

Handling SQL Databases in Automated System Test Generation

ANDREA ARCURI, Kristiania University College, Oslo, Norway

JUAN P. GALEOTTI, Depto. de Computación, FCEyN-UBA, and ICC, CONICET-UBA. Argentina

Automated system test generation for web/enterprise systems requires either a sequence of actions on a GUI (e.g., clicking on HTML links and form buttons), or direct HTTP calls when dealing with web services (e.g., REST and SOAP). When doing *white-box* testing of such systems, their code can be analyzed, and the same type of heuristics (e.g., the *branch distance*) used in search-based unit testing can be employed to improve performance. 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 when it is too difficult or time consuming to generate the right sequence of events to put the database in the right state. Also, it is 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 six RESTful APIs (five open-source, and one industrial) show that our novel techniques improve coverage significantly (up to +16.5%), finding 7 new faults in those systems.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; **Search-based software engineering**.

Additional Key Words and Phrases: SQL, database, SBST, automated test generation, system testing, REST, web service

ACM Reference Format:

Andrea Arcuri and Juan P. Galeotti. 2019. Handling SQL Databases in Automated System Test Generation. 1, 1 (April 2019), 31 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Web and enterprise applications are very popular in industry. They can be very complex, which makes their automated *system testing* quite difficult. Hence, it is common that automated approaches only deal with *black-box* testing. Crawlers like Crawljax [43] are an example of this.

Search-based testing tools like EvoMASTER [15] 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 [26] are employed, like for example the *branch distance* popularly used in the search-based software

Authors' addresses: 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.

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.

© 2019 Association for Computing Machinery.

XXXX-XXXX/2019/4-ART \$15.00

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

testing literature [36]. 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 the generation of data in the database for which this WHERE clause would be then satisfied. However, bytecode heuristics (e.g., [38]) 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. This heuristic will be similar to the branch distance, but applied to the WHERE clauses of these queries. This 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.

Our SQL heuristics can guide the search to generate the right sequence of inputs on the SUT. For example, in a test case for a RESTful API with a HTTP POST followed by a HTTP GET, it can provide us gradient to generate the right data (e.g., URL query parameters and HTTP body payload) in the POST (data that will then be saved into the database). Such data will then be read by the following GET via SQL SELECTs. This approach works as long as the SUT provides us operations which can lead to the generation of data into the database. For example, in a RESTful API those would be HTTP endpoints such as POST, PUT and PATCH.

However, populating the database with data that enables interesting application behaviour might not be that simple. For example, the database might be “read-only” from the point of view of the application (e.g., a 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 should these be combined? How should the search be driven 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 [15], which is an open-source tool aimed at generating system tests for RESTful APIs. However, our approach 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) as part of Continuous Integration. EvoMASTER optimizes for several criteria, including white-box (e.g., source lines and bytecode-level branches) and black-box (e.g., HTTP status codes) ones. Experiments on six RESTful APIs show improvements of *target*

coverage (which includes all those criteria aggregated) of up to +16.5%, finding 7 new faults in those APIs.

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.

This work is an extension of a conference paper [20]. We have extended such work in several directions, like for example: handling of several strategies to aggregate the SQL heuristics (Section 5.3); handling of test length bloat (Section 5.4); supporting regex genes (Section 6.5); studying different strategies for the mutation rate (Section 6.4); answering six research questions instead of just two (Section 7); extending the case study with an industrial API (Section 7.1).

To the best of our knowledge, this paper (which extends [20]) 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 inputs to the SUT (e.g., HTTP calls in our case study) and direct insertions into SQL databases.

The paper is organized as follows. Section 2 starts by presenting and analyzing a motivating example. Background information to better understand the remanding of the paper is provided in Section 3. Related work is discussed and compared in Section 4. How we integrate our SQL heuristics during the search is presented in Section 5, whereas Section 6 discusses how we generate SQL data directly into the database. Our empirical study to evaluate those novel techniques follows in Section 7. Threats to the validity of such study follow in Section 8. Finally, Section 9 concludes the paper.

