Test Data Generation for Database Applications

Pooja Agrawal, Bikash Chandra, K. Venkatesh Emani, Neha Garg, S. Sudarshan
IIT Bombay
{*poojaagrwl15, bikash, venkateshek, nehagarg15, sudarsha*}@*cse.iitb.ac.in*

***Abstract*—Unit test cases have become an essential tool to test application code. Several applications make use of SQL queries in order to retrieve or update information from a database. Database queries for these applications are written natively in SQL using JDBC or using ORM frameworks like Hibernate. Unit testing these applications is typically done by loading a fixed dataset and running unit tests. However with fixed datasets, errors in queries may be missed. In this demonstration, we present a system that takes as input a database application program, and generates datasets and unit tests using the datasets to test the correctness of function with queries in the application. Our techniques are based on static program analysis and mutation testing. We consider database applications written in Java using JDBC or Hibernate APIs. The front-end of our system is a plugin to the IntelliJ IDEA IDE. We believe that such a system would be of great value to application developers and testers.**

I. Introduction

Application testing is usually done by running multiple unit test cases, each with a different set of inputs, and then checking if the results match the expected results or not. Several applications use databases to query and update stored data. Database calls from an application are typically made using either native SQL queries using frameworks like JDBC, or using ORM (Object-Relational Mapping) frameworks like Hibernate.

Unit testing of applications that make calls to the database is usually done by loading a fixed dataset into the database and then running unit test cases. However, this approach of testing using fixed datasets does not guarantee application correctness in all cases. Subtle errors in queries may be missed if the dataset does not contain data to test these errors. We will illustrate this using an example.

Consider the function in Figure 1[1] derived from a real world application. The function returns the list of builidings along with the number of venues that are free on a particular date. The function takes as input *user_id* and *date* and fetches the corresponding *group_id*. Users can access the information on buildings corresponding to their group. However, an admin user (*group_id=0*) can access information for all buildings. Both SQL queries on lines 6 and 11 are incorrect since they will not return buildings when there are no venues in the building that are free. For example, with the database instance shown in Figure 2 and with an input date as ‘2017-12-10’ and a user corresponding to *group_id= 3* or *0*, the query will not list the Nilgiri building which has no remaining venues with *booking_status* as false, when it should with count 0. On other datasets the query may return correct results.

[1]We use a pseudo function executeQuery that takes a query, executes it and returns the result as a scalar/collection.

```
1  getFreeVenues(user_id, date) {
2    q1 = "select group_id from users where user_id=?";
3    q1.setParam(1, user_id);
4    group_id = executeQuery(q1);
5    if(group_id==0) { //admin user
6      q2 = "select b.b_name,count(*) from
7        building b join venue_booking v on (b.b_name=v.b_name)
8        where date=? and v.status=false
9         group by b_name";
10     q2.setParam(1, date);
11   } else {
12     q2 = "select b.b_name,count(*) from
13       building b join venue_booking v on (b.b_name=v.b_name)
14       where date=? and g.group_id=? and v.status=false
15        group by b_name";
16     q2.setParam(1, date);
17     q2.setParam(2, group_id);
18   }
19   return(executeQuery(q2));
20 }
```

Basic block region — Conditional region — Sequential region

Fig. 1: Booking status program

b_name	group_id
Himalaya	3
Nilgiri	3

(a) builing instance

venue_id	b_name	date	status
21	Himalaya	2017-12-10	false
11	Nilgiri	2017-12-10	true

(b) venue_booking instance

building_name	count
Himalaya	1

(c) Figure 1 result

building_name	count
Himalaya	1
Nilgiri	0

(d) Expected result

Fig. 2: Database instances (only relevant columns are shown) and results for date ‘2017-12-10’ and user with group_id=3

In this demonstration, we present the *XDataPro* system that solves the problem of test data generation for queries in database applications. Given an input program with embedded SQL queries, XDataPro generates test datasets, program input values, and unit tests that use these datasets and values. The unit tests are aimed at checking the correctness of functions with queries in the program.

