A practical guide to SQL white-box testing

Javier Tuya, Mª José Suárez-Cabal, Claudio de la Riva
Departamento de Informática, Universidad de Oviedo
Campus de Viesques, s/n, E-33204 GIJON / SPAIN
{tuya | cabal | claudio}@uniovi.es

Abstract

SQL is a ubiquitous language used in a wide range of applications for accessing the data stored in relational databases. However, the usual software testing techniques are not designed to address some important features of SQL. We present a set of practical guidelines for designing white-box tests cases that reasonably exercise the way in which an SQL query processes the stored data. These guidelines are illustrated using an example.

Keywords

Software Testing, Database Testing, SQL Query Language

1. Introduction

Although many software testing techniques have been proposed and adopted in day to day industrial practice, these are not specifically tailored for handling the particularities of the Structured Query Language (SQL). However, many commercial and bespoken applications, ranging from legacy to modern web-based software, employ relational databases and we still write their database queries using SQL. Even though new promising approaches such as persistence systems and XML-based data stores are emerging, investments in applications using relational databases are both huge and growing. This indicates that old and new applications will continue to use SQL for years. On the other hand, agile development processes lead to an increase in the unit testing effort, database queries included.

Many studies conducted in academic and industrial settings [1,7] characterize the sources of common faults that programmers commit when writing queries. In general, the majority of SQL features are likely to contain bugs and many of these are different in nature compared to the bugs in other languages: common problems involve the joins, duplicates, missing values, etc. The process of writing and unit testing database queries has some significant distinguishing particularities:

- The SQL language is not procedural (or is at least far from being so) and programs involve a mixture of procedural (e.g. Java) and non procedural (SQL) code.
- Queries have two different kinds of inputs: parameters and a complex set of data structures stored in the database tables, which make the input space very large.
- Output is in the form of a table, contributing to complicating the task of determining the desired outputs of a test.
- Database queries processing is highly dependent on the database schema: both its structure and constraints. Small changes in any of these may entail a change in the behaviour of many queries.
- Decisions use three-valued logic instead Boolean logic. Fields in the database that contain undefined values sometimes cause unexpected behaviour if the programmer is not aware of this.
- When designing a test case, we identify the different situations or database states that exercise the query. We try to include quite a lot of these database states in each single test case in order to keep the number of database loads small. Therefore, we usually have fewer, but more complex test cases in comparison with the testing of imperative programs.
- There is a lack of sound adequacy criteria for assessing the completeness of a database test suite and for guiding the tester to develop test cases.

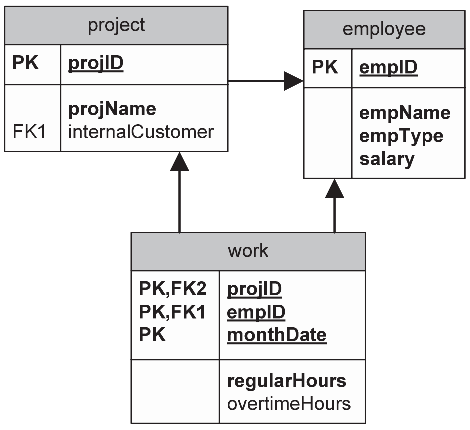
The main processing that SQL performs consists in the selection and joining of data rows from one or more tables. The `select` clause determines which fields (columns) constitute the query output, the `from` clause determines which tables are used and the `join` determines the criterion for joining rows from different tables (join-conditions). Then the `where` clause filters the rows based on some other criteria (where-conditions). The `group by` clause indicates how to combine the selected rows and the `having` clause performs a final filter based on other conditions (having-conditions). Additionally, the `order by` clause determines how to order the result set.

In the rest of the paper we present a number of SQL testing guidelines adapted from well-known testing techniques (Section 2), some issues on current research and test automation (Section 3) and the concluding remarks (Section 3).