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 [9], with the default Hibernate [3] 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 `FooRepository`. 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_
```

```

@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);
}

```

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

```
WHERE foo0_.x=? and foo0_.y=?
```

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

In the above example, a bytecode heuristic on the decision `if(list.isEmpty())` provides no gradient, and it is a so called instance of the *flag problem* [21]. 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. Note that EvoMASTER uses the library RestAssured [8] to do the HTTP calls against the SUT. Also, it uses @Before and @After methods to start/stop/reset the SUT at each test execution via a reference to an object called controller. The reset method also must clean the state of the databases (if any is present). To simplify such

```

@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);
}

```

Fig. 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);
}

```

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

operation, EvoMASTER provides a library in which such database reset can be easily done with a single command. Resetting the state of the application, and databases, at each test execution is paramount, as each test case must be independent from each other.

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.¹ However, note that this is a simple example just to show the issues with the lack of gradient in the fitness function. Other techniques like *seeding* [47] from bytecode constants would likely be as effective here.

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

¹ A primitive Java integer value is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31} - 1$. For more information, refer to the Java Language Specification [31].

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. This test will first add the values 1177 and 536 into the database, and then do a GET with the same values.

But how were these values 1177 and 536 chosen? When sampling new individuals, their genes are set at random. Throughout the search, these genes will be modified with mutation. The fitness function provides gradient to reward mutations that lead the numbers to be the same in both the SQL insertions and URL path parameters.

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 6.6). 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. Once our technique was implemented, EvoMASTER successfully generates JUnit tests that cover all branches in FooRest, even when the POST endpoint is disabled. The JUnit tests that we generate can then be executed by any IDE, like for example IntelliJ.

3 BACKGROUND

3.1 Structured Query Language (SQL)

The Structured Query Language (SQL) [10] is a domain specific language specially designed for retrieving and storing data in a relational database management system (RDMS). In RDMSs, data is stored in *tables*. The SQL language includes constructs for creating a new table, and altering or removing existing tables. 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. Although data types might vary across different SQL databases (e.g., Oracle databases do not support the TIME data type), common data types such as INTEGER or VARCHAR are usually supported. The blueprint of the database instance (i.e., the tables with their corresponding columns and data types) is specified in the database *schema*.

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.

Table 1. Branch distance for relational operands. a and b are numeric values, whereas x and y are boolean predicates/expressions.

Relational operation	Branch distance ρ
$a > b$	ρ : if $a > b$ then 0 else $(b - a)$
$a \geq b$	ρ : if $a \geq b$ then 0 else $(b - a) + K$
$a < b$	ρ : if $a < b$ then 0 else $(a - b) + K$
$a \leq b$	ρ : if $a \leq b$ then 0 else $(a - b)$
$a = b$	ρ : if $a = b$ then 0 else $abs(a - b)$
$x y$	ρ : $\min(\rho(x), \rho(y))$
$x \& \& y$	ρ : $\rho(x) + \rho(y)$

Among the supported SQL data types, some databases allow integer columns to automatically generate their values whenever a new insertion is performed. This alleviates choosing values satisfying unique (and primary key) constraints including integer columns.

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 functions on the returned sets (e.g., count, avg, sum).

3.2 Search-Based Software Testing

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 [33], in particular software testing [11], with tools such as EvoSuite [26] for unit testing and Sapienz [39] for Android testing.

Software testing can be modeled as an optimization problem, where one wants to maximize the code coverage and fault detection of the generated test suites. Then, once a fitness function is defined for a given testing problem, a search algorithm can be employed to explore the space of all possible solutions (test cases in this context).

There are several kinds of search algorithms, where Genetic Algorithms are perhaps the most popular. In the particular case of generating test suites, there are different specialized search algorithms, such as for example Whole Test Suite (WTS) [28], MOSA [45] and MIO [16].

The branch distance is a common heuristic to guide the search for input data to solve the constraints in the logical predicates of the branches [36, 40]. The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules as described in Table 1. For example, for predicate $x \geq 100$ and x having the value 42, the branch distance to the true branch is $100 - 42 + k$, with $k > 0$. In practice, to determine the branch distance each predicate has to be instrumented to keep track of the distances for each execution. For more details regarding branch distances the reader is referred to [36, 40].

3.3 EvoMaster

EvoMASTER [15] aims at generating system level test cases for RESTful APIs [17]. It employs evolutionary algorithms, like for example MIO [16] and MOSA [45].

