.

[9] M.Y. Chan, S.C. Cheung, Testing database applications with SQL semantics, Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications, Springer, Singapore, 1999, pp. 363–374.

[10] W.K. Chan, S.C. Cheung, T.H. Tse, Fault-Based Testing of Database Application Programs with Conceptual Data Model, Proceedings of the fifth International Conference on Quality Software, IEEE Computer Society Press, Los Alamitos, California, 2005, pp. 187-196.

[11] D. Chays, Y. Deng, P.G. Frankl, S. Dan, F.I. Vokolos, E.J. Weyuker, An AGENDA for testing relational database applications, Software Testing, Verification and Reliability 14(1) (2004) 17-44.

[12] R.A. Davies, R.J.A. Beynon, B.F. Jones, Automating the testing of databases. Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, IEEE Computer Society, 2000, pp. 15-20.

[13] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer 11(4) (1978) 34-43.
[14] R.A. DeMillo, A.J. Offutt, Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering 17(9) (1991) 900-910.
[15] Y. Deng, P. Frankl, D. Chays, Testing Database Transactions with AGENDA. Proceedings of the 27th International Conference on Software Engineering, ACM Press, New York, NY, USA, 2005, pp. 78-87.
[16] Y. Deng, P. Frankl, J. Wang, Testing web database applications, Workshop on Testing, Analysis and Verification of Web Services, ACM Press, New York, NY, USA, 2004, pp. 1-10.
[17] S. Elbaum, G. Rothermel, S. Karre, M. Fisher II, Leveraging User-Session Data to Support Web Application Testing, IEEE Transactions on Software Engineering 31(3) (2005) 187-202.
[18] S. Elbaum, A.G. Malishevssky, G. Rothermel, Test Case Priorization: A Family of Empirical Studies, IEEE Transactions on Software Engineering 28(2) (2002) 159-182.
[19] R.G. Hamlet, Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering 3(4) (1977) 270-290.
[20] R.A. Haraty, N. Mansour, B.A. Daou, Regression Testing of Database Applications, Journal of Database Management 13(2) 2002 31-42.
[21] T. Imielinski, W. Lipski Jr, Incomplete Information in Relational Databases. Journal of the Association for Computing Machinery 31(4) (1984) 761-79.
[22] International Standards Organisation, Information technology – Database languages – SQL, ISO/IEC 9075:1992, third edition.
[23] G.M. Kapfhammer, M.L. Soffa, A family of test adequacy criteria for database-driven applications. Proceedings of the 9th European software engineering conference and the 11th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, New York, NY, USA, 2003, pp. 98- 107.
[24] K.N. King, A.J. Offutt. A Fortran Language System for Mutation-Based Software Testing. Software Practice and Experience 21(7) (1991) 686-718.
[25] H.J. Klein, How to Modify SQL Queries in Order to Guarantee Sure Answers, ACM SIGMOD Record 23(3) (1994) 14-20.
[26] D.R. Kuhn, Fault Classes and Error Detection Capability of Specification-Based Testing, ACM Transactions on Software Engineering and Methodology, 8(4) (1999) 411–424.
[27] C. Liao, P.C. Palvia, The impact of data models and task complexity on end-user performance: an experimental investigation, International Journal of Human-Computer Studies 52 (2000) 831-845.
[28] H. Lu, H.C. Chan, K.K. Wei, A Survey on Usage of SQL, ACM SIGMOD Record 22(4) (1993) 60-65.
[29] Y.S. Ma, J. Offutt, Y.R. Kwon, MuJava: an automated class mutation system, Software Testing, Verification and Reliability 15(2) (2005) 97-133.
[30] H. Mannila, K.J. Räihä, Test Data for Relational Queries. Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems, ACM Press, New York, NY, USA, 1986, pp. 217-223.
[31] N. Mansour, M. Houri, Testing web applications, Information and Software Technology, 48(1) (2005) 31-42.
[32] A.P. Mathur, Performance, effectiveness, and reliability issues in software testing. Proceedings of the 15th Annual International Computer Software and Applications Conference, 1991, pp. 604-605.
[33] National Institute of Standards and Technology, Software Diagnostics and Conformance Testing Division, Conformance Test Suite Software (http://www.itl.nist.gov/div897/ctg/software.htm), accessed July 2005.
[34] A.J. Offutt, Investigations on the Software Testing Coupling Effect. ACM Transactions on Software Engineering and Methodology 1(1) (1992) 5-20.
[35] A.J. Offutt, A. Lee, T. Rottermel, R.H. Untch, C. Zapf, An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology 5(2) (1996) 99-118.