XDataPro leverages the DBridge [1] system for static program analysis to identify queries and relevant constraints for all execution paths of the program (details in Section II). These are then passed on to XData [2], [3] for data generation. Given an input query, XData generates multiple datasets each

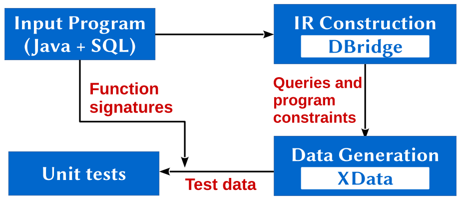


Fig. 3: XDataPro architecture

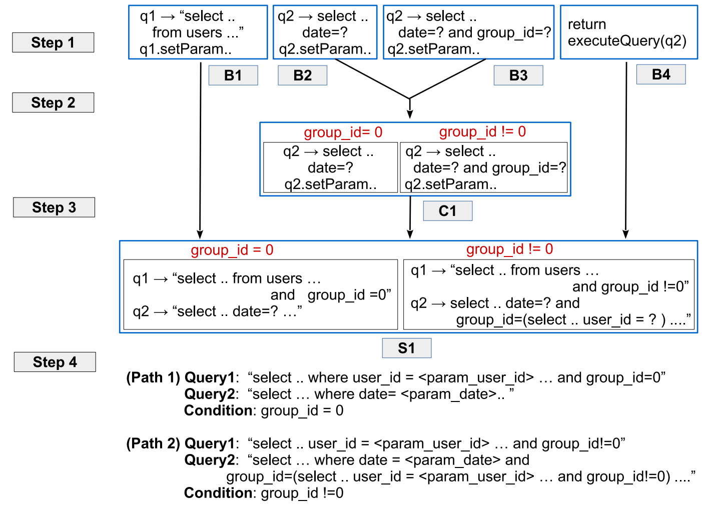


Fig. 4: Walk-through of IR construction

targeted at catching one or more common errors in the query. XDataPro extends XData to generate test data for queries as well as program input parameters by taking into account program constraints (details in Section III).

A key difference between XData and XDataPro is that the input to the latter is a database application program, in which SQL queries are intertwined with imperative code. Thus the queries are not readily available and must be identified from the given program. However, this is not trivial since queries in programs are often constructed dynamically and the query may be different in different execution paths of the program. Moreover, there may be constraints on query parameters and results imposed by the program, and results of one query may be used in another query. For example, the program in Figure 1 may execute one of two queries corresponding to lines 5 or 11 based on whether *user_id*=0 or not. The *user_id* itself is determined by the result of the query *q1*. Hence when extracting the query from program, we also need to take into account the context of the program under which the query runs.

Once the datasets for queries have been generated, XDataPro generates unit tests that can load the datasets one by one, and check the correctness of result of functions containing these queries. Our generation of unit tests is template-based. A unit test for testing a function containing queries consists of the inputs to the function, the dataset on which queries are run, and the output of the function. The developer or tester may check if the output matches the expected output or not, and then the test cases are added for regression testing in future. The user may add outputs that dont match the desired output as negative regression unit tests, if they desire.

Figure 3 summarizes the architecture of the XDataPro system. Our implementation focuses on Java programs using JDBC or Hibernate for database access, but the techniques themselves are not tied to any programming language or data access framework. The front-end to our test generation tool is a plugin for the IntelliJ IDEA IDE. The plugin enables users to interact with our system through a simple graphical user interface. Details are described in Section IV.

II. Program Analysis

In this section, we discuss our techniques that use static program analysis to identify the queries, and the constraints on query inputs/outputs from a database application program. We first discuss our intermediate representation (IR) before outlining our approach and its capabilities.