Can you spot the bugs in this query?

```
SELECT p.projName, sum(w.overtimeHours) Overtime, sum(w.regularHours) Regular
FROM project p
LEFT JOIN work w ON p.projID=w.projID
LEFT JOIN employee e ON w.empID=e.empID
WHERE (e.empType='P' OR e.salary<3000)
  AND e.empID NOT IN (select internalCustomer from project)
GROUP BY p.projName
HAVING sum(regularHours)< 0.95*sum(regularHours+overtimeHours)
```

Information request: Management wants to know the projects and hours (both regular and overtime) in which the number of regular hours is less than 95% of the total number of hours. Only take into account the hours spent by programmers or by personnel whose monthly salary does not exceed $3,000 provided that they have not been the internal customer in any project.



Data model:
The database stores information about projects, employees and the monthly time spent by each employee on each project. Note: we assume constraints for primary and foreign keys as indicated in the figure. All values are valid and inside of its domain. Integer numbers are positive.

Project:
- projID int not null – the unique ID for each project.
- projName char(32) not null – the descriptive name for the project.
- internalCustomer int – if the customer of the project is an employee, this field stores his/her ID.

Employee:
- empID int not null – the unique ID for each employee.
- empName char(32) not null – the full name for the employee.
- empType char(1) not null – employee category (P: programmer, M: manager).
- salary int not null – monthly salary in US $.

Work:
- monthDate char(6) not null – the month of a work record, format yyyymm.
- regularHours int not null – the total number of regular hours spent by an employee on a project in a month
- overtimeHours int – the total number of overtime hours spent by an employee on a project in a month

Figure 1. Example query containing a number of bugs.

2. Five guidelines for designing SQL unit test cases

Figure 1 displays an example of an SQL query containing a number of bugs. We shall perform a systematic white-box test on it using the following guidelines.

2.1. Adopting MCDC for SQL conditions

A query takes decisions about the data that it will retrieve at three places: join, where and having conditions. We can see these decisions as a sequential series of filters or as a decision composed of a conjunction of conditions. It therefore seems reasonable to adopt some kind of control-flow based criterion for designing tests. Since a test can only reveal faults by comparing the actual output against the expected output, it is important to prevent tests designed to cover a condition from being masked by other conditions. For instance, if a test fulfils a multiple condition criterion in the where clause, it becomes useless if the rows selected by the where-condition are further discarded by the having-condition.

A reasonable criterion that we can use here is the Modified Condition Decision Coverage (MCDC), defined in the RTCA/DO-178B standard, which has been demonstrated as representing a good balance of test-set size and fault detecting ability [11]. MCDC requires that every condition has taken all possible outcomes at least once, and each condition has been shown to independently affect the decision's outcome.

Each compound AND condition will require a test state where all conditions are true, and additional test states for each of the conditions such that only this condition is false. In the case of OR's, these will require a test state where all conditions are false and additional test states for each one such that only this condition is true.

Many errors occur in the boundaries of decisions. Consequently we impose an additional criterion: When designing the test states that show that a condition independently affects the decision's outcome, this change in the outcome must be caused by the shortest possible variation of one of the operands involved in the condition.

We must also apply these guidelines to the other main SQL statements (update, insert and delete), as they usually include select and where clauses.

Figure 2, part A provides an example of the application of this criterion in the design of test cases for the example query in Figure 1.

2.2. Adapting MCDC for tackling with nulls

SQL's handling of unknown values (null) sometimes turns into a nightmare if developers are not aware of it. For instance, the evaluation of a where condition of the form `a AND b`, returns unknown when either `a` or `b` evaluate to unknown, and the row that would result if it were evaluated to true is excluded from the result set. The way in which a query must handle the nulls depends on its exact meaning in terms of business (e.g. not yet known but possibly in the future, definitively unknown, not relevant or not appropriate). As a result, we need to include tests that exercise nulls whenever possible.

We must switch over from the Boolean logic used in the MCDC criterion to a three-valued logic. For instance, a compound condition of the form `a AND b` will require two additional test states in which each of the conditions is null while the other is true.