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

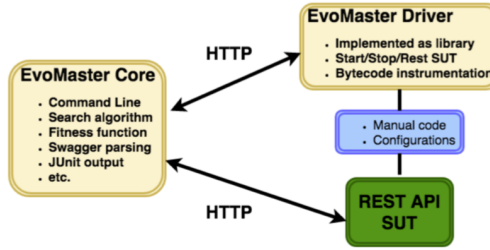


Fig. 4. High level architecture of EvoMASTER.

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. Figure 4 shows a high level overview of EvoMASTER’s architecture.

To detect faults, EvoMASTER checks the returned status codes from the HTTP calls, using a 5xx code as a representative for a fault. For example, when there is an exception in the SUT due to a bug (e.g., a null-pointer exception), a framework like Spring would not crash the whole server, and rather would return a HTTP response with 500 as status code. To distinguish between different bugs in the same endpoint, EvoMASTER uses the heuristic of keeping track of the last executed statement in the core classes of the SUT (not of the third-party libraries, like Spring and Tomcat) when a 5xx code is returned.

EvoMASTER outputs 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 [8] library. EvoMASTER is open-source, and available at <https://www.evomaster.org>.

4 RELATED WORK

EvoSuite [26] 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 [19]. 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.

The work in [13] proposes a fitness function to measure how close a string is to match a given regular expression. In this work we not only measure how close is a string to match a pattern, we also introduce new specific search operators for such inputs.

EvoSQL [24] 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. [49]. In this work, since our primary goal is to increase *target* 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 [22], ADUSA [34] and the work in [30], but relying on a SAT/SMT solver instead of applying search-based heuristics. However, lack of support of

strings, string operations, date/time functions and other JOIN expressions, subqueries, and specific database procedures have hampered the applicability of constraint-based data generators [24].

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 [42] can generate JUnit tests that create SQL data optimizing different criteria on the coverage of the constraints in the schema [41] (but no SUT software involved).

As our approach, Emmi et al. [25] 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 [3] and JOOQ [5]). Similar work has been done by Fuchs and Kuchen [29], 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 [25, 29] 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.

How a SUT interacts with a SQL database can have major security implications. Different testing techniques have been investigated to find SQL Injection vulnerabilities [35, 48].

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 [46]. 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* [36], which helps provides 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 in 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 is 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 [9] creates a proxy class at runtime, autowired where such class is used. From