A. Intermediate Representation

Our IR is based on the DAG based representation for database applications proposed by Emani et al. [1] for translating imperative code to SQL. The IR from [1] is essentially a variable to expression map. The expression represents the value of the variable at any point in the program in terms of the program inputs (intermediate assignments are bypassed). In this paper, we use an array of such variable-expression maps, one map for each alternative execution path in the program. Each map is also annotated with a condition. The map is valid for the program execution path in which the annotated condition evaluates to true.

For example, consider our IR in Figure 4 after Step 4 (labeled S1). It consists of two maps corresponding to whether the *group_id* corresponding to the user is '0' or not. These correspond to the two execution paths generated by the if-else construct from line 5 of Figure 1.

B. IR Construction using Regions

Real world programs can contain complex control flow including branching and loops. In our approach, we use the concept of *program regions* to systematically construct our IR for such complex programs.

Regions [4] are structured fragments in a program, such as straight line code, if-else blocks, loops, etc. A *basic block region* represents straight line code, a *conditional region* represents an if-else block, a *loop region* represents a loop, and a *sequential region* represents a sequence of two (or more) regions one after another. Program regions for Figure 1 are shown alongside the code.

A walk-through of our IR construction for Figure 1 is shown in Figure 4. Each IR is annotated with its corresponding region, as marked in the program. The first step is to construct IR for basic blocks. This is shown alongside *Step 1* in Figure 4. Note that the IR for each basic block consists of a single map, and there are no conditions associated with the map. Merging the blocks B2 and B3 info conditional region C1 in step 2 gives us two maps, one corresponding to group_id=0 and the other corresponding to group_id!=0. Merging the blocks B1, C1 and B4 in step 3 gives us the final IR with maps and relevant conditions for each program execution path. Once we have the final IR, in step 4, we consider each path separately and generate unit tests for paths after extracting the queries and the conditions for the path. The extracted SQL queries are then passed to XData for generating test data for each execution path. Note that for path 2 the *group_id* input of q2 depends on the result of query q1. We take this into account by expressing the *group_id* parameter in q2 in terms of the query q1.

Our analysis is flow-sensitive (takes into account the order of the program) and path-sensitive (takes into account different paths in a program). Our approach for IR construction also performs constant folding for dynamically constructed queries.

C. Supported Program Constructs

Our system is able to extract queries and constraints from real world programs with complex control flow. The program constructs handled by our system includes

- Arbitrary levels of **if-else** branching, interspersed with straight line code. Figure 5a in Section IV is one such example.
- Arbitrary levels of nested **function calls** without recursion.
- **Reuse** and reassignment of variables. The same variable may be used to construct and execute multiple queries, at different program points. Our system is able to extract all such queries.
- **Multiple queries** in the same program execution path.
- **Chained queries** where the results of one query are used (directly or indirectly) to construct another query.
- **Constraints** on query parameters and constraints on result set attributes.
- **Loops**: We only consider cursor loops with some restrictions, detailed below.

Restrictions on Loops: In general, the number of iterations in a loop is unknown at compile time. A special case of loops that iterate over a query result set/collection, which are called *cursor loops*, are widely used in database applications for iteratively processing query results. Our system supports test data generation for programs containing cursor loops.

When the loop body does not contain any branching, all the paths in the loop are covered by generating datasets for the following paths: (i) empty dataset to cover the case with no iterations of the loop, and (ii) other datasets to cover the loop body.

If the loop body has branching and if the branch conditions are all predicates of the current tuple or loop invariant variables only, we generate SQL queries such that generated datasets would be sufficient to cover every path present inside the loop at least once. In general, if the loop body has branching the number of possible paths is not bounded by the program size, and it may not be possible to determine the sequence of paths using static program analysis techniques.