If a condition is composed of more than one column reference, we require one test state that makes each of them null while the other/s is/are not null. In the case of join-conditions, the null values appear for two different reasons: a) the joined column is null in the database, though most frequently b) a null value has been generated as a consequence of an incomplete join (left, right or outer join). The latter case will allow the typical situations of master without detail and vice versa to be exercised, detecting faults in the type of join.

Figure 2, part B shows an example of the design of this kind of null aware test cases.

2.3. Category partitioning selected data

We can consider an SQL query as a short formal declarative specification and then apply a sort of specification-based method such as the Category Partition Method [8] (CPM). We define a set of categories in the data and then, for each one, a set of choices that will specify the test inputs. We provide the following set of categories for SQL:

- **Rows that are retrieved**: We include a test state to force the query to not select any row. This does not always imply that the result set is empty. In the case of queries involving aggregate functions in the output, when the query does not select any row, the result set gives 0 for `count` and null for the other aggregate functions.
- **Rows that are merged**: The presence of unwanted duplicate rows in the output is a common failure in some queries. We include a test state in which identical rows are selected. Then the `select` clause will output rows that are exactly the same, while the `select distinct` will remove duplicates. The same is applicable to the union of queries. In this case, the `union` clause will prevent obtaining duplicate rows, while the `union all` clause will include all rows, including duplicates, if any.
- **Rows that are grouped**: For each of the group-by columns, we design test states to obtain at least two different groups at the output, such that the value used for the grouping is the same, and all the other are different. Null values are considered as an independent value when grouping the rows. Therefore, for each of the group-by-columns, we also need to include a test state that generates at least one null group while all the other are not null.
- **Rows that are selected in a subquery**: For each subquery, we include test states that return zero and more rows, with at least one null and two different values in the selected column. We include additional test states for subqueries with `all`, `any` or `some`.
- **Values that participate in aggregate functions**: For each aggregate function (excluding count), we include at least one test state in which the function computes two equal values and another one that is different. This allows us to differentiate the functions that evaluate over distinct values. Additionally, each aggregate function must be evaluated over some null value.
- **Other expressions:** We also design test states for expressions involving the like predicate, date management, string management, data type conversions or other functions using category partitioning and boundary checking.

Input: project

	proj ID	proj Name	internal Customer
P1	1	prj1	2
P2	2	prj2	2
P3	3	prj3	2
P4	4	prj4	2
P5	5	prj5	2
P6	6	prj1	2
P9	9	prj9	2
P10	10	prj10	null
P11	11	prj11	9

Input: work

	proj ID	emp ID	month Date	regular Hours	overtime Hours
W11	1	1	200601	100	10
W12	1	2	200602	200	20
W13	1	3	200603	300	30
W14	1	4	200604	100	10
W15	1	5	200605	500	50
W21	2	1	200601	100	5
W21b	2	1	200602	200	11
W31	3	1	200601	100	5
W31b	3	1	200602	200	10
W41	4	1	200601	100	30
W41b	4	1	200602	200	null
W51	5	1	200601	0	10
W61	6	1	200601	100	10

Input: employee

	emp ID	emp Name	emp Type	salary
E1	1	emp1	P	2000
E2	2	emp2	P	2000
E3	3	emp3	M	3000
E4	4	emp4	P	4000
E5	5	emp5	M	2999
E9	9	emp9	P	2000

Desired output

	proj Name	overtime	regular
O1	prj1	100	1000
O2	prj2	16	300
O3	prj4	30	300
O4	prj5	10	0
O5	prj1	10	100

Rationale:

Part A: Conditions without nulls

- E1, E2, P1, W11: Initial rows, added to cover all conditions as true.
- E3, E4, E5, W13, W14, W15: Added to cover where conditions in the OR.
- W12: Added to cover where conditions in the subquery.
- P9: Added to cover the first join condition, the other is already covered.
- P2, W21, W21b, P3, W31, W31b: Added to cover the having conditions (in the boundaries).

