

# SQL Data Generation to Enhance Search-Based System Testing

Andrea Arcuri

Kristiania University College, Oslo, Norway  
andrea.arcuri@kristiania.no

Juan P. Galeotti

Depto. de Computación, FCEyN-UBA, and ICC,  
CONICET-UBA. Argentina  
jgaleotti@dc.uba.ar

## ABSTRACT

Automated system test generation for web/enterprise systems requires either a sequence of actions on a GUI (e.g., clicking on HTML links), or direct HTTP calls when dealing with web services (e.g., REST and SOAP). However, web/enterprise systems do often interact with a database. To obtain higher coverage and find new faults, the state of the databases needs to be taken into account when generating white-box tests. In this work, we present a novel heuristic to enhance search-based software testing of web/enterprise systems, which takes into account the state of the accessed databases. Furthermore, we enable the generation of SQL data directly from the test cases. This is useful for when it is too difficult or time consuming to generate the right sequence of events to put the database in the right state. And it is also useful when dealing with databases that are “read-only” for the system under test, and the actual data is generated by other services. We implemented our technique as an extension of EvoMASTER, where system tests are generated in the JUnit format. Experiments on five RESTful APIs show that our novel technique improves code coverage significantly (up to +18%).

## CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Search-based software engineering**;

## KEYWORDS

SQL, database, SBST, automated test generation, system testing, REST, web service

### ACM Reference Format:

Andrea Arcuri and Juan P. Galeotti. 2019. SQL Data Generation to Enhance Search-Based System Testing. In *Genetic and Evolutionary Computation Conference (GECCO '19)*, July 13–17, 2019, Prague, Czech Republic. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3321707.3321732>

## 1 INTRODUCTION

Web and enterprise applications are very popular in industry. They can be very complex, which makes their automated *system testing* quite difficult. It is hence not uncommon that automated approaches only deal with *black-box* testing. Crawlers like Crawljax [30] are an example of this.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

GECCO '19, July 13–17, 2019, Prague, Czech Republic

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6111-8/19/07...\$15.00

<https://doi.org/10.1145/3321707.3321732>

Search-based testing tools like EvoMASTER [12] aim at generating *white-box* system tests, where the code of the system under test (SUT) is analyzed and instrumented. By using code information and white-box heuristics, such tools can generate better data to achieve higher code coverage and fault detection. The same type of search-based heuristics from *unit testing* tools like EvoSuite [22] are employed, like for example the *branch distance* popularly used in the search-based software testing literature [26]. In the case of JVM-based programs, this requires manipulating the bytecode of the SUT when it is loaded.

Instrumenting the SUT with search-based heuristics can give us gradient to generate more effective test data. However, web and enterprise applications often interact with external systems, where SQL databases are very common. The behavior of the SUT can depend on the current status of the database. For example, based on the results of a SQL SELECT query, different execution paths in the SUT could be taken. Such a SELECT query might return no data due to its WHERE clause not being satisfied. We might need to do some previous operations on the SUT (e.g., direct HTTP calls or clicks on a GUI) which lead to generate data in the database for which such WHERE clause would be then satisfied. However, bytecode heuristics on the SUT will not give us any gradient to guide the search to do previous SUT operations to put the right data into the database.

To overcome this issue, in this article we extend search-based software testing by defining SQL heuristics which can be integrated together with bytecode heuristics. The idea is to intercept every single SELECT query made by the SUT, and have new optimization targets aimed at having such queries returning non-empty sets of data. Such heuristic will be similar to the branch distance, but applied to the WHERE clauses of these queries. Such heuristic has to be integrated in the whole search, where a test case can be composed of many operations (e.g., several HTTP calls), and where there can be tens/hundreds of SQL queries for each test case.

However, populating the database with data that enables interesting application behaviour might not be that simple. For example, the application might be “read-only” (e.g., RESTful API with only GET endpoints), or the exact sequence of HTTP operations for populating the database could be too complicated to generate (e.g., a sequence of GET and POST operations in an specific order with certain parameter values). To handle these cases, we also enabled the insertion of SQL data directly from the test cases.

This introduces several research challenges, as now a test case is not only composed of operations on the SUT (e.g., HTTP calls and clicks on a GUI) but also SQL insertions directly executed to the underlying database. How to combine them both? How to drive the search to spend more/less time on the generation of SQL data instead of optimizing the SUT operations? For example, if a test case never executes any SQL SELECT, there is no point to generate SQL data as part of the search. If during the execution of a test

case such test case does SELECTs only from a specific table, then there would be no point in having the search generating data for the other tables that are never read during the test case execution.

We implemented our novel techniques as an extension of EvoMASTER [12], which is an open-source tool aimed at generating system tests for RESTful APIs. However, our techniques to handle SQL databases could be used also in other system testing contexts, like in GUI testing. Our tool extension is able to generate system tests in JUnit format. Those tests are self-contained (e.g., they can start and stop the SUT), and so can be run directly from an IDE (e.g., IntelliJ and Eclipse) or build system (e.g., Maven and Gradle). Experiments on five RESTful APIs show improvements of code coverage of up to +18%.

This article provides the following research and engineering contributions:

- We provide SQL heuristics that can be integrated into search-based test generation tools for system testing.
- We provide techniques to enable the direct generation of SQL data, and how they should be integrated with the regular search for the SUT's inputs.
- To enable the replicability of our experiments, our extensions to EvoMASTER are released as open-source software.

To the best of our knowledge, this is the first work in the literature in which *white-box*, *system tests* are automatically generated where a test case can be composed of both SUT's inputs (e.g., HTTP calls in our case study) and direct insertions into SQL databases.