[36] A.J. Offutt, R.H. Untch RH. Mutation 2000: Uniting the Orthogonal. Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, 2000, pp. 45-55.
[37] A.J. Offut, W. Xu, Generating Test Cases for Web Services Using Data Perturbation, Workshop on Testing, Analysis and Verification of Web Services, ACM Press, New York, NY, USA, 2004, pp. 1-10.
[38] A.J. Offutt, J. Pan, J.M. Voas, Procedures for Reducing the Size of Coverage based Test Sets, Proceedings of the 12th International Conference on Testing Computer Software, Washington DC, 1995, pp. 111-123.
[39] V. Okun, P.E. Black, Y. Yesha, Comparison of fault classes in specification-based testing, Information and Software Technology 46(8) (2004) 525–533.
[40] R. Pönighaus, 'Favourite' SQL-Statements – An Empirical Analysis of SQL-Usage in Commercial Applications. Proceedings of the sixth International Conference on Information Systems and Management of Data (Lecture Notes in Computer Science, vol. 1006), Springer, 1995, pp. 75-91.
[41] P. Reisner, Use of Psychological Experimentation as an Aid to Development of a Query Language, IEEE Transactions on Software Engineering 3(3) (1977) 218-229.
[42] M.A. Robbert, F.J. Maryanski, Automated test plan generator for database application systems, Proceedings of the ACM SIGSMALL/PC symposium on Small systems, ACM Press New York, NY, USA 1991, pp. 100-106.
[43] G. Rothermel, S. Elbaum, Putting Your Best Tests Forward, IEEE Software 20(5) (2003) 74-77.
[44] G. Rothermel1, M.J. Mary Jean Harrold, J. von Ronne, C. Hong, Empirical studies of test-suite reduction, Software Testing, Verification and Reliability 12(4) (2002) 219-249.
[45] G. Rothermel, R.H. Untch, M.J. Harrold, Prioritizing Test Cases For Regression Testing, IEEE Transactions on Software Engineering 27(10) 2001 929-948.
[46] K. L. Siau, H.C. Chan, K.L. Wei, Effects of Query Complexity and Learning on Novice User Query Performance With Conceptual and Logical Database Interfaces, IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans, 34(2) (2004) 276-281.
[47] D. Slutz, Massive Stochastic Testing of SQL, Proceedings of the 24th International Conference on Very Large Data Bases, Morgan Kaufmann, San Francisco, CA, USA. 1998, pp. 618-622.
[48] M.J. Suárez-Cabal, J. Tuya, Using an SQL Coverage Measurement for Testing Database Applications, Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, New York, NY, USA, 2004, pp. 253-262.
[49] J. Sullivan, SQL Test Suite Goes Online, Computer 30(6) (1997) 103-105.
[50] W.T. Tsai, D. Volovik, T.F. Keefe, Automated test case generation for programs specified by relational algebra queries, IEEE Transactions on Software Engineering 16(3) (1990) 316-324.
[51] Y. Vassiliou, Null values in database management. A denotational semantics approach, Proceedings of the ACM SIGMOD international conference on Management of data, ACM Press, New York, NY, USA, 1979, pp. 162-169.
[52] D. Willmor, S.M. Embury, A Safe Regression Test Selection Technique for Database Driven Applications, Proceedings of the IEEE International Conference on Software Maintenance, September 2005.
[53] D. Willmor, S.M. Embury, Exploring test adequacy for database systems, 3rd UK Software Testing Research workshop, September 2005.
[54] D. Willmor, S.M. Embury, An Intensional Approach to the Specification of Test Cases for Database Applications. Proceedings of the 28th International Conference on Software Engineering, ACM Press, New York, NY, USA, 2006.
[55] M.R. Woodward, Insights into software testing. Software Focus 2(3) (2001) 93-103.
[56] M.R. Woodward, Mutation testing - its origin and evolution. Information and Software Technology 35(3) (1993) 163-169.
[57] X. Wu, Y. Wang, Y. Zheng, Privacy preserving database application testing. Proceedings of the ACM Workshop on Privacy in Electronic Society, ACM Press, New York, NY, USA, 2003, pp. 118-128.
[58] J. Zhang, C. Xu, S.C. Cheung, Automatic generation of database instances for white-box testing. Proceedings of the 25th Annual International Computer Software and Applications Conference, IEEE Computer Society Press, Los Alamitos, CA, 2001, pp. 161–165.

