SQLMutation: A tool to generate mutants of SQL database queries

Javier Tuya, Mª José Suárez-Cabal, Claudio de la Riva
University of Oviedo (SPAIN)
{tuya | cabal | claudio} @ uniovi.es

Abstract

We present a tool to automatically generate mutants of SQL database queries. The SQLMutation tool is available on the Web and it can be accessed using two different interfaces: A Web application to interactively generate the mutants and a Web service that allows it to be integrated with other applications developed using different platforms.

Keywords: Database testing, SQL testing, SQL query, Mutation operators

1. Introduction

The Structured Query Language (SQL) [6] is a semi-declarative language used by the applications to access the information stored in relational database systems.

The most frequent SQL queries used in applications are those that retrieve information from one or more tables of the database [8]. The `select` clause determines which fields (columns) constitute the query output, the `from` clause determines which tables are used and the join determines the criterion for joining rows from different tables. Then the `where` clause filters the rows based on some other criteria. The `group by` clause indicates how to combine the selected rows and the `having` clause performs a final filter based on other conditions. Additionally, the `order by` clause determines how to order the resulting set of data.

When testing SQL queries we must take into account a number of issues that are specific to this language, such as the non procedural character of SQL, the high input and output spaces, the dependence on the database schema, the presence of unknown values (due to the use of tri-valued logics) and in general, the existence of few specifically tailored adequacy criteria to assess the test cases [9].

Mutation testing has been demonstrated as a powerful approach to evaluate test cases and for comparing different testing strategies or techniques. Empirical studies show that the generated mutants provide a good indication of the fault detection ability of a test suite [1,2]. Some previous works in database testing have used mutants to evaluate the fault detection capability of database test cases [4,5,11] in order to assess the effectiveness of test generation techniques. In [3] a set of SQL mutants based on features present in a conceptual model of the database schema is presented. However, all the above approaches are either manual or focus on a reduced subset of SQL features.

In [10] we described a set of mutation operators for SQL `select` queries that covers a wide range of the SQL syntax and semantics. The goal of this paper is to present some internal details about the tool that automates the mutation process. This tool is named *SQLMutation* and it is publicly accessible at *http://in2test.lsi.uniovi.es/sqlmutation*. It allows the mutants to be generated interactively from a Web browser, or from other programs by consuming a Web service.

In the rest of the paper we provide an overview of the mutants (Section 2), some technical details about the tool (Section 3) and typical faults (Section 4). Finally, we present some conclusions (Section 5).

2. SQL mutants

The mutation operators are organized into four categories:

- *SC - SQL clause mutation operators*: These perform mutations on the main clauses: select, join, sub-queries, group by, union, order by and aggregate functions.
- *OR - Operator replacement mutation operators*: These are similar to the expression modification operators described in [7] plus additional operators specific to between and like predicates.