## 2 MOTIVATING EXAMPLE

In this section, we will show a brief example of a Web API accessing a database. It is written in Java, using the SpringBoot framework [8], with the default Hibernate [2] as JPA provider. Figure 1 presents an excerpt of class `FooRest` and interface `FooRepository`. We do not show all the classes and needed configuration files, but just those we consider necessary to understand the presented example.

In this trivial example only two operations are allowed: POST and GET. The POST operation creates a new row in a database table, containing the columns *X* and *Y* with some fixed values (i.e., *X* = 42, and *Y* = 77). The primary key will be automatically handled by the `FooEntity` entity in a third, auto-increment column. The GET operation queries the database, trying to retrieve all the rows in such table with columns *X/Y* having values equal to what passed as path parameters in the URL. For example, a GET on URL `"/api/foo/2/3"` will search for all rows with *X* = 2 and *Y* = 3. If there is any row in the table satisfying that constraint, the GET will have status code 200, otherwise 400.

Note that in this example, there is no direct SQL query. The REST controller `FooRest` does autowire an instance of the interface `FooRepository`. When Spring autowires such bean, it will create a singleton instance, analyzing the name of the methods in such interface. For each such method, Spring Data will *automatically* create a concrete implementation doing a SQL command based on the name of the method itself. For example, `findByXIsAndYIs` will automatically generate code with the following SQL command:

```
SELECT foo0_.id, foo0_.x, foo0_.y
FROM   foo_entity foo0_
WHERE  foo0_.x=? and foo0_.y=?
```

```
@RestController
@RequestMapping(path = "/api/foo")
public class FooRest {
    @Autowired private FooRepository repo;
    @RequestMapping(method = RequestMethod.POST)
    public void post() {
        FooEntity entity = new FooEntity();
        entity.setX(42);
        entity.setY(77);
        repo.save(entity);
    }
    @RequestMapping(path =("/{x}/{y}",
        method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity get(
        @PathVariable("x") int x,
        @PathVariable("y") int y) {
        List<FooEntity> list = repo.findByXIsAndYIs(x, y);
        if (list.isEmpty())
            return ResponseEntity.status(400).build();
        else
            return ResponseEntity.status(200).build();
    }
}

public interface FooRepository
    extends CrudRepository<FooEntity, Long> {
    List<FooEntity> findByXIsAndYIs(Integer x,
        Integer y);
}
```

**Figure 1: Example of Web API in Java SpringBoot, with two endpoints (a GET and a POST) accessing a database via Spring Data.**

which will then be executed by Hibernate and the JDBC driver against the current database.

In such example, a bytecode heuristic on the decision `if(list.isEmpty())` provides no gradient, and it is a so called instance of the *flag problem* [17]. To obtain gradient to help the search, we need to analyze the heuristics on the WHERE clause of the SQL command, i.e., the `"WHERE foo0_.x=? and foo0_.y=?"`. Such result will depend on the state of the database (i.e., the content of the rows in that table), and will be computed internally inside the database. The JDBC driver will then just collect the result of such SELECT command. In this work, we propose a technique to intercept all these SELECT commands, and construct heuristics to analyze the WHERE clauses.

By extending EvoMASTER with our technique, the tool can generate tests like the one in Figure 2 very easily, as there is a direct gradient to find the right input parameters (i.e., 42 and 77) for the URL of the GET call. Without our heuristics, by default EvoMASTER would cover such data only at random, where each GET call will only have 1 possibility out of  $2^{32} \times 2^{32}$  to get the right data.

But let us consider a more challenging scenario in which only the GET endpoint is defined. Therefore, there is no POST operation

```
@Test
public void test1() throws Exception {
    given().accept("*/*")
        .post(baseUrlOfSut + "/api/foo")
        .then()
        .statusCode(200);

    given().accept("*/*")
        .get(baseUrlOfSut + "/api/foo/42/77")
        .then()
        .statusCode(200);
}
```

**Figure 2: Example of JUnit test generated for the Web API in Figure 1.**

```
@Test
public void test1() throws Exception {
    List<InsertionDto> insertions = sql()
        .insertInto("FOO_ENTITY", 8L)
        .d("X", "1177")
        .d("Y", "536")
        .dtos();
    controller.insertIntoDatabase(insertions);

    given().accept("*/*")
        .get(baseUrlOfSut + "/api/foo/1177/536")
        .then()
        .statusCode(200);
}
```

**Figure 3: Example of JUnit test generated for the Web API in Figure 1 where we have direct writing into the SQL database.**

that adds data into the database. In this scenario, the only way to achieve full coverage on the GET method is by injecting data directly to the database from the test cases. In our technique, we extend EvoMASTER to be able to generate SQL data, and have such data generation as part of the search, together with the generation of the inputs for the SUT (e.g., the GET call in this example). Figure 3 shows an example of JUnit test generated with our novel technique. Such test will first add the values 1177 and 536 into the database, and then do a GET with the same values. For doing the SQL insertion we wrote our own library with its own DSL, which is automatically integrated when the tests are generated.

We needed a DSL (and not writing the SQL directly as strings in the JUnit tests) to handle the case of foreign keys pointing to data generated in a previous insertion where the primary key is automatically generated by the database, as it would be unknown before the SQL queries are actually executed (this will be explained in more details in Section 6.2). Without our novel technique, it would be impossible for a tool like EvoMASTER to achieve full coverage by only relying on the available GET endpoint in the Web API.