An example of cursor loops using the Hiberanate ORM framework is shown below, which is extracted from Wilos, an open source orchestration software.

```
for(Project p: getAllProjects())
  if(!(p.isFinished()))
    unfinP.add(p.getId());
```

The above function computes the set of projects whose status is marked as unfinished. getAllProjects() (line 3) internally uses Hibernate API calls to fetch the list of all projects. This list is then filtered inside the application and a set of project id's satisfying the condition are returned.

Given such a program, our system first translates this program into an equivalent program that uses SQL queries, using DBridge. DBridge contains techniques to translate relational operations such as projections, selections, joins and aggregations performed using loops in imperative code into a query. For instance, the above program is translated as follows:

```
Query query = Utils.getSession().createSQLQuery
    (“select id from Project where isFinished <> 1”);
```

After rewriting the program as above, the approach discussed in Section II-B can be used to extract queries and relevant constraints.

Applications Using ORM: SQL queries are explicit in JDBC programs. However, in programs using Hibernate, joins may also be implicitly realized by specifying associations between attributes of mapped classes. DBridge is able to obtain explicit SQL queries in such cases [1], from which XData can generate datasets. We omit details for lack of space.

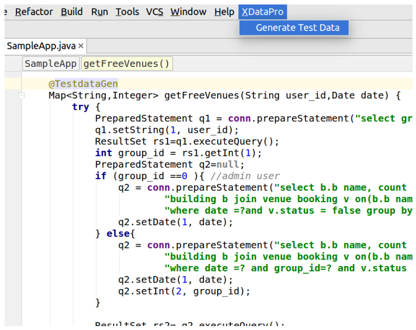
III. TEST DATA GENERATION

Once the SQL query and relevant constraints from the program are obtained, we use the XData [2], [3] system for generating the test datasets. The datasets are designed to catch common errors in SQL queries. The errors in queries are modeled as query mutations. A dataset that is able to produce different results on the correct query and its mutant (thereby showing that the mutant is not equivalent to the correct query) is said to kill the mutations.

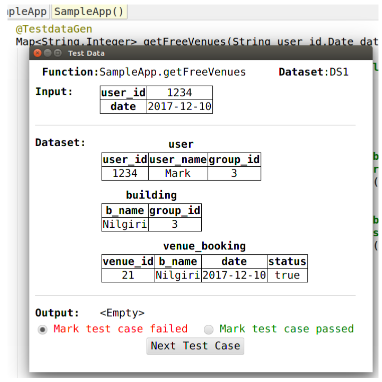
The type of mutations considered include join type mutations (inner/outer), join condition mutations, selection condition mutations, aggregate operator mutations, group by attribute mutations, mutations in string patterns, like clause mutations, distinct clause mutations, subquery connective mutations and set operator mutations, amongst others. XData generates several datasets for each query. Each dataset is targeted to kill one or more mutations. In order to kill a mutation we need to ensure that the dataset satisfies some constraints. XData encodes these constraints along with database constraints in the CVC3 [5] solver. XData then uses the solver to generate a dataset that satisfies the constraints.

In the case of testing applications with embedded queries, which is the focus of this paper, there may be additional constraints due to the program in addition to the constraints imposed by the query. We appropriately encode any such arithmetic/string constraints imposed by the program into constraints that we pass to the solver. We also pass the program input parameters to the solver to get back values that may be used when invoking the program/interface for unit testing.

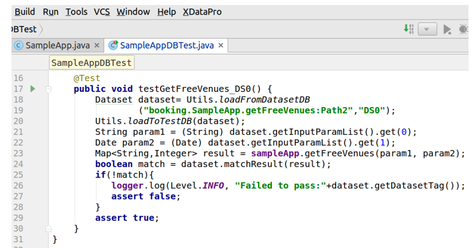
Related Work: Although mutation testing is a well know technique for testing applications in general, these techniques do not consider queries embedded in the application. [6], [7] focus on test data generation to ensure path coverage for database applications but do not take into account testing of SQL queries. Qex [8] generates a test database for a database application along with query parameters such that certain



(a) Function for test data generation



(b) User interaction on test results



(c) Unit test case

Fig. 5: Unit test generation for a sample application

properties in the query results are satisfied (e.g. the query result is non-empty) but does not consider mutation testing of queries. [9] considers mutation testing of queries in database applications but only handles mutations involving WHERE and HAVING clause, unlike our system.