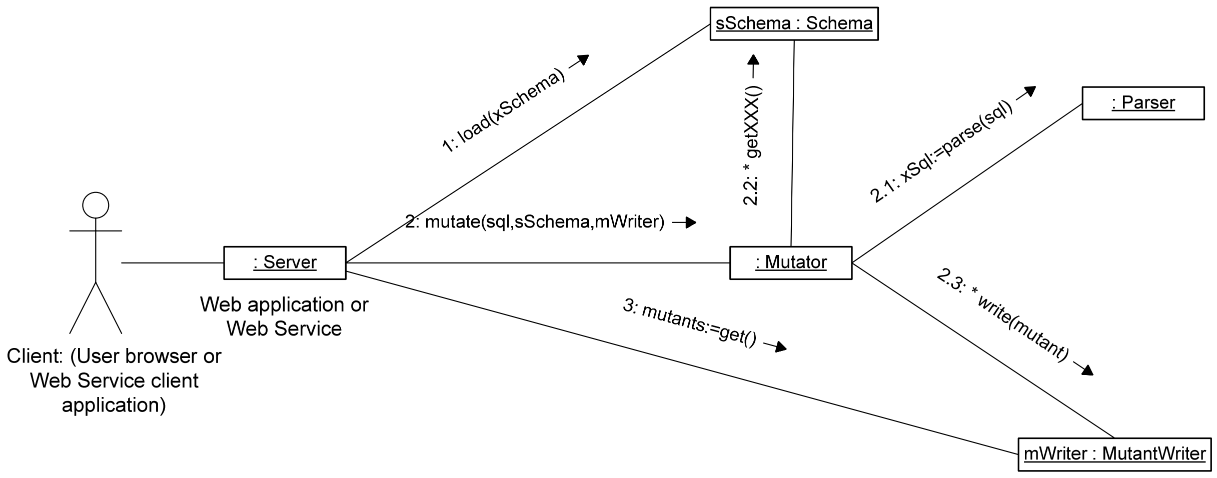


Figure 1. Architecture of the tool

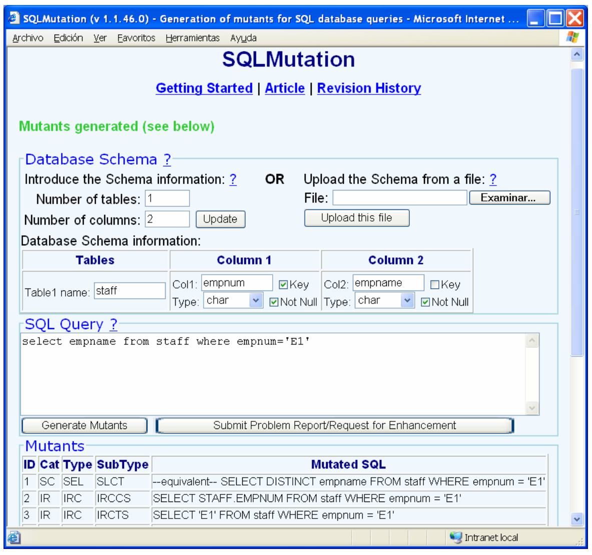
- *NL - NULL mutation operators*: Mutations related to the handling of null values, whose aim is to ensure that test cases exist that exercise the nulls both in the conditions and the query outputs.
- *IR - Identifier replacement mutation operators*: Replacement of columns, constants and query parameters that are present either in the query or in the tables used by the query.

Each category defines several mutation operators or mutant types. As most of the operators can be applied in different SQL clauses, each type is further decomposed into subtypes, each of which refers to a particular mutant type when applied to a given clause. The mutation operators are described in detail in [10].

3. The SQLMutation tool

Figure 1 depicts the main architecture of the tool. Two implementations of the *Server* front-end are available: a Web application for interactive usage and a Web service for use from other applications. The core of the system is the *Mutator* that creates the mutants. This is helped by *Schema* (which provides information about all elements in the database schema), the *Parser* (transforms the SQL query into an internal representation) and the *MutantWriter* which stores the mutants that are being generated and returns them to the *Server* in a suitable format.

In the following subsections each of the components is described in more detail.


SQLMutation (v 1.1.46.0) - Generation of mutants for SQL database queries - Microsoft Internet ...
Archivo Edición Ver Favoritos Herramientas Ayuda
Getting Started | Article | Revision History
Examinar...
Upload this file
Update
Generate Mutants
Submit Problem Report/Request for Enhancement
Intranet local

SQLMutation

Mutants generated (see below)

Database Schema ?

Introduce the Schema information: ? OR Upload the Schema from a file: ?

Number of tables: 1 File:

Number of columns: 2

Database Schema information:

Tables	Column 1	Column 2
Table1 name: staff	Col1: empnum ✓Key Type: char ✓Not Null	Col2: empname ☐Key Type: char ✓Not Null

SQL Query ?

```
select empname from staff where empnum='E1'
```

Mutants

ID	Cat	Type	SubType	Mutated SQL
1	SC	SEL	SLCT	--equivalent-- SELECT DISTINCT empname FROM staff WHERE empnum = 'E1'
2	IR	IRC	IRCCS	SELECT STAFF.EMPNUM FROM staff WHERE empnum = 'E1'
3	IR	IRC	IRCTS	SELECT 'E1' FROM staff WHERE empnum = 'E1'

Figure 2. Main Screen (Web interface)

3.1. Server front-ends

The mutation process requires the user to specify both the SQL query and information about the database schema (table names, column names, data types as well as primary keys and null constraints). The first front-end to *SQLMutation* is a Web application that allows the user to introduce this information from a browser and then generate the mutants. Figure 2 shows a sample of the main screen.