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

## Appendix I – Features of the SQL set

The following table displays the features being tested in the SQL suite for each module and test as specified in the NIST SQL Conformance Test Suite [33]. Column entitled NQ indicates the number of queries in the test (the number of T-Queries and between brackets the number of views, if any).

| Module | Test | NQ | Features being tested |
|---|---|---|---|
| dml001 | 0001 | 1 | SELECT with ORDER BY DESC! |
| dml001 | 0002 | 1 | SELECT with ORDER BY integer ASC! |
| dml001 | 0003 | 1 | SELECT with ORDER BY DESC integer, named column! |
| dml001 | 0004 | 1 | SELECT with UNION, ORDER BY integer DESC! |
| dml001 | 0005 | 1 | SELECT with UNION ALL! |
| dml001 | 0158 | 1 | SELECT with UNION and NOT EXISTS subquery! |
| dml001 | 0160 | 1 | SELECT with parenthesized UNION, UNION ALL! |
| dml004 | 0008 | 1 | SQLCODE 100:SELECT on empty table ! |
| dml004 | 0009 | 2 | SELECT NULL value! |
| dml008 | 0016 | 1 | SELECT ALL syntax! |
| dml008 | 0017 | 1 | SELECT:checks DISTINCT! |
| dml008 | 0018 | 1 | SQLCODE = 100, SELECT with no data! |
| dml008 | 0019 | 1 | SQLCODE = 0, SELECT with data! |
| dml008 | 0020 | 1 | SELECT NULL value ! |
| dml008 | 0164 | 1 | SELECT:default is ALL, not DISTINCT! |
| dml013 | 0039 | 1 | COUNT DISTINCT function! |
| dml013 | 0040 | 1 | SUM function with WHERE clause! |
| dml013 | 0041 | 1 | MAX function in subquery! |
| dml013 | 0042 | 1 | MIN function in subquery! |
| dml013 | 0043 | 1 | AVG function! |
| dml013 | 0044 | 1 | AVG function - empty result NULL value! |
| dml013 | 0167 | 1 | SUM ALL function! |
| dml013 | 0168 | 1 | SUM function! |
| dml013 | 0169 | 1 | COUNT(*) function ! |
| dml013 | 0170 | 1 | SUM DISTINCT function with WHERE clause! |
| dml013 | 0171 | 1 | SUM(column) + value! |
| dml014 | 0045 | 2 | BETWEEN predicate! |
| dml014 | 0046 | 2 | NOT BETWEEN predicate ! |
| dml014 | 0047 | 2 | IN predicate! |
| dml014 | 0048 | 2 | NOT IN predicate! |
| dml014 | 0049 | 2 | IN predicate value list! |
| dml014 | 0050 | 1 | LIKE predicate -- %! |
| dml014 | 0051 | 1 | LIKE predicate -- underscore! |
| dml014 | 0052 | 1 | LIKE predicate -- ESCAPE character! |
| dml014 | 0053 | 2 | NOT LIKE predicate |
| dml014 | 0054 | 1 | IS NULL predicate! |
| dml014 | 0055 | 2 | NOT NULL predicate! |
| dml014 | 0056 | 1 | NOT EXISTS predicate! |
| dml014 | 0057 | 1 | ALL quantifier ! |
| dml014 | 0058 | 1 | SOME quantifier! |
| dml014 | 0059 | 1 | ANY quantifier ! |
| dml018 | 0069 | 1 | HAVING COUNT with WHERE, GROUP BY! |
| dml018 | 0070 | 1 | HAVING COUNT with GROUP BY! |
| dml018 | 0071 | 1 | HAVING MIN, MAX with GROUP BY 3 columns! |
| dml018 | 0072 | 1 | Nested HAVING IN with no outer reference! |
| dml018 | 0073 | 1 | HAVING MIN with no GROUP BY! |
| dml019 | 0074 | 1 | GROUP BY col with SELECT col., SUM! |
| dml019 | 0075 | 1 | GROUP BY clause! |
| dml019 | 0076 | 1 | GROUP BY 2 columns! |
| dml019 | 0077 | 1 | GROUP BY all columns with SELECT * ! |
| dml019 | 0078 | 1 | GROUP BY three columns, SELECT two! |
| dml019 | 0079 | 1 | GROUP BY NULL value! |
| dml020 | 0080 | 1 | Simple two-table join! |
| dml020 | 0081 | 1 | Simple two-table join with filter! |