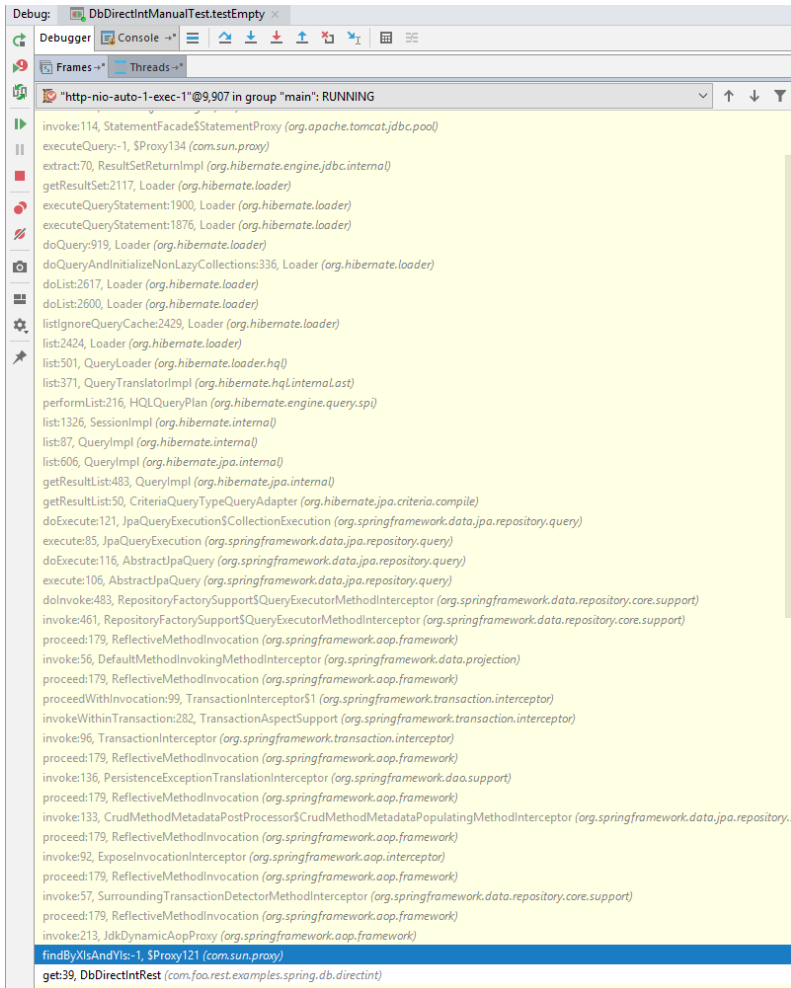


Fig. 5. Example of method-call-stack while executing the method `findByXIsAndYIs(x,y)` from the snippet in Figure 1.

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 frameworks used, like Spring [9] and Hibernate [3]. Figure 5 shows an example of the state of the method-call-stack while executing `findByXIsAndYIs(x,y)`, taken by running a test in debug mode with IntelliJ, which shows how complex and deep such calls can be.

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 a heuristic value to determine how far it was from returning any data, which is based on the result of the boolean predicate in the WHERE clause. Besides SELECT, we

also consider the other SQL commands that can have a WHERE clause, such as UPDATE and DELETE.

- (3) Such heuristic values for each WHERE clause SELECT/UPDATE/DELETE 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 [3], EclipseLink [2], JDBI [4], JOOQ [5], 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 [7]. 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 [9], 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. Intuitively, the closer is the *minimum* d_r to 0, the higher the chances of returning a non-empty row set.