### 3 BACKGROUND

#### 3.1 Structured Query Language (SQL)

The Structured Query Language (SQL) [9] is a domain specific language specially designed for retrieving and storing data in a relational database management system (RDMS). Each table  $T_i$  is composed of rows, i.e.,  $T_i = \{R_1, \dots, R_k\}$ . Each row contains an ordered list of values, i.e.,  $R_j = \langle V_1, \dots, V_c \rangle$ , such that the number of columns is fixed for all rows  $R_j$  in table  $T_i$ . In turn, each column can store values of a given SQL data type.

Additionally, more restrictive constraints on the data can be specified. If a *unique* constraint on a given column (or set of columns) is specified, the RDMS will not allow any insertion or update leading to a duplicated value in that specific column (or combination of values if the unique constraint is specified over more than one column).

A *primary key* on table  $T_i$  is a unique constraint that univocally identifies each row of table  $T_i$ . A table  $T_j$  can refer to values contained in the primary key of table  $T_i$  by specifying a *foreign key* on  $T_j$ . Both primary and foreign keys are specified as non-empty lists of columns of a table. If a foreign key exists, the database engine will forbid any insertion (or update) on  $T_j$  that might introduce a value (or combination of values) on the foreign key column (or columns) of  $T_j$  that is not present in the primary key column (or columns) of  $T_i$ . Similarly, the database engine does not allow any update or removal of rows in  $T_i$  that might lead to the same scenario. Custom constraints can also be specified through CHECK conditions (e.g., forcing all values stored in a column to be greater than or equal to a fixed value). The expressiveness of the allowed conditions depend on the specific SQL database.

SQL provides language constructs for inserting new rows (i.e., INSERT INTO commands), updating values in existing rows (i.e., UPDATE commands), removing rows (i.e., DELETE FROM commands). For retrieving rows from the database, SQL provides the SELECT FROM query. All operations can be filtered by means of a WHERE clause that specifies conditions that the rows must meet in order to apply the desired command. SQL also provides pre-defined functions that can be used to compute numerical function on the returned sets (e.g., count, avg, sum).

#### 3.2 EvoMaster

Among the different techniques proposed throughout the years, search-based software engineering has been particularly effective at solving many different kinds of software engineering problems [24], in particular software testing [10], with tools such as EvoSuite [22] for unit testing and Sapienz [27] for Android testing. EvoMASTER [12] aims at generating system level test cases for RESTful APIs [14]. It employs evolutionary algorithms, like for example MIO [13] and MOSA [32].

EvoMASTER is divided in two main parts: (1) a *core* process that is responsible for the command line interface, search algorithms, and generation of test cases; and (2) a *controller* library which is needed by the user to write manual configuration classes to tell EvoMASTER how to start, reset and stop the SUT. Such controller library is also responsible for automatically instrumenting the SUT when it starts to collect heuristics such as the branch distance.

EvoMASTER does output test cases in the JUnit format. These tests are “self-contained”, as they use the *controller* reference to start/stop/reset the SUT when needed through `@Before` and `@After` methods. These tests can be directly started from an IDE (like for example IntelliJ) or as part of a build system (e.g., Maven and Gradle). Each generated test will be a series of HTTP calls on the SUT, using the popular RestAssured [7] library.

## 4 RELATED WORK

EvoSuite [22] generates unit tests for Java classes. But Java methods might access a SQL database, and the generated test cases would fail if such databases are not automatically initialized. EvoSuite does have support to handle some parts of the JEE (Java Enterprise Edition) specification [16]. In particular, it can automatically start a H2 database and properly inject valid references to `EntityManager` objects used by JPA to access such database. However, no heuristic is provided on how to help the generation of inputs used in the database.

EvoSQL [20] is a search-based approach for generating database data to exercise SQL queries directly (i.e., not in the context of testing a SUT where it is the SUT that invokes the SQL queries). EvoSQL focuses on generating data that exercises all conditions in a SQL query following the coverage criteria defined as *full predicate coverage* by Tuya et al. [34]. In this work, since our primary goal is to increase code coverage in a SUT, we do not aim at achieving full predicate coverage of the SQL query, but to generate any data satisfying the WHERE clause. Other approaches also focus on generating SQL data such as QAGen [18] and ADUSA [25], but relying on a SAT solver instead of applying search-based heuristics.

Database schemas can have constraints on the data, typically primary and foreign keys. Furthermore, arbitrary constraints can be added to the columns of the tables via the CHECK keyword. Search-based tools like SchemaAnalyst [29] can generate JUnit tests that create SQL data optimizing different criteria on the coverage of the constraints in the schema [28] (but no SUT software involved).

As our approach, Emmi et al. [21] focuses on SQL data generation as a means to increase coverage of a SUT that accesses an off-the-shelf database engine. In order to do so, their technique collects constraints from both the SUT as well as from executed SQL queries. These constraints are later fed to a constraint solver to derive new test inputs. However, such technique can only work when the tool can identify which statements do execute SQL queries (e.g., when doing direct calls on JDBC methods from the `javax.sql` package), which can be problematic considering all the different kinds of libraries in Java that abstract from SQL commands (e.g., Hibernate [2] and JOOQ [4]). Similar work has been done by Fuchs and Kuchen [23], targeting unit testing of JEE applications where symbolic execution on the JPA interfaces is used to generate data in the database. In this article we also consider both code and SQL queries, but EvoMASTER relies on them to compute gradient for its search-based algorithm, not for applying constraint solving. As the work in [21, 23] was applied at the *unit* level, the scalability to *system* test generation still remains to be studied. On the contrary, EvoMASTER targets whole system test generation, specifically RESTful APIs, where each test case can be composed of several

HTTP calls, and with no particular limitation on which libraries are used to execute the SQL commands.