| Module | Test | NQ | Features being tested |
|---|---|---|---|
| dml020 | 0082 | 1 | Join 3 tables! |
| dml020 | 0083 | 1 | Join a table with itself! |
| dml022 | 0096 | 1 | Subquery with MAX in < comparison predicate! |
| dml022 | 0097 | 1 | Subquery with AVG - 1 in <= comparison predicate! |
| dml022 | 0098 | 1 | IN predicate with simple subquery! |
| dml022 | 0099 | 1 | Nested IN predicate - 2 levels! |
| dml022 | 0101 | 1 | Quantified predicate <= ALL with AVG in GROUP BY! |
| dml022 | 0102 | 1 | Nested NOT EXISTS with correlated subquery and DISTINCT! |
| dml023 | 0103 | 1 | Subquery with comparison predicate! |
| dml023 | 0105 | 2 | Subquery in comparison predicate is empty! |
| dml023 | 0106 | 1 | Comparison predicate <> ! |
| dml023 | 0107 | 2 | Comp predicate with short string logically blank padded! |
| dml023 | 0180 | 1 | NULLs sort together in ORDER BY! |
| dml023 | 0181 | 1 | NULLs are equal for DISTINCT! |
| dml024 | 0108 | 1 | Search condition true OR NOT(true)! |
| dml024 | 0109 | 1 | Search condition true AND NOT(true)! |
| dml024 | 0110 | 1 | Search condition unknown OR NOT(unknown)! |
| dml024 | 0111 | 1 | Search condition unknown AND NOT(unknown)! |
| dml024 | 0112 | 1 | Search condition unknown AND true! |
| dml024 | 0113 | 1 | Search condition unknown OR true! |
| dml025 | 0114 | 1 | Set functions without GROUP BY returns 1 row! |
| dml025 | 0115 | 1 | GROUP BY col, set function: 0 groups returns empty table! |
| dml025 | 0116 | 1 | GROUP BY set functions: zero groups returns empty table! |
| dml025 | 0117 | 1 | GROUP BY column, set functions with several groups! |
| dml026 | 0118 | 1 | Monadic arithmetic operator +! |
| dml026 | 0119 | 1 | Monadic arithmetic operator -! |
| dml026 | 0120 | 3 | Value expression with NULL primary IS NULL! |
| dml026 | 0121 | 1 | Dyadic operators +, -, *, /! |
| dml026 | 0123 | 1 | Evaluation order of expression! |
| dml038 | 0205 | 1 | Cartesian product is produced without WHERE clause! |
| dml040 | 0209 | 1 | Join 2 tables from different schemas! |
| dml051 | 0227 | 2 | BETWEEN predicate with character string values! |
| dml051 | 0228 | 2 | NOT BETWEEN predicate with character string values |
| dml052 | 0229 | 2 | Case-sensitive LIKE predicate! |
| dml059 | 0257 | 1 | SELECT MAX, MIN (COL1 + or - COL2)! |
| dml059 | 0258 | 1 | SELECT SUM(2*COL1*COL2) in HAVING SUM(COL2*COL3) |
| dml059 | 0259 | 1 | SOME, ANY in HAVING clause! |
| dml059 | 0260 | 1 | EXISTS in HAVING clause! |
| dml059 | 0264 | 2 | WHERE, HAVING without GROUP BY! |
| dml060 | 0261 | 1 | WHERE (2 * (c1 - c2)) BETWEEN! |
| dml060 | 0262 | 1 | WHERE clause with computation, ANY/ ALL subqueries! |
| dml060 | 0263 | 1 | Computed column in ORDER BY! |
| dml061 | 0269 | 3 | BETWEEN value expressions in wrong order! |
| dml061 | 0270 | 1 | BETWEEN approximate and exact numeric values! |
| dml061 | 0271 | 1 | COUNT(*) with Cartesian product subset ! |
| dml061 | 0273 | 1 | SUM, MAX, MIN = NULL for empty arguments ! |
| dml061 | 0278 | 1 | IN value list with USER, literal, variable spec.! |
| dml069 | 0406 | 1 | Subquery from different schema! |
| dml070 | 0409 | 2 | Effective outer join -- with 2 cursors! |
| dml070 | 0411 | 1 | Effective set difference! |
| dml070 | 0412 | 1 | Effective set intersection! |
| dml073 | 0393 | 1 | SUM, MAX on Cartesian product! |
| dml073 | 0394 | 1 | AVG, MIN on joined table with WHERE without GROUP! |
| dml073 | 0395 | 1 | SUM, MIN on joined table with GROUP without WHERE |
| dml073 | 0396 | 1 | SUM, MIN on joined table with WHERE, GROUP BY, HAVING! |
| dml073 | 0417 | 1 | Cartesian product GROUP BY 2 columns with NULLs! |
| dml073 | 0418 | 1 | AVG, SUM, COUNT on Cartesian product with NULL! |
| dml073 | 0419 | 1(1) | SUM, MAX, MIN on joined table view! |
| dml075 | 0431 | 2 | Redundant rows in IN subquery |
| dml075 | 0432 | 6 | Unknown comparison predicate in ALL, SOME, ANY! |
| dml075 | 0433 | 6 | Empty subquery in ALL, SOME, ANY! |
| dml075 | 0434 | 1 | GROUP BY with HAVING EXISTS-correlated set function! |
| dml075 | 0442 | 3 | DISTINCT with GROUP BY, HAVING! |
| dml079 | 0452 | 2 | Order of precedence, left-to-right in UNION [ALL]! |
| dml079 | 0453 | 6 | NULL with empty subquery of ALL, SOME, ANY! |
| dml090 | 0512 | 2 | (value expression) for IN predicate! |