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 the following example:

```
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* [36] in source code, and it is inspired/adapted from existing work on solving OCL [12] and SQL [24] constraints. For example, a constraint like $x == 0$ would have a distance $\rho(x) = \text{abs}(x - 0)$, whereas A and B would have $\rho(A, B) = \rho(A) + \rho(B)$, while for A or B it would be $\rho(A, B) = \min(\rho(A), \rho(B))$ (recall Table 1).

We support most of the boolean operations in SQL, including predicates on strings and collections. For string operations, we use the same heuristics as when EvoMASTER encounters strings in the Java bytecode. Those are based on the work of Alshraideh and Bottaci [13] (e.g., Levenshtein distance). For comparisons of time values, we convert them to their “instant” representation (e.g., number of milliseconds), and then compute their numerical distance.

The SQL standard also provides constraints such as BETWEEN and IN. Similarly to what done in [24], an expression such as $(x \text{ BETWEEN } a \text{ AND } b)$ is simply converted into $(x \geq a \ \&\& \ x \leq b)$. For the IN constraint, we first compute the distance for equality on all elements in the collection, and then return the minimum. In other words, the distance for $(x \text{ IN } \{c_0, c_1, \dots, c_n\})$ is equal to $\min(\rho(x == c_i))$, for all elements c_i in the collection.

One current technical limitation is that we do not support boolean operations involving sub-queries, like EXISTS and UNIQUE, as well as for IN when the collection is defined by a SELECT, as that impacts how we currently fetch data from the database to compute the heuristics. However, based on the logs of our experiments, it seems there is only one such kind of constraints in our whole case study. In the future, we will extend EvoMASTER to apply a similar approach as done in [24] to handle sub-queries.

For UPDATE and DELETE commands, the approach is similar. We execute a custom SELECT command to retrieve the data of all the columns involved in the WHERE clauses of the UPDATE/DELETE. Then, the same kind of distance d_r is computed.

5.3 SQL Fitness Function

Once we can collect a heuristic distance for each executed SELECT with no returned data, we need to use such distances to improve the search. While there is existing work on how to define effective distance functions on SQL predicates (e.g., [24]), how to integrate such distances into the search for system test generation poses many new research challenges.

First and foremost, our final goal is to improve *target* coverage on the SUT. Having a SELECT operation returning non-empty data sets is something that likely will improve *target* 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 *target* coverage of the SUT). Furthermore, when trying to cover a target goal g in the SUT, many SELECT/UPDATE/DELETE 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 $h(t, g) \in [0, 1]$, where $h(t, g) = 0$ means the target goal is not reached when executing test case t , and $h(t, g) = 1$ 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 $h(t, g)$ for each target goal g (e.g., branches and statements in the SUT), each such target goal will also have a secondary heuristics $s(t, g)$. When two individuals t_1 and t_2 have the *same* primary $h(t_1, g) = h(t_2, g)$ value, then the fitness function will look at the

secondary objective $s(t_1, g)$ and $s(t_2, g)$ to decide which one of the two individuals is most fit for reproduction.

The secondary $s(t, g)$ can be computed in different ways, but first we need to:

- Keep track of the action i in N which led to the best value for $h(t, g)$.
- Collect the distances d_r for *all* the SELECT queries with empty return-sets (and all failed UPDATE/DELETE) done *only* at action i of N . Let us call this group of distances D_i (which will not contain any 0 value).

Once we collect such D_i set, we consider three different ways to compute $s(t, g)$. Let k_i^t be the number of SQL commands executed by the test t at action i , where $k_i^t \geq |D_i|$, then:

- H_1 : compute the average of all these distances in a single value, i.e., $s(t, g) = \sum_{d_r \in D_i} \frac{d_r}{|D_i|}$. If D_i is empty, then $s(t, g)$ gets the worst possible value. This is because D_i being empty means that the table/view has no rows. If no rows are in the table/view, then the SELECT query will always return an empty set. Already having some rows in the table is preferable to non having any rows at all.
- H_2 : still compute $s(t, g)$ as an average like in H_1 , but only if the number k_i^t of SQL commands is the same. If when comparing a test t_1 with a test t_2 we have $k_i^{t_1} \neq k_j^{t_2}$ (where i and j are independent), then we rather prefer and select the one with larger k , i.e., doing more SQL operations.
- H_3 : instead of looking at the average values for d_r , we look at the minimum value, and prefer the lower one. If two tests t_1 and t_2 have the same minimum value, then we look at second-lowest value, and so on, until we find a difference. If there is no difference, then as in H_2 we reward the one with larger number k_i of SQL commands.

By using $s(t, g)$ as a secondary objective to minimize, the aim is to reward test cases that get closer to having SELECT operations returning non-empty data sets, and UPDATE and DELETE operations that succeed. 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 .

The three different strategies to compute $s(t, g)$ have the following motivations:

- H_1 : H_1 is the simplest, and was what we used in [20]: given a set of secondary objectives, reward any improvement on average.