## 5 SQL HEURISTICS

When we test a SUT, there can be different criteria that we want to maximize for. For example, we might want to generate test cases that maximize statement coverage, branch coverage, mutation testing scores, fault detection, or two or more of such types of criteria at the same time [33]. These kinds of metrics can then be measured when test cases are run in an IDE or in a Continuous Integration server.

To achieve such goals, a typical approach in the literature of search-based software testing is to use the *branch distance* [26], which helps providing gradient to solve the constraints in the branch statements of the SUT’s code. To collect branch distances, the code of the SUT needs to be instrumented. However, these heuristics would not help us when the constraints are handled externally, like a SQL database.

During the execution of an *action* on the SUT (e.g., an HTTP call in a web service or a button click on a GUI application), the SUT might execute one or more SQL commands on a database. The execution paths on the SUT can depend on what is returned from these SQL commands. What returned from the SQL queries depends on the current data stored in the database, which could be modified by previous actions on the SUT (e.g., POST/PUT operations in a RESTful API), or other external services working on the same database.

In theory, it could be possible to have a complete analysis of how SQL queries are executed and their direct impact on the data-flow execution of the SUT. But whether such an approach would scale with current techniques/technologies remains to be seen. SUTs could be arbitrarily complex, using even more complex frameworks. For example, recall the snippet in Figure 1, where the execution of a SQL query depends on the name of a method of an interface, for which the Spring framework [8] creates a proxy class at runtime, autowired where such class is used. From where the method `findByXIsAndYIs(x, y)` is called and the actual SQL call, there can be hundreds of method calls before the actual SQL command is executed on a JDBC driver. And what is called depends on the used frameworks, like Spring [8] and Hibernate [2].

To overcome these problems, and be able to analyze the impact of the SQL queries regardless of which framework or library is used, in this work we propose the following approach:

- (1) We monitor all commands submitted to the SQL database at the JDBC driver level.
- (2) Every time the SUT does a SELECT operation on the database for which no data is returned, we compute an heuristic value to determine how far it was from returning some data.
- (3) Such heuristic values for each SELECT are then integrated in the search as secondary objectives to optimize.

The rest of this section discusses these three points in more details.

### 5.1 Monitoring SQL commands

For the JVM, there are many different libraries and frameworks to access a SQL database, such as: Hibernate [2], EclipseLink [1],

JDBI [3], JOOQ [4], etc. Those libraries help the writing of code dealing with databases, often abstracting from the SQL syntax. In the end, such libraries will execute actual SQL commands on the JDBC driver based on the SUT code. JDBC is the standard API in Java to access databases. It abstracts from the low level details of how to communicate with the different databases, like Postgres, MySQL, H2, Derby, etc.

When the SUT executes, it requires the specific JDBC driver implementation for the target database. In order to monitor all calls performed on a JDBC driver regardless of which is the specific SQL library the SUT uses, we employ P6Spy [6]. P6Spy is a special JDBC driver that wraps an existing driver, and can log all events on it.

Adding P6Spy to an existing SUT is a rather straightforward task. It just needs to be included as a third-party library (e.g., as a Maven/Gradle dependency), and the SUT must be set to include the P6Spy driver. In many frameworks like Spring [8], this can be easily set with input parameters without needing to change the source code. For example, a Spring application could be started with the option `--spring.datasource.url=jdbc:p6spy:h2:mem`, which would override the existing `jdbc:h2:mem` datasource URL by wrapping it in P6Spy.

## 5.2 SQL Distance

Each time the SUT executes a SELECT operation that returns no data, we intercept such call to compute a heuristic distance without altering the behaviour of the running SUT. We repeat each such SELECT *without* any WHERE clause, and collect a result set  $S$ . On such set, on each row  $r$ , we compute a distance measure  $d_r$  to check how far such data row was heuristically from being able to satisfy the predicate in the WHERE clause. Then, the *minimum*  $d_r$  on  $S$  is what we report as heuristic distance for the executed SELECT.

To collect  $S$  to compute  $d_r$ , it is not enough to repeat the SELECT query with no WHERE clause. We might need to apply further transformations. For example, consider the SQL query:

```
SELECT f.x FROM Foo f WHERE f.y=42
```

Here, the predicate is based on a column  $y$  which is not among the fields returned by the SELECT. So, we need to modify the SELECT to also collect the columns mentioned in the WHERE predicate. Therefore, the transformed query will be:

```
SELECT f.x, f.y FROM Foo f
```

However, this might not be enough. Consider this:

```
SELECT count(*) FROM Foo f WHERE f.y=42
```

Here, the query would just return a number, and not a set  $S$ . So, we need to remove all the numerical functions on the returned sets, such as: count, avg, sum, etc. The transformed query will hence be:

```
SELECT * FROM Foo f
```

Finally, once the SELECT is transformed and re-executed to collect  $S$ , we can compute the heuristic distance. The distance computation on the WHERE predicate we employ is similar to the *branch distance* [26] in source code, and it is inspired from existing work on solving OCL [11] and SQL [20] constraints. For example, a constraint like  $x == 0$  would have a distance  $d(x) = \text{abs}(x-0)$ , whereas  $A$  and  $B$  would have  $d(A, B) = d(A) + d(B)$ , while for  $A$  or  $B$  it would

be  $d(A, B) = \min(d(A), d(B))$ . We support all the boolean operations in SQL, including predicates on strings and collections.