Part B: Conditions with nulls

- E9: An employee without work records to cover null conditions in the second join. The other conditions are already covered or forbidden by referential integrity constraints.
- P10: A project without internalCustomer to cover null condition in the subquery (it leads the subquery to return a null value).
- P4, W41, W41b: Added to cover nulls in having (it leads the total time in the having to return null).

Part C: Other

- To attain an output with zero rows, add a new independent test case by removing all works with the exception of W13.
- P6, W61: Two projects are merged in the same group (they have the same name).
- Rows that are grouped and values in aggregate functions are already covered.
- P11: Added to force the subquery to return different values.
- P5, W51: Forces zero overtime hours at the outputs, other zero and null values are not possible due to the having condition.

Figure 2. Test cases designed for the example query.

2.4. Checking the outputs

Before reading the following paragraphs, please switch to Figure 3 and try to find the bugs in the programs.

A common-sense rule of thumb is to design test cases to cover the output domain by checking valid and invalid values. As null values belong to the domain of variables, we also design a special set of test states to select rows having null values in each column in the result set. This kind of test is of major importance, as the behaviour of database nulls stored in program variables is highly dependent on the host language and does not match the behaviour of nulls in SQL.

In the query in Figure 3, if a row stores a null temperature, then the Java program gets a 0.0 value in the `temp` variable which is wrongly converted to 32.0 Fahrenheit degrees. The program should check the nullity of the value using the `wasNull()` method and then skip the calculation for these cases.

The Visual Basic implementation behaves differently, as it breaks down due to an invalid cast exception. In this case, null values do not provide any valid value and the `temp` variable stores a special value named `DBNull` that causes the failure. Note that the breakdown is not due to the selection of incorrect column (in this implementation, column numbers start at 0). As in the previous case, the program should also check the nullity of the value using the `Convert.IsDBNull()` function.

A well known testing principle states that we must check whether a program does what it is not supposed to do. To update queries, we need to check the whole content of tables that are being updated. This task is easy if we have some kind of test automation and we keep the test database small.

Can you spot which program has a bug?

Problem statement: We store in a table the pairs of cities and their temperatures in degrees Celsius. For each of the rows, display the temperature in degrees Fahrenheit.

Java/J2SE platform:
```
// assume that cn is an instance of an open Connection
String sql="select city,celsiusTemp from temperatures";
Statement stmt = cn.createStatement();
ResultSet rs=stmt.executeQuery(sql);
while (rs.next()) {
      double celsius=rs.getDouble(2); //the first column is 1
      double fahrenheit=celsius*1.8+32;
      System.out.println(celsius + " " + fahrenheit);
}
```

Visual Basic/.NET platform:
```
' assume that cn is an instance of an open OdbcConnection
Dim Sql As String = "select city,celsiusTemp from temperatures"
Dim cm As New OdbcCommand(Sql, cn)
Dim dr As OdbcDataReader = cm.ExecuteReader()
Do While dr.Read()
    Dim Celsius As Double = dr.GetValue(1)  'the first column is 0
    Dim Fahrenheit As Double = Celsius * 1.8 + 32
    Console.WriteLine(Celsius & " " & Fahrenheit)
Loop
```

Figure 3. Behaviour of database nulls in programming languages.

2.5. Checking the database constraints

Many of the test states described here may be impossible due to database constraints. But changing the database schema is rather frequent during development. If changes lead to an increase in constraints, then the designed test cases will probably fail to set-up while loading data. However, if changes lead to a decrease in constraints, test cases may become incomplete, as some situation was not exercised when designing the test case. We must include additional test cases to ensure that that all assumed constraints are enforced by the database. All kinds of constraints must be checked (e.g. insert a record with a duplicate primary key, insert and delete records to violate referential integrity, insert null values in not null fields, or update fields with values outside of their domains).