- H_2 : Given a target g (e.g., a line in the SUT), there could be several paths in the code to reach it. Different paths could execute a different number of SQL commands, and lead to completely different and unrelated average heuristic values. So, the idea in H_2 is to check the average only if the two tests are doing the same SQL commands, i.e., we want to try to improve the secondary objectives without changing the execution path in the SUT. For simplicity, and to avoid complex SUT instrumentation, we use the number k_i of SQL command executions as a coarse-grained surrogate measure to check such differences. If the k_i values are different, we reward the highest. The motivation here is that, given the same heuristic $h(t, g)$, then a test that does more SQL commands will have more chances during its evolution of having SELECTs that return data. In the end, only when a WHERE predicate changes (i.e., from *false* to *true*, or vice-versa) there might be a change in the execution flow on the SUT, possibly impacting the target g . In other words, for the SUT it does not really matter if we reduce the values of the $d_r \in D_i$, unless we get some of them down to $d_r = 0$. This is the reason why, when k_i values are different, we reward higher k_i values, and not other metrics such as the number of SELECTs that return data, i.e., $r_i = k_i - |D_i|$. Such latter approach would have the negative side effect of rewarding test cases with higher r_i but lower k_i , which would result in

lower $|D_i|$, i.e., less chances of having SELECTs that might impact the execution flow when they will return data.

H_3 : Average values might be deceptive: we could improve one objective while worsening another one by the same amount, and the average would stay the same. Therefore, in H_3 we aim at trying to solve at least one of the objectives, concentrating on the most promising one. Furthermore, if different execution paths lead to g by executing more SQL commands, that would not impact the minimum value (unless a new minimum is found), whereas it would likely affect the average.

5.4 Test Bloat

For each testing target g (e.g., lines and branches, and those can be tens of thousands in a non-trivial RESTful API), EvOMASTER computes a heuristic score $h(t, g)$ when evaluating a test t . However, for the final users, it would not be so useful to end up with hundreds of thousands of long test cases. Therefore, given the same coverage, EvOMASTER tries to minimize the size and number of the generated test cases which will be included in the final test suite given as output to the user. Let $len(t)$ be the length of the action sequence of t . When comparing two test cases t_1 and t_2 , if they have the same score $h(t_1, g) = h(t_2, g)$, then EvOMASTER chooses the shorter one, e.g. a if $len(t_1) < len(t_2)$, or t_2 otherwise. Minimizing test length is important, but it is a *secondary* objective compared to achieving coverage. For example, one could trivially optimize test length by generating empty tests. But those tests would be useless, as they do not cover anything.

Longer test cases can also have a detrimental effect on the search, as they take more time to execute. Given a fixed search budget, having longer test cases means having fewer fitness evaluations. If the length is not kept under control, test cases could have gradient to increase in size indiscriminately, severely harming the search. This problem is known as *length bloat* [27].

The objective $s(t, g)$ presented in Section 5.3 is a secondary objective as well. Should s be considered less or more important than len ? Would increasing the test size be advisable if we can get an improvement at solving the WHERE clauses in the SQL commands? Or would an increase in length just have a negative effect due to fewer fitness evaluations? In this paper, we aim at answering these questions by considering two different configurations in EvOMASTER: B_1 where $s(t, g)$ is more important than $len(t)$, and the other way round in B_2 . For example, given $h(t_1, g) = h(t_2, g)$, in B_1 we would first compare s , and only if the same (i.e., $s(t_1, g) = s(t_2, g)$) we would look at test length (i.e. len). However, having in B_2 the $len(t)$ always be more important would be too naive, as it could lead to tests having $len(t) = 1$. This is a problem because, to generate data to have a non-empty SELECT in a GET request, likely we would need at least another action before it to create such data (e.g., with a POST). Therefore, B_2 only applies if both compared tests more than a single action (i.e., $len(t) > 1$), otherwise it behaves as B_1 .

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 a 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 does this make 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 could be “read-only”. This can be 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. To know what can be inserted, our technique first queries the database to extract its meta-data schema.

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.

6.2 Handling Constraints

There are some SQL column values that need to be handled specially. For example, custom CHECK expressions can bound column values (e.g., `age_max ≤ 100`) or explicitly enumerate the allowed values for a given column (e.g., `status in ('A', 'B')`). In these cases it is possible to restrict the allowed values so EvoMASTER always randomly selects among a pool of values that by construction will satisfy the CHECK expression. Currently, EvoMASTER automatically parses CHECK expressions supporting the generation of restricted data with bounds on numerical column types, enumerations and regular expressions on a single column.

However, some constraints do not depend on a single column value. Consider the case where a CHECK constraint describes a condition over several columns (e.g., `(status = 'B') iff (p_at IS NOT NULL)`). In the aforementioned example, both columns (`status` and `p_at`) must be evaluated in order to conclude if the CHECK constraint has been satisfied or not. Furthermore, if a primary key constraint is specified, we need to guarantee that all inserted values keys in the table are *unique*. In contrast, foreign keys must point to existing data inserted in previous INSERT operations.

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 when dealing with constraints over multiple column values. 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, or
- insert a value that does not satisfy a custom constraint (e.g., CHECK expressions).