## 5.3 SQL Fitness Function

Once we can collect an heuristic distance for each executed SELECT with no returned data, we need to use such distances to improve the search. Albeit there is existing work on how to define effective distance functions on SQL predicates, how to integrate such distances into the search for system test generation poses many research challenges.

First and foremost, our final goal is to improve code coverage on the SUT. Having a SELECT operation returning non-empty data sets is something that likely will improve code coverage, but it is not our main goal: it is just a *secondary* objective we used under the assumption that it will be beneficial to achieve the primary objective (i.e., increasing code coverage of the SUT). Furthermore, when trying to cover a target goal  $g$  in the SUT, many SELECT operations could be executed, and not all of them are relevant for that specific target goal  $g$ .

In EvoMASTER, for each target goal  $g$  (e.g., a branch, a statement or a HTTP status code) there is a heuristic value  $h(g) \in [0, 1]$ , where  $h(g) = 0$  means the target goal is not reached, and  $h(g) = 1$  means the target goal is covered. Values in between give heuristic gradient to guide the search. During the execution of a test case composed of  $N$  actions (e.g.,  $N$  HTTP calls toward a RESTful API), a certain target goal  $g$  could be reached several times, for which different heuristic values are computed. To guide the search, EvoMASTER keeps track of the best  $h$  values during the execution of a test, and that is the one used in the fitness function.

To enhance EvoMASTER to handle SQL queries, we extend its fitness function to handle *secondary* objectives. Besides a primary heuristics  $h(g)$  for each target goal  $g$  (e.g., branches and statements in the SUT), each such target goal will also have a secondary value  $s(g)$ . When two individuals have the *same* primary  $h(g)$  value, then the fitness function will look at the secondary objective  $s(g)$  to decide which one of the two individuals is most fit for reproduction.

The secondary  $s(g)$  is a value to minimize, and it is computed as follows:

- Keep track of the action  $i$  in  $N$  which led to the best value for  $h(g)$ .
- Collect the distances  $d(r)$  for *all* the SELECT queries with empty return-sets done *only* at action  $i$  of  $N$ . Let us call this group of distances  $D_i$ .
- Compute the average of all these distances in a single value, i.e.,  $s(g) = \sum_{d_r \in D_i} \frac{d_r}{|D_i|}$ . If  $D_i$  is empty, then  $s(g)$  gets the worst possible value.

So, by using  $s(g)$  as a secondary objective to minimize, we are rewarding test cases that get closer to have SELECT operations returning non-empty data sets. However, this is just a secondary objective, as we cannot be sure that having data returned by the database will necessarily have a beneficial impact on achieving coverage of  $g$ .



## 6 SQL GENERATION

### 6.1 Generation During the Search

To cover a specific testing goal  $g$  (e.g., a branch in the SUT), we might need the database to be in a certain state, as such goal might be covered only if a SELECT query returns some specific data. Therefore, in order to cover  $g$ , we might need to take several actions on the SUT to put the database in the required state.

This approach is feasible, but it has two potential problems:

- (1) In order to store the required data in the database, we might need to execute *many* actions on the SUT. Not only this makes the search more difficult, but it could also negatively affect test *readability*. For example, it might be hard to understand the output of a single action if the action first requires another 30 actions before it to set up the database.
- (2) It might be impossible to insert the required data into the database using the available actions. In other words, from the perspective of the SUT, parts of the database are “read-only”. This is the case when a database is accessed by more than one system besides the SUT.

To overcome these problems, we extend the search in EVOMASTER by adding the possibility to write data directly into the database, and have such data optimized as part of the search. Adding data will be done by executing *only* INSERT operations on the database.

The first issue is that we do not want to increase the search space when it is unnecessary. If a test case does not execute any SELECT on the database, then there is no point in generating SQL data directly. Furthermore, if a test case only access a SQL table called  $X$ , then we only need to create data to add to  $X$ , and not for the other tables in the database. To achieve these goals, SQL actions are only added as an “initialization” step before the main actions on the SUT (e.g., HTTP calls). Once a test case is executed, we analyze which tables (if any) have been accessed. If after the mutation and re-execution of a test case new tables are accessed, then new SQL actions (randomly between 1 and  $n$ , where for example we could have  $n = 5$ ) are added to the initialization phase for these tables. If a table  $X$  has a non-null foreign key for a different table  $Y$ , we need to insert data also for  $Y$  before inserting data for  $X$ . This might need to be done recursively, if  $Y$  itself has non-null foreign keys toward other tables. In this manner, SQL actions are only added as initialization and they are not intertwined with actions on the SUT.

A test case is now composed of zero or more initializing SQL actions on the database, followed by the actual actions on the SUT (e.g., HTTP calls or GUI clicks). For example, recall Figure 3. During the search, the content of each action can be mutated (e.g., the values in the SQL INSERTs and the HTTP calls). However, actions will be added and removed as part of the search only for the main actions on the SUT, and not for the SQL initializing ones, as these latter are automatically added/removed when needed.

Mutating values in the SQL actions follows the same rules of mutating values in the SUT actions. For example, mutating a number or a string does follow the same algorithms used in EVOMASTER to mutate such kind of values when present for example in URL query parameters and HTTP body payloads in JSON. However, there are some SQL types that need to be handled specially. For example, in a list of SQL INSERTs, we need to guarantee that all primary keys in

a table are *unique*. Furthermore, foreign keys must point to existing data inserted in previous INSERT operations in such a list.