| Module | Test | NQ | Features being tested |
|---|---|---|---|
| dml090 | 0513 | 1 | NUMERIC(4) implies CHECK BETWEEN -9999 AND 9999! |
| dml090 | 0523 | 3 | (value expression) for BETWEEN predicate! |
| dml090 | 0564 | 1 | Outer ref. directly contained in HAVING clause! |
| dml104 | 0591 | 2(2) | NATURAL JOIN (feature 4) (static)! |
| dml104 | 0592 | 4(4) | INNER JOIN (feature 4) (static)! |
| dml104 | 0593 | 4(4) | LEFT OUTER JOIN (feature 4) (static)! |
| dml104 | 0594 | 2(2) | RIGHT OUTER JOIN (feature 4) (static)! |
| dml106 | 0599 | 9(9) | UNION in views (feature 8) (static)! |
| dml112 | 0623 | 11(15) | OUTER JOINs with NULLs and empty tables! |
| dml114 | 0635 | 6(5) | Feature 13, grouped operations (static)! |
| dml114 | 0637 | 7(8) | Feature 14, Qualified * in select list (static)! |
| dml134 | 0689 | 1(3) | Many Trans SQL features #1:  inventory system! |
| dml135 | 0692 | 4(6) | Many TSQL features #3:  enhanced proj/works! |
| dml147 | 0841 | 2 | Multiple-join and default order of joins ! |
| dml147 | 0842 | 7 | Multi-column joins ! |
| dml148 | 0843 | 3 | Ordering of column names in joins ! |
| dml148 | 0844 | 7 | Outer join predicates ! |
| dml158 | 0857 | 3 | join condition set function, outer reference! |
| dml162 | 0863 | 3(2) | joined table directly contained in cursor,view! |
| dml165 | 0870 | 1 | Non-identical descriptors in UNION! |
| dml171 | 0882 | 1 | More full outer join! |

Table 1: Mutations for conditions using tri-valued logic

| | cond(A) is true | cond(A) is false | cond(A) is undefined |
|---|---|---|---|
| cond(A) | true | false | undefined |
| NOT cond(A) | false | true | undefined |
| FALSEOP | false | false | false |
| TRUEOP | true | true | true |
| cond(A) OR A IS NULL | true | false | true |
| NOT cond(A) or A IS NULL | false | true | true |
| A IS NULL | false | false | true |
| A IS NOT NULL | true | true | false |

Table 2: Characteristics of the SQL suite

| | Entry SQL suite | Transitional & Intermediate SQL suite |
|---|---|---|
| Number of Modules | 26 | 12 |
| Number of Queries | 165 | 137 |
| Number of T-Queries | 164 | 77 |
| Number of different views | 1 | 26 |
| Number of different Tables | 8 | 20 |
| Number of mutants generated | 6,885 | 5,545 |

Table 3: Mutation scores by category and Mutant type (entry level, modules dml001 to dml090)

| Mutant Category | Mutant Type | Number of Mutants | Mutation score at each step | | | | | Equivalent (Automatic) | | Equivalent (Manual) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | Tot. | % | Tot. | % |
| SC | AGR | 560 | 72.9 | 78.9 | 80.7 | 80.7 | 94.1 | | | 33 | 5.9 |
| | GRU | 72 | 88.9 | 88.9 | 91.7 | 91.7 | 91.7 | | | 6 | 8.3 |
| | JOI | 84 | 61.9 | 61.9 | 70.2 | 70.2 | 71.4 | | | 24 | 28.6 |
| | ORD | 39 | 82.1 | 87.2 | 89.7 | 94.9 | 97.4 | | | 1 | 2.6 |
| | SEL | 241 | 5.0 | 6.2 | 12.4 | 12.4 | 18.3 | 188 | 78.0 | 12 | 5.0 |
| | SUB | 379 | 84.7 | 94.5 | 97.4 | 97.4 | 97.9 | | | 8 | 2.1 |
| | UNI | 23 | 87.0 | 87.0 | 91.3 | 91.3 | 91.3 | | | 2 | 8.7 |
| Total SC | | 1,398 | 65.0 | 70.5 | 73.8 | 74.0 | 80.6 | 188 | 13.4 | 86 | 6.2 |
| OR | ABS | 510 | 44.7 | 48.8 | 74.3 | 74.5 | 95.9 | | | 21 | 4.1 |
| | AOR | 253 | 90.5 | 90.5 | 90.5 | 90.9 | 100.0 | | | | |
| | BTW | 76 | 55.3 | 56.6 | 60.5 | 60.5 | 94.7 | | | 4 | 5.3 |
| | LCR | 145 | 82.1 | 93.8 | 95.2 | 95.2 | 98.6 | | | 2 | 1.4 |
| | LKE | 33 | 57.6 | 57.6 | 57.6 | 57.6 | 87.9 | | | 4 | 12.1 |
| | ROR | 1,211 | 69.9 | 93.2 | 94.8 | 95.1 | 98.5 | | | 18 | 1.5 |
| | UOI | 741 | 69.2 | 76.2 | 76.9 | 77.1 | 97.8 | | | 16 | 2.2 |
| Total OR | | 2,969 | 67.2 | 79.8 | 85.2 | 85.4 | 97.8 | | | 65 | 2.2 |
| NL | NLF | 8 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | | | 0 | 0.0 |
| | NLI | 92 | 9.8 | 12.0 | 12.0 | 84.8 | 92.4 | | | 7 | 7.6 |
| | NLO | 276 | 88.4 | 92.8 | 93.8 | 95.3 | 99.6 | | | 1 | 0.4 |
| | NLS | 153 | 7.2 | 13.1 | 13.1 | 54.2 | 81.7 | 11 | 7.2 | 17 | 11.1 |
| Total NL | | 529 | 51.4 | 55.8 | 56.3 | 81.7 | 93.2 | 11 | 2.1 | 25 | 4.7 |
| IR | IRC | 989 | 81.3 | 93.1 | 94.5 | 95.2 | 97.7 | 9 | 0.9 | 14 | 1.4 |
| | IRP | 562 | 67.0 | 87.5 | 91.0 | 91.0 | 99.5 | | | 2 | 0.4 |
| | IRT | 200 | 87.8 | 88.7 | 89.5 | 91.2 | 98.7 | | | 1 | 0.5 |
| | IRH | 238 | 82.7 | 94.7 | 97.3 | 98.0 | 99.6 | | | 3 | 1.3 |
| Total IR | | 1,989 | 81.0 | 92.5 | 94.4 | 95.1 | 98.5 | 9 | 0.5 | 20 | 1.0 |
| TOTAL | | 6,885 | 69.6 | 79.7 | 83.3 | 85.6 | 94.2 | 208 | 3.0 | 196 | 2.8 |