The mutants are presented to the browser in a table, including the classification of each mutant (category, type and subtype) along with the mutated SQL. An additional form is available to submit problem reports or requests for enhancement.

The second front-end is a Web service that provides a method *getMutants* which takes two input parameters: the SQL query to be mutated and the database schema. The database schema must be supplied in an internal XML format used by *SQLMutation*. The following is a sample of a schema that declares a table with two columns, the first is the primary key, and the second has a not null constraint:

```
<schema><table name="staff">
<column name="empnum" type="char" key="true"
  notnull="true"/>
<column name="empname" type="char"
  notnull="true"/>
</table></schema>
```

The Web service response is an XML formatted string which contains all the mutants and/or error information if applicable. A sample of this response is presented below:

```
<sqlmutationws><version>1.1.46.0</version>
<mutants>
  <mutant><id>1</id><category>SC</category>
    <type>SEL</type><subtype>SLCT</subtype>
    <equivalent/>
    <sql>SELECT DISTINCT empname FROM staff
    WHERE empnum = 'E1'</sql>
</mutant>
  <mutant><id>2</id><category>IR</category>
    <type>IRC</type><subtype>IRCCS</subtype>
    <sql>SELECT STAFF.EMPNUM FROM staff
    WHERE empnum = 'E1'</sql>
</mutant>
  . . .
</mutants></sqlmutationws>
```

Using the Web service front-end, third party applications written in different platforms such as Java or .NET are able to integrate the *SQLmutation* tool. In the online documentation, two sample clients written in VB.NET and Java with Eclipse Web Tools Platform are supplied.

3.2. Schema, Parser and MutantWriter

Before generating the mutants, the *Schema* class is instantiated into the *sSchema* object and loaded with the XML representation of the database schema (see above). During the mutation process the *sSchema* object is called repeatedly to obtain columns, constants, their data types, constraints and to resolve the table aliases.

The *Parser* transforms the SQL query into an internal XML format by replacing the keywords by elements and then reorganizing the resulting XML document in order to produce a suitable structure of the query. Keywords are represented as elements and columns, tables, parameters and constants as text. An example of the internal XML representation of the query presented in Figure 1 is the following:

```
<sql><select>empname</select>
<from>staff</from>
<where>empnum<eq/>'E1'</where></sql>
```

While the *Mutator* generates each mutant it calls the *MutantWriter* interface which stores an application-dependent representation of each mutant. For instance, the implementation of the interface used in the Web application stores the mutants in an HTML table object, and the Web service stores the mutants in XML.

3.3. Mutator

The core of *SQLMutation* is a set of classes (represented by the *Mutator* in Figure 1) that receives the schema and the SQL query, calls the *Parser* and loads the XML representation of the query into a DOM model. Then the *Mutator* traverses recursively each element in the DOM and whenever it finds an element or text node performs one or more of the following operations:

- Scope setting: Column references must be mapped to the corresponding table in the database schema that declares it. When visiting each `select` clause (a query may have more than one `select` clause when there is a union or a subquery) or each `join` clause, a list of the tables that are in the scope of this node is updated. An additional list is created including the columns, constants and parameters needed for the replacements made in by the IR category of mutants.
- Column resolution: When a text node is found, the Schema determines whether it is a constant, parameter or column reference. In the last case, the table that declares the column is searched for in the list. When a table appears more than once, the references are mapped to the corresponding aliased table.
- Mutation: If the visited node is an element or text node suitable for generating one or more mutants, a clone of it is first created and then every necessary transformation to obtain the mutant(s) is performed on the clone. Immediately after generating each mutant, the *MutantWriter* is called to save the transformed clone into a new mutant.