The goal of the SQL actions is to insert the desired data into the database before the actual SUT execution. As INSERT operations would just fail in case of submitting invalid data (e.g., a string value on an integer column), we need to guarantee that the list of initializing SQL actions is valid. After sampling and mutation, but before fitness evaluation, we execute a *repair* phase in which we try to fix any broken constraint (e.g., unique primary keys and non-null foreign keys pointing to existing rows). This is not always trivial. Let us consider two tables *Foo* and *Bar* such that the primary key *ID* in table *Bar* is also a foreign key for the table *Foo*. Now, consider the following sequence of SQL commands:

```
INSERT INTO Foo ID values (1);
INSERT INTO Bar ID values (1);
INSERT INTO Bar ID values (1);
```

Here, as cannot have the same primary key used twice in a table, the third insertion would hence fail. However, no other value is possible, because there is only one single row in *Foo*. Here, to fix this constraint, we either must remove the last INSERT on *Bar*, or create a new INSERT on *Foo* with different primary key (e.g., 2), and have the last INSERT on *Bar* using its ID.

Our approach to SQL action repair works as follows: given a list of initializing SQL actions  $a_0, \dots, a_n$  where each  $a_i$  corresponds to a INSERT command on a specific table in the database, we start by finding the smallest index  $i$  such that  $a_i$  attempts to:

- insert a value that violates a unique constraint or primary key constraint, or
- insert a value that does not satisfy a foreign key constraint.

Then, the value is randomized in an attempt to satisfy the constraint that was not met. For foreign keys a new value is randomly selected among those values already observed in the preceding actions. If the new value satisfies all constraints, the repair continues to the following action  $a_j$ , with  $j > i$  such that any of the above conditions are found.

The process is repeated until either all problematic SQL actions are repaired, or a  $K \geq 0$  bound on the number of attempts to repair  $a_i$  is reached. As shown in the above example, in some cases no new value exists for a prefix sequence of initialization actions. For example, consider three consecutive INSERT actions attempting to append boolean values to a non-null unique column. It is easy to see that no third value could be used to repair the third INSERT action as both true and false were already inserted. In other cases, although constraints are indeed satisfiable, the randomization search might fail to deliver a new suitable value satisfying the constraints due to its probabilistic behaviour. While in the former scenario it is not possible to recover, in the latter this could be mitigated by increasing the value of  $K$ . For the evaluation in this article, we have fixed  $K = 5$  after some preliminary experiences with the experimental subjects.

After  $K \geq 0$  unsuccessful tries to repair action  $a_i$ , the list of initialization SQL actions is truncated by removing all SQL actions  $a_j$  such that  $j \geq i$ . This guarantees that the resulting list of initialization SQL actions  $a_0, \dots, a_{i-1}$  will not invalidate any of the aforementioned constraints.

```

List<InsertionDto> insertions = sql()
    .insertInto("Foo", 1L)
    .d("X", "42")
    .and().insertInto("Bar", 2L)
    .d("Y", "123")
    .r("ID", 1L)
    .dtos();
controller.insertIntoDatabase(insertions);

```

**Figure 4: Example of SQL INSERTs where a foreign key points to an auto-increment primary key added in a previous insertion.**

## 6.2 Insertions In Output Tests

Once the search is finished, EvoMASTER does output a class file with JUnit tests. To be able to execute the same kind of SQL INSERTs done during the search, we had to extend EvoMASTER to handle SQL commands directly from the generated tests.

To be able to connect to the database used by the SUT, we need to get a reference to the used JDBC driver. Therefore, we extended the *controller* classes in EvoMASTER to get a reference to the driver. In frameworks like Spring, this is as easy as calling `getBean(JdbcTemplate.class)` on the `ConfigurableApplicationContext` object starting the SUT, and then calling `getDataSource().getConnection()` on such `JdbcTemplate` reference.

When generating JUnit files, SQL commands could be added directly as strings. However, this approach is limited. For example, a table *Foo* might have a primary key with an auto-increment value. The user would not choose such value, as it would be automatically created by the database when a new row in such table is added. The actual used id will depend on the state of the employed “sequence” generator in the database, and how it operates. If then a second INSERT on a different table *Bar* has a foreign key pointing to *Foo*, then it would not be feasible for us to know for certain such primary key in advance when the JUnit files are generated.

To solve this issue, our solution is to write our own DSL (domain-specific-language). When an INSERT needs to handle a foreign key pointing to an auto-increment primary key, in the DSL we will just point a reference to that insertion operation. Each INSERT in the list of SQL initializing actions will be executed one at a time, collecting the values of each generated auto-increment primary key. When we need to insert a reference foreign key, we will use the actual valid values collected in the previous insertions.

Consider the example in Figure 4. There, two INSERTs are executed. In the first one, a new row is added for table *Foo* with value 42 for the column *X*. This is done by using a call on `d(column, value)` (where *d* stands for “data”). The primary key *ID* is automatically generated by the database, as being an auto-increment one. The second INSERT adds a new row into the table *Bar*, where the column *Y* gets the value 123. The primary key for *Bar* is also a foreign key pointing to *Foo*. But, when this code is generated and outputted in a JUnit file, the value of the primary key in *Foo* is unknown. Therefore, we use the call `r(column, insertId)` (where *r* stands for “reference”). Such call tells the runtime to use the value

**Table 1: RESTful web services used in the empirical study. We report number of Java/Kotlin classes, lines of code (LOC), number of Endpoints, total number of tables and columns in the underlying database.**

Name	Classes	LOC	Endpoints	Tables	Columns
<i>catwatch</i>	69	5442	23	5	45
<i>features-service</i>	23	1247	18	6	20
<i>proxyprint</i>	68	7534	74	15	92
<i>rest-news</i>	10	718	7	1	4
<i>scout-api</i>	75	7479	49	14	70
Total	245	22420	171	41	231

of the primary key in a previous INSERT operation with `insertId` (1L in this case). A dto (data-transfer-object) is generated out of this DSL, and then sent to our running environment to execute those SQL INSERTs on the database, one at a time.