Table 4: Mutation scores by category and Mutant type (transitional and intermediate levels, modules dml104 to dml171)

| Mutant Category | Mutant Type | Number of Mutants | Mutation score at each step | | | | | Equivalent (Automatic) | | Equivalent (Manual) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | Tot. | % | Tot. | % |
| SC | AGR | 79 | 84.8 | 87.3 | 87.3 | 87.3 | 97.5 | | | 2 | 2.5 |
| | GRU | 41 | 75.6 | 87.8 | 90.2 | 90.2 | 90.2 | | | 4 | 9.8 |
| | JOI | 267 | 61.0 | 66.7 | 75.3 | 78.3 | 79.0 | | | 56 | 21.0 |
| | ORD | 66 | 86.4 | 89.4 | 89.4 | 89.4 | 97.0 | | | 2 | 3.0 |
| | SEL | 163 | 6.7 | 9.8 | 28.8 | 28.8 | 36.2 | 91 | 55.8 | 13 | 8.0 |
| | SUB | 29 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | | | | |
| | UNI | 50 | 62.0 | 64.0 | 72.0 | 74.0 | 82.0 | | | 9 | 18.0 |
| Total SC | | 695 | 56.0 | 60.3 | 68.8 | 70.1 | 74.5 | 91 | 13.1 | 86 | 12.4 |
| OR | ABS | 510 | 42.7 | 45.9 | 62.9 | 64.5 | 91.4 | | | 44 | 8.6 |
| | AOR | 61 | 88.5 | 98.4 | 100.0 | 100.0 | 100.0 | | | | |
| | BTW | 4 | 50.0 | 50.0 | 50.0 | 50.0 | 100.0 | | | | |
| | LCR | 312 | 62.8 | 82.4 | 89.4 | 97.4 | 99.0 | | | 3 | 1.0 |
| | LKE | 20 | 55.0 | 60.0 | 60.0 | 60.0 | 100.0 | | | | |
| | ROR | 1176 | 74.1 | 87.2 | 92.4 | 94.7 | 99.8 | | | 2 | 0.2 |
| | UOI | 735 | 78.8 | 84.1 | 85.3 | 88.7 | 97.1 | | | 21 | 2.9 |
| Total OR | | 2,818 | 68.6 | 78.4 | 84.8 | 87.8 | 97.5 | | | 70 | 2.5 |
| NL | NLF | 26 | 88.5 | 92.3 | 96.2 | 100.0 | 100.0 | | | | |
| | NLI | 75 | 6.7 | 9.3 | 9.3 | 48.0 | 64.0 | | | 27 | 36.0 |
| | NLO | 225 | 72.0 | 87.1 | 92.0 | 96.9 | 99.6 | | | 1 | 0.4 |
| | NLS | 180 | 15.6 | 15.6 | 18.3 | 63.9 | 70.6 | 8 | 4.4 | 45 | 25.0 |
| Total NL | | 506 | 43.1 | 50.4 | 53.8 | 78.1 | 84.0 | 8 | 1.6 | 73 | 14.4 |
| IR | IRC | 946 | 87.6 | 92.0 | 92.7 | 96.5 | 98.5 | 4 | 0.4 | 10 | 1.1 |
| | IRP | 315 | 92.8 | 94.2 | 94.2 | 100.0 | 100.0 | | | | |
| | IRT | 69 | 79.6 | 81.6 | 82.7 | 82.7 | 98.0 | | | | |
| | IRH | 196 | 91.1 | 92.1 | 93.7 | 94.6 | 100.0 | | | 4 | 2.0 |
| Total IR | | 1,526 | 87.5 | 90.8 | 91.7 | 94.5 | 98.8 | 4 | 0.3 | 14 | 0.9 |
| TOTAL | | 5,545 | 69.9 | 77.0 | 81.8 | 86.5 | 93.8 | 103 | 1.9 | 243 | 4.4 |
| GRAND TOTAL | | 12,430 | 69.7 | 78.5 | 82.7 | 86.0 | 94.0 | 311 | 2.5 | 439 | 3.5 |