3. Research and tools

Although research in the testing field has originated or suggested many practices in use in industry, our current testing technique knowledge is still limited [5]. Some recent research on white-box testing for database applications addresses issues such as test case generation [3,4,12], the translation of SQL into a procedural language [2] or the establishment of adequacy criteria, either using control-flow based [9] or data-flow based [6,10] criteria. This field of research has, however, received little attention.

We write unit tests and easily run these using tools such as JUnit (www.junit.org). Also, we employ coverage tools to highlight program statements and conditions that have not been tested. But an SQL query is executed in a single statement inside an imperative program and we would like to evaluate the coverage of that single query in relation to the test data. The bad news is that we do not have such a tool.

One of the problems with the management of database test cases is the amount of data needed to perform the assertions for comparing expected and actual outputs. This task is facilitated using JUnit extensions such as DBUnit (www.dbunit.org). Setting a test case is as easy as extending the `DatabaseTestCase` class (which is a subclass of `TestCase`), specifying the database connection information and then implementing the test methods. DBUnit implements special assert methods for comparing tabular structures (`IDataSet` interface) as well as for reading and writing them from/to a variety of formats (XML, database or CSV). You can find a number of practices and more documentation and links on the DBUnit's web page.

Table 1. Faults found in the example query.

Failure detected	Fault	Bug fix
Query does not include employees whose salary is 3,000	Wrong where condition in the OR	Set `<=` instead of `<`
Query result is empty if a project does not have an internal customer	If the subquery returns any null, the NOT IN predicate will never return true.	Add a where condition to the subquery of the form: `internalCustomer is not null`
Work records without overtime hours are not taken into account	If a record has null in overtimeHours, the total number of hours in this month is counted as null	Enclose this field in a coalesce predicate: `coalesce(overtimeHours,0)`
Information about two projects with the same name is merged in the same group	The grouping is performed only in the projName field	Add `projID` to the group by clause

4. Concluding remarks

Table 1 displays four bugs discovered in the example query in Figure 1 after running the test cases developed in Figure 2. All are common errors that programmers commit when writing queries and three of them are particular to SQL.

The guidelines presented are an adaptation of well-known techniques and they are simple and straightforward. When applied systematically, they can help the practitioner to achieve more complete unit tests for queries and be used for training in the task of writing and testing SQL.

Acknowledgements

The National Program for Research, Development and Innovation of the Ministry of Education and Science (Spain) supported our work under grants TIN2004-06689-C03-02 and TIN2005-24792-E.

References

[1] S. Brass, C. Goldberg, "Semantic Errors in SQL Queries: A Quite Complete List". *Journal of Systems and Software*, 2005, in press.

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

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

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

[5] N. Juristo, A.M. Moreno, S. Vegas, "Reviewing 25 Years of Testing Technique Experiments". *Empirical Software Engineering*, Vol. 9(1), 2004, pp. 7-44.

[6] G.M. Kapfhammer, M.L. Soffa, "A family of test adequacy criteria for database-driven applications". *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.

[7] H. Lu, H.C. Chan, K.K. Wei, "A Survey on Usage of SQL", *SIGMOD Record*, Vol. 22(4), 1993, pp. 60-65.

[8] T.J. Ostrand, M.J. Balcer, "The category-partition method for specifying and generating functional tests". *Communications of the ACM*, Vol. 31(6), 1988, pp. 676-686.

[9] M.J. Suárez-Cabal, J. Tuya, "Using an SQL Coverage Measurement for Testing Database Applications", *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, NY, USA, 2004, pp. 253-262.

[10] D. Willmor, S.M. Embury, "Exploring test adequacy for database systems", *Proceedings of the 3rd UK Software Testing Research workshop*, 2005, pp. 123-133.

[11] Y.T. Yu, M.F. Lau, "A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions". *Journal of Systems and Software*, 2005, in press.

[12] J. Zhang, C. Xu, S.C. Cheung, "Automatic generation of database instances for white-box testing", *25th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, Los Alamitos, CA, 2001, pp. 161-165.