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.

6.3 Existing Data

To use EvOMASTER, a developer needs to write a driver class [15], where they need to specify how to start, stop and reset the SUT. As we can now generate SQL data directly as part of the search, there is no need for the users to provide any already existing data in the databases to help the testing of the SUT. This will be done automatically by EvOMASTER.

However, there are still cases in which users might need to initialize the database with some data (done every time the SUT is reset). This is the case for example for user tables having password columns as strings (e.g., text or varchar types), where the password values are hashed (e.g., with

BCrypt). Having EvoMASTER creating valid hashed values might not be viable (at least for the moment), as the details of the hashing algorithm (and its configuration) would be part of the SUT, and it can be very complex.

Therefore, we enable EvoMASTER to deal with data already existing (if any) in the database. We do not mutate those existing values (e.g., by doing UPDATE operations). However, every time we mutate a foreign-key in one of the INSERT operations, we enable EvoMASTER to point those foreign-keys to the existing data.

6.4 Search Space

In the moment in which we enable EvoMASTER to generate data directly into the database, the search space becomes much larger. Not only we need to evolve inputs like URL query parameters and HTTP body payloads (e.g., JSON objects), but now also every SQL column would be a new input to evolve.

In evolutionary computation, given a chromosome represented by n genes, it is a common practice to have a mutation rate of $1/n$. This means that each gene would be independently mutated with probability $1/n$. The motivation is that it would lead to only 1 mutation on average, while still allowing (albeit with lower probability) several mutations at the same time to escape from local optima. In a test case with w genes for the HTTP calls (e.g., representing query parameters), adding SQL commands would lead to have z new more genes as part of the chromosome, leading to $n = w + z$ (which is what we had in [20]). This means that, in the moment in which we add SQL INSERTs in the tests, the probability of mutating the HTTP actions could drastically decrease if z is large.

In those cases, it could make sense to use a higher mutation rate (for both the SQL insertions and the HTTP actions). In this paper, we analyzed two different configurations for EvoMASTER: M_1 where the mutation rate is the standard $1/(w + z)$, and M_2 where it is $1/w$. In other words, in M_2 the probability of mutating genes is independent of the number of SQL insertions.

6.5 Regular Expressions

When dealing with databases, we saw that many columns could have constraints based on regular expressions. Different databases can define regular expressions in different ways, using different grammars. Even the same database could support different ways to write regular expressions. For example, Postgres has three different ways to write regular expressions: LIKE, SIMILAR TO and \sim . They do have different syntax, and different level of expressiveness. For example, in LIKE the underscore ‘_’ is used to match any single character, whereas a percent sign ‘%’ matches any sequence of zero or more characters. On the other hand, in \sim those symbols would be replaced by ‘.’ and ‘*’.

Besides databases, regular expressions could also be used to define constraints in Swagger on the string parameters, like URI query and path elements, using the pattern attribute. Such regular expressions would then be defined using the grammar of JavaScript (ECMA 262 to be precise).

It would be unlikely that a random string would match a given regular expression. Heuristics like the branch distance could be designed to reward evolving strings that get closer to match a given regular expression constraint. However, once we know that a given string x is bound by a regular expression, instead of sampling such a string at random, we could rather directly sample a valid string.

Given a regular expression, sampling a valid string instance is rather straightforward. However, search-based tools such as EvoMASTER would still require the ability to evolve (i.e., modify) such strings. The reason is that such strings x could be then used in other constraints as well, and affect the execution flow of the SUT. To clarify this point, let us consider a simple example. Consider

a varchar column that has a regular expression constraint specifying that such a string should be a date (e.g., something like $\backslash d\backslash d\backslash d\backslash d - \backslash d\backslash d - \backslash d\backslash d$, where $\backslash d$ is used to represent any digit). Once such a date is read from the database, the SUT extracts the year part in a variable called *year*, and then includes a branch like $\text{if}(\text{year}==2019)$. Now, the branch distance would give gradient to reward modifying the the first 4 characters of the string to be close to “2019”. So, mutation operations (which do small changes to the genes) would eventually cover that branch. However, this would be inefficient if we simply generate random strings from a regular expression (as there would be only 1 out of 10,000 possibilities to get 2019 at random). To be effective, not only we need to sample strings that match a regular expression, but we also need to be able to do small modifications on them while still satisfying the regular expression. For example, given a random instance “3412-03-21”, a mutation operation should be able to add ± 1 to any of those chars, but not replacing them with a letter like ‘a’, or adding/removing chars (i.e., modify the length of the string, assuming the pattern is not a partial match on the string), as otherwise the regular expression would not be satisfied any more.

We apply the same heuristics as in [13] to measure how close a string is to match a given pattern. We also extended the EvoMASTER gene system [17] introducing new gene