Table 5: Selective mutation scores by mutant category

| Excluded category | Percent score considering all T-Queries | | Percent score considering only the T-Queries that do not achieve 100% | | | Percent score considering the worst case | |
|---|---|---|---|---|---|---|---|
| | Excluding a category | Over all mutants | Number of T-Queries | Excluding a category | Over all mutants | Excluding a category | Over all mutants |
| IR | 98.67% | 99.61% | 25 | 92.27% | 97.30% | 84.35% | 91.43% |
| NL | 78.49% | 98.31% | 157 | 74.15% | 97.57% | 0.00% | 0.00% |
| OR | 89.82% | 95.08% | 191 | 88.93% | 94.40% | 38.33% | 69.17% |
| SC | 94.50% | 99.23% | 96 | 88.14% | 98.18% | 25.00% | 70.00% |

Table 6: Selective mutation scores by mutant type

| Category | Excluded type | Percent score considering all T-Queries | | Percent score considering only the T-Queries that do not achieve 100% | | | Percent score considering the worst case | |
|---|---|---|---|---|---|---|---|---|
| | | Excluding a type | Over all mutants | Number of T-Queries | Excluding a type | Over all mutants | Excluding a type | Over all mutants |
| IR | IRC | 99.11% | 99.86% | 159 | 89.37% | 97.57% | 86.96% | 95.17% |
| | IRH | 99.44% | 99.96% | 28 | 82.50% | 98.53% | 40.00% | 91.43% |
| | IRP | 98.13% | 99.96% | 28 | 82.14% | 97.60% | 75.00% | 95.45% |
| | IRT | 96.07% | 99.86% | 77 | 78.18% | 96.11% | 69.09% | 92.34% |
| NL | NLF | 96.47% | 99.99% | 2 | 40.00% | 95.38% | 0.00% | 0.00% |
| | NLI | 70.00% | 99.66% | 91 | 56.15% | 98.71% | 0.00% | 94.67% |
| | NLO | 99.94% | 100.00% | 6 | 95.00% | 99.51% | 95.00% | 99.51% |
| | NLS | 47.74% | 98.87% | 218 | 39.59% | 97.53% | 0.00% | 85.00% |
| OR | ABS | 79.10% | 98.29% | 847 | 76.43% | 97.46% | 40.00% | 86.67% |
| | AOR | 100.00% | 100.00% | 0 | 100.00% | 100.00% | 100.00% | 100.00% |
| | BTW | 95.39% | 99.97% | 20 | 82.50% | 98.61% | 75.00% | 95.91% |
| | LCR | 99.47% | 99.98% | 43 | 94.42% | 99.29% | 88.00% | 98.85% |
| | LKE | 89.18% | 99.95% | 20 | 73.50% | 92.43% | 68.00% | 90.59% |
| | ROR | 95.93% | 99.17% | 1485 | 93.51% | 98.19% | 78.57% | 87.50% |
| SC | AGR | 97.19% | 99.85% | 219 | 92.24% | 98.64% | 80.00% | 95.38% |
| | GRU | 98.06% | 99.98% | 14 | 85.71% | 98.61% | 75.00% | 98.10% |
| | JOI | 96.16% | 99.91% | 64 | 83.75% | 98.31% | 55.00% | 95.88% |
| | ORD | 91.76% | 99.93% | 30 | 72.00% | 98.15% | 20.00% | 84.00% |
| | SEL | 73.30% | 99.76% | 50 | 45.00% | 98.78% | 0.00% | 76.67% |
| | SUB | 95.33% | 99.84% | 143 | 86.92% | 97.31% | 60.00% | 88.89% |
| | UNI | 98.06% | 99.99% | 8 | 85.00% | 99.36% | 80.00% | 98.79% |
| | UOI | 96.82% | 99.61% | 407 | 88.77% | 98.35% | 66.67% | 95.77% |