## 7 EMPIRICAL STUDY

In this article, we have carried out an empirical study aimed at answering the following research questions.

**RQ1:** How much code coverage improvement can our novel SQL heuristics achieve in the generation of system-level test cases?

**RQ2:** Does generating SQL data directly improve performance of the search?

### 7.1 Artifact Selection

In our experiments, we used the same case study from previous work on EvoMASTER [13, 14], but only considering the SUTs accessing SQL databases. This includes five different RESTful APIs written in Java and Kotlin, with EvoMASTER controllers to start/reset/stop those SUTs. We just needed to modify these controllers to enable accessing the JDBC drivers used to connect to the databases.

Data about these five RESTful web services is summarized in Table 1. Those five RESTful web services contain up to 7500 lines of codes (tests excluded). This is a typical size for a RESTful API, especially in a microservice architecture [31]. The reason is that, to avoid the issues of monolithic applications, such services usually become split if growing too large, as to make them manageable by a single, small team. This, however, does also imply that enterprise applications can end up being composed of hundreds of different services. Each SUT does access a SQL database (H2 or Derby), with up to 15 tables and 92 columns (for the *proxyprint* SUT).

### 7.2 Experiment Settings

On each of the five SUTs, we ran EvoMASTER 30 times, with the search budget of 10k HTTP calls. We considered three different configuration settings. In total, we had  $5 \times 30 \times 3 = 450$  independent searches, for a total of  $450 \times 10,000 = 4.5m$  HTTP calls.

The configurations we considered were: **Base**, the default version of EvoMASTER, with none of our novel techniques presented in this article. **SQL Heuristics (H)**: default EvoMASTER, with the fitness function extended with the secondary objectives described in Section 5. **SQL Heuristics & Generation (G+H)**: version in which we enable the SQL Heuristics and generating SQL data directly, as discussed in Section 6.

**Table 2: Average coverage for Base, SQL Heuristics (H) and SQL Generation (G+H) configurations. Effect-size  $\hat{A}_{12}$  are reported for all configuration pair combinations. Values in bold when U-tests  $p$ -value  $\leq 0.05$ .**

SUT	Base	H	G+H	$\hat{A}_{hb}$	$\hat{A}_{gb}$	$\hat{A}_{gh}$
<i>catwatch</i>	967.3	967.4	1149.0	0.52	<b>1.00</b>	<b>1.00</b>
<i>features-service</i>	597.1	614.1	655.7	0.61	<b>0.98</b>	<b>0.91</b>
<i>proxyprint</i>	1469.3	1486.0	1481.7	<b>0.77</b>	<b>0.77</b>	0.52
<i>rest-news</i>	279.1	278.2	305.2	0.50	<b>0.96</b>	<b>0.96</b>
<i>scout-api</i>	1468.9	1420.0	1442.5	0.43	0.46	0.53

We did not use the number of fitness evaluations as stopping criterion, because each test can have a different number of HTTP calls. Considering that each HTTP call requires sending data over a TCP connection (plus the SUT that could write/read data from a database), their cost is not negligible (even if still in the order of milliseconds when both EvoMASTER and the SUT run on the same machine). As the cost of running a system test is much, much larger than the overhead of the search algorithm code in EvoMASTER, we did not use time as stopping criterion. This will help future comparisons and replications of these experiments, especially when run on different hardware. Tracking SQL queries does add a computational overhead though. However, when looking at the execution times of the experiments running on a cluster, such overhead was negligible.

### 7.3 Experiment Results

Table 2 shows the results of the experiments on the three different settings for EvoMASTER. Note that EvoMASTER does optimize for several different testing criteria at the same time, such as statement coverage, branch coverage and HTTP status coverage. It returns a total number of covered targets, and not a percentage.

To analyze the results, we followed the guidelines from [15]. Not only we computed average values out of 30 runs, but also computed Wilcoxon-Mann-Whitney U-tests (at  $\alpha = 0.05$  significance level) and Vargha-Delaney  $\hat{A}_{12}$  effect sizes.

Regarding the SQL Heuristics, it seems to provide strong improvement only for one SUT (i.e., *proxyprint*), with  $\hat{A}_{hb} = 0.77$  and  $p$ -value smaller than 0.05. The improvement is statistical significant, but relatively low in terms of code coverage improvements (i.e., from 1469.3 to 1486.0). Such heuristic seems to provide some benefits on two other SUTs, and worse results in another SUT. However, in these three latter cases, the 30 runs were not enough to provide strong enough statistical evidence.

**RQ1:** *On its own, the SQL Heuristics provide only limited improvements to the achieved coverage.*

When generating SQL data directly, results are much stronger. Improvements are statistically significant on three SUTs, with very strong effect-sizes ( $\hat{A}_{12} \geq 0.91$ ). An effect-size like 1.00 for *catwatch* means that, in every single of the 30 runs we got better results than any of the other 30 runs without SQL generation. This is the strongest possible achievable  $\hat{A}_{12}$  effect-size. In terms of raw coverage improvement, it is a  $\frac{1149.0 - 967.3}{967.3} = +18.7\%$  improvement.

**RQ2:** *When generating SQL data directly, strong improvements are obtained, even up to +18.7% coverage.*

## 8 THREATS TO VALIDITY

Threats to internal validity come from the fact that our empirical study is based on an extension to the EvoMASTER tool. Faults in such extension might compromise the validity of our conclusions. Although we have carefully tested our implementation, we cannot provide a guarantee that it is bug-free. However, to mitigate such risk, and enable independent replication of our experiments, our extension is freely available as open-source.

As our techniques are based on randomized algorithms, such randomness might affect the results. To mitigate such problem, each experiment was repeated 30 times with different random seeds, and the appropriate statistical tests were used to analyse the results.

Threats to external validity come from the fact that only five RESTful web services were used in the empirical study. This was due to the difficulty of finding this kind of applications among open-source projects. As EvoMASTER controllers need to be manually written for each SUT, we simply re-used the existing configured SUTs provided by EvoMASTER for experimentation. Although those five services are not trivial (i.e., up to 7500 lines of code), and heavily dependent on SQL databases (i.e., up to 15 tables and 92 columns), we cannot currently generalize our results to other web services and other kinds of system testing (e.g., for GUI applications).

Our approach is only compared to the default version of EvoMASTER. But it could also be applied to other search-based tools.

## 9 CONCLUSION

We have presented a novel search-based approach in which we enhance automated system test generation by handling SQL databases. Not only we proposed a novel approach to introduce heuristics on SQL queries as secondary objectives to optimize, but we also added generating SQL data as part of the search.

We implemented our novel approach as an extension of EvoMASTER. The generated JUnit tests can have an initialization phase in which SQL data is automatically inserted into the database before any command is executed on the system under test. Experiments on five different RESTful APIs show that our novel techniques can improve coverage significantly, up to a +18% improvement.

To the best of our knowledge, this is the first work in the literature in which initializing SQL data for system testing is generated automatically. Future work will investigate other approaches to generate such type of SQL data, like for example by using Dynamic Symbolic Execution [19]. Furthermore, it will also be important to support other kinds of databases, like NoSQL (e.g., MongoDB [5]).

EvoMASTER is freely available online as open-source, accessible at <https://www.evomaster.org>.

## ACKNOWLEDGMENTS

This work is funded by the Research Council of Norway (project on Evolutionary Enterprise Testing, grant agreement No 274385), and partially by UBACYT-2018 20020170200249BA, PICT-2015-2741.



## REFERENCES

- [1] [n. d.]. EclipseLink. <http://www.eclipse.org/eclipselink/>. ([n. d.]).
- [2] [n. d.]. Hibernate. <http://hibernate.org/>. ([n. d.]).
- [3] [n. d.]. JDBC. <http://jdbc.org/>. ([n. d.]).
- [4] [n. d.]. JOOQ. <https://www.jooq.org/>. ([n. d.]).
- [5] [n. d.]. MongoDB. <https://www.mongodb.com/>. ([n. d.]).
- [6] [n. d.]. P6Spy. <https://github.com/p6spy/p6spy>. ([n. d.]).
- [7] [n. d.]. RestAssured. <https://github.com/rest-assured/rest-assured>. ([n. d.]).
- [8] [n. d.]. Spring Framework. <https://spring.io/>. ([n. d.]).
- [9] [n. d.]. SQL. <https://www.iso.org/standard/63555.html>. ([n. d.]).
- [10] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)* 36, 6 (2010), 742--762.
- [11] S. Ali, M. Z. Iqbal, A. Arcuri, and L.C. Briand. 2013. Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering (TSE)* 39, 10 (2013), 1376--1402.
- [12] A. Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 394--397.
- [13] A. Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology (IST)* (2018).
- [14] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
- [15] A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* 24, 3 (2014), 219--250.
- [16] A. Arcuri and G. Fraser. 2016. Java enterprise edition support in search-based junit test generation. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 3--17.
- [17] A. Baresel and H. Sthamer. 2003. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation Conference (GECCO)*. 2442--2454.
- [18] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *ACM SIGMOD international conference on Management of data*. ACM, 341--352.
- [19] C. Cadar and K. Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82--90.
- [20] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen. 2018. Search-based test data generation for SQL queries. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 1230--1230.
- [21] M. Emami, R. Majumdar, and K. Sen. 2007. Dynamic test input generation for database applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 151--162.
- [22] G. Fraser and A. Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*. 416--419.
- [23] A. Fuchs and H. Kuchen. 2017. Unit testing of database-driven Java enterprise edition applications. In *International Conference on Tests and Proofs*. Springer, 59--76.
- [24] M. Harman, S. A. Mansouri, and Y. Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.
- [25] S. Khalek, B. Elkarablieh, Y. Laleye, and S. Khurshid. 2008. Query-aware test generation using a relational constraint solver. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE, 238--247.
- [26] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* (1990), 870--879.
- [27] K. Mao, M. Harman, and Y. Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 94--105.
- [28] P. McMinn, C. J. Wright, and G. M. Kapfhammer. 2015. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 8.
- [29] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer. 2016. SchemaAnalyst: Search-based test data generation for relational database schemas. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 586--590.
- [30] A. Mesbah, A. Van Deursen, and S. Lenselink. 2012. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)* 6, 1 (2012), 3.
- [31] S. Newman. 2015. *Building Microservices*. "O'Reilly Media, Inc."
- [32] A. Panichella, F. Kifetew, and P. Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering (TSE)* 44, 2 (2018), 122--158.
- [33] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 93--108.
- [34] J. Tuya, M. J. Suárez-Cabal, and C. De La Riva. 2010. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability (STVR)* 20, 3 (2010), 237--288.