After finishing the mutation process, the server application will call the *MutantWriter* to get all generated mutants and send them back to the client.

4. Using the SQL mutants

Next, we will present some results of the faults introduced in a set of queries developed by seven students during an exercise of SQL development. The exercise consisted in writing four queries using the same database schema. All of the queries must join three tables using different kinds of join-types. The queries have simple where-conditions and two of them have a `group by` and `having` clauses.

We generated the mutants of the desired queries, removed the equivalent mutants and executed them against the test databases developed by the students. The mutation scores are presented in Table 1.

Table 1. Mutation Scores

Category	Q1	Q2	Q3	Q4	Tot.
IR – Identifier Replac.	98,8	94,2	87,9	78,1	89,1
NL – Nulls		96,4	77,1	60,0	78,4
OR – Operator Replac.	87,4	80,0	81,4	74,0	80,3
SC – SQL Clauses	86,8	39,0	50,4	69,6	59,6
Total	89,6	80,7	75,0	73,6	78,5

We examined the queries written after the exercise and counted the kind of faults committed in each one. The result is presented in Table 2.

Table 2. Faults committed in the queries

	Q1	Q2	Q3	Q4	Tot.
Omitted distinct in select	1				1
One join with wrong type				3	3
Two joins with wrong type	3	3			6
Use a wrong SQL86 join		1		1	2
Join incorrect tables		1			1
Wrong columns in order by		1			1
Order by omitted		3	4	4	11
Unnecessary IS NULL		1			1
Omitted IS NULL			5	4	11
Wrong columns in select-list		1			1

The faults were related to the `distinct` quantifier, the `join` and `order by` clauses, the use of the `is null` predicate and the usage of wrong columns. All of them are faults that are represented by the mutants. Most of the faults committed are represented by the mutants that belong to the SC and NL categories, which are those that have achieved the lowest mutation scores (Table 1).

The above provides us with some insight into how well the mutants model the real faults committed in this exercise. However, if we use the SQL mutants to assess the adequacy of database tests developed while writing a query, some relevant faults may not be represented by the mutants. For instance, a frequent fault is the omission of the `order by` clause. If we generate the mutants of a faulty query that has an incorrect `order by` clause, the mutants can assist us in detecting this fault. But, if the faulty query does not include an `order by` clause, no mutants will be generated for this clause and consequently this fault will not be detected. A potential new mutation operator may be needed in that case to insert additional `order by` clauses.

5. Conclusions

We presented a tool that automatically generates mutants of SQL database queries. The tool can be used interactively or integrated in other tools that consume the Web service exposed. This system is intended to be used by researchers in database applications testing for assessing the adequacy of database test cases and for comparing different techniques.

The set of the mutants generated covers a wide range of SQL features, although there is room for improvement and refinement of the mutants. We encourage researchers to use it and thus, to collaborate in its evolution.

6. Acknowledgements

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

7. References

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

[2] J. Andrews, L. Briand, Y. Labiche, A. S. Namin, Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria, IEEE Transactions on Software Engineering 32(8) 608-624.

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

[4] Y. Deng, P. Frankl, D. Chays, Testing Database Transactions with AGENDA. Proc. of the 27th International Conference on Software Engineering, ACM Press, New York, NY, USA, 2005, pp. 78-87.

[5] 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.

[6] International Standards Organisation, Information technology – Database languages – SQL, ISO/IEC 9075:1992, third edition.

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

[8] R. Pönighaus, 'Favourite' SQL-Statements – An Empirical Analysis of SQL-Usage in Commercial Applications. Proc. of the 6th International Conference on Information Systems and Management of Data (LNCS, vol. 1006), Springer, 1995, pp. 75-91.

[9] J. Tuya, M.J. Suárez-Cabal, C. de la Riva, A practical guide to SQL white-box testing, ACM SIGPLAN Notices, 41(4) 36-41, 2006.

[10] J. Tuya, M.J. Suárez-Cabal, C. de la Riva, Mutating Database Queries, Information and Software Technology, 2006. (In press, doi:10.1016/j.infsof.2006.06.009).

[11] 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.