Table 7: Homogeneous subsets for the different orderings of the mutants (Tukey's HSD Test)

| Ordering | Tukey's Rank | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| RA | | | | | | | | | | | | x |
| Cinos | | | | x | x | x | | | | | | |
| Cinso | | | | x | x | x | | | | | | |
| Cions | | | | x | x | x | | | | | | |
| Ciosn | | | | x | x | x | x | | | | | |
| Cisno | | | | x | x | x | | | | | | |
| Cison | | | | | x | x | x | x | | | | |
| Cnios | | | | x | | | | | | | | |
| Cniso | | | | x | x | | | | | | | |
| Cnois | | | | x | x | x | | | | | | |
| Cnosi | | | | x | x | x | | | | | | |
| Cnsio | | | | x | x | | | | | | | |
| Cnsoi | | | | x | x | x | | | | | | |
| Coins | | | | | | | | | | x | x | |
| Coisn | | | | | | | | | | | x | |
| Conis | | | | | | | | | x | x | x | |
| Consi | | | | | | | | | x | x | x | |
| Cosin | | | | | | | | | | x | x | |
| Cosni | | | | | | | | | x | x | x | |
| Csino | | | | x | x | x | | | | | | |
| Csion | | | | x | x | x | | | | | | |
| Csnio | | | | x | x | x | | | | | | |
| Csnoi | | | | x | x | x | | | | | | |
| Csoin | | | | | | x | x | x | x | x | | |
| Csoni | | | | | | x | x | x | x | | | |
| GSa | | x | x | | | | | | | | | |
| GSd | | | | x | x | x | | | | | | |
| GTa | | | x | | | | | | | | | |
| GTd | | | | x | x | x | | | | | | |
| LSa | x | | | | | | | | | | | |
| LSd | | | | | | | | x | x | x | x | |
| LTa | x | x | | | | | | | | | | |
| LTd | | | | | | | x | x | x | x | x | |

Table 8: Comparison of the mean number of test cases generated using different mutant orderings

| Sequence/ordering of mutants | Number of test cases | Percent of test cases in relation to the unordered sequence | Percent of test cases in relation to the random sequence |
|---|---|---|---|
| Unordered sequence | 1,353.0 | | |
| Unordered reverse sequence | 1,160.0 | 85.7% | |
| Random sequence (*RA*) | 1,103.9 | 81.6% | |
| Global by type ascending (*GTa*) | 961.2 | 71.0% | 87.0% |
| Global by subtype descending (*GSa*) | 941.3 | 69.6% | 85.3% |
| Local by type descending (*LTa*) | 929.6 | 68.8% | 84.2% |
| Local by subtype descending (*LSa*) | 917.3 | 67.8% | 83.1% |

Figure 1: Boxplot for the total number of test cases generated under different mutant orderings

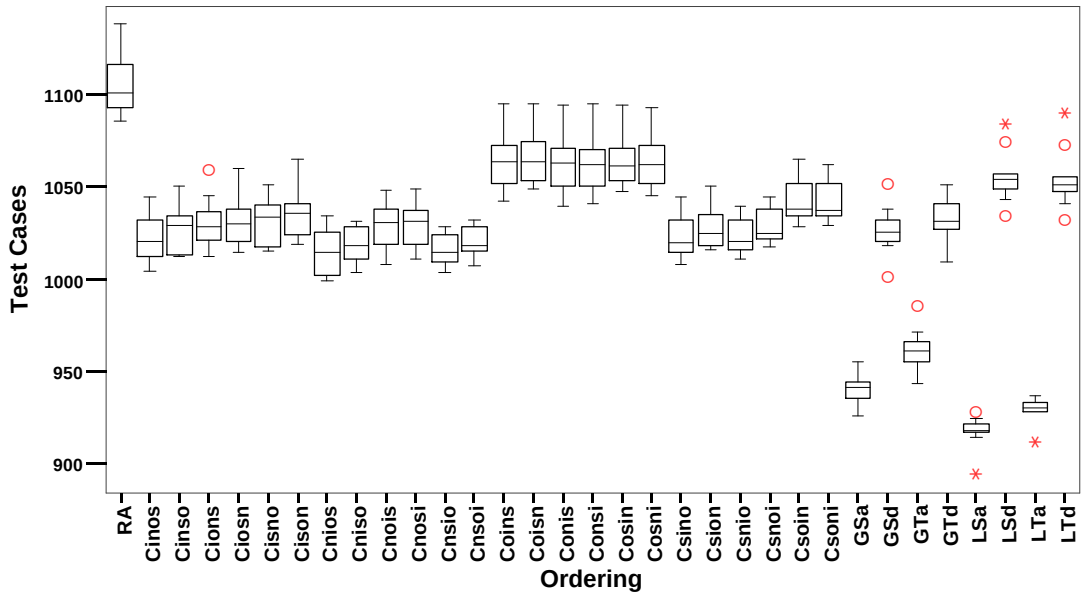


Figure 2: Marginal means of the number of test cases under different orderings and quartiles