IV. Demonstration

In this section, we describe the use of our plugin to configure and use the XDataPro system. Our demonstrations will showcase the ability of XDataPro to (a) identify queries in database applications, (b) generate test data for these queries and program inputs, and (c) generate unit tests that use the generated test data.

Our demonstrations will use Java programs that access the database using JDBC or Hibernate. Programs derived from real world applications as well as sample programs based on the University schema from [10] and the TPC-H schema will be provided. These applications will contain SQL queries that have some errors. A PostgreSQL database would also be provided against which the programs can run.

The plugin can be installed as a third party tool on top of an existing IntelliJ IDEA installation. Installing the plugin will add a new main menu item titled “XData”. This is shown in Figure 5a. Selecting the “Generate Test Data” sub-menu item triggers XDataPro to identify all the queries in the currently active file, and generate datasets and function parameter values for testing the correctness of functions containing queries. The generated datasets and parameter values are stored in a database, and loaded as required for unit tests. Users can also direct the plugin to consider only certain functions for testing, by using the annotation @TestDataGen. This annotation is used in Figure 5a for the function *getFreeVenues*.

For each function containing queries, the generated datasets are loaded one at a time, the function is executed on the generated parameter values, and the result is displayed to the user in the form of a user interaction window, as shown in Figure 5b. Figure 5b corresponds to a specific invocation of our plugin on the *SampleApp* class from Figure 5a. The window displays the function name (*SampleApp.getFreeVenues*), dataset id (DS1), function input parameter values (user_id:1234, date:2017-12-10), and the generated dataset, along with the output of running the function using these values.

The user is asked to mark if the function’s output matches the expected output for the given function inputs values and the dataset. Once all the datasets have been marked for a function, unit tests are generated for the function from a predefined template, using the function signature and details of the database containing generated datasets and parameter values. One sample unit test case generated for the function *getFreeVenues* is shown in Figure 5c These unit tests are added to the test suite for use in future regression testing.

V. Conclusion

Test cases for application testing usually focus on testing imperative code in the applications. XDataPro, on the other hand, focuses on testing correctness of SQL queries embedded in the application. Our framework can be used to complement the existing test cases so that both imperative code and database queries can be tested. Areas of future work include handling more SQL query mutations and suggesting correct queries based on the datasets.

References

[1] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan, “Extracting equivalent SQL from imperative code in database applications,” in *SIGMOD*, 2016.

[2] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan, “Data generation for testing and grading SQL queries,” *The VLDB Journal*, vol. 24, no. 6, 2015.

[3] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta, and D. Vira, “Generating test data for killing SQL mutants: A constraint-based approach,” in *ICDE*, 2011.

[4] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[5] C. Barrett and C. Tinelli, “CVC3,” in *Computer Aided Verification (CAV)*, 2007, pp. 298–302.

[6] K. Pan, X. Wu, and T. Xie, “Generating program inputs for database application testing,” in *ASE*, 2011, pp. 73–82.

[7] M. Emmi, R. Majumdar, and K. Sen, “Dynamic test input generation for database applications,” in *ISSTA*, 2007, pp. 151–162.

[8] M. Veanes, N. Tillmann, and J. de Halleux, “Qex: Symbolic SQL query explorer,” in *LPAR*, 2010, pp. 425–446.

[9] T. Sarkar, S. Basu, and J. Wong, “iConSMutate: Concolic testing of database applications using existing database states guided by SQL mutants,” in *ITNG*, 2014, pp. 479–484.

[10] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*. McGraw Hill, 6th ed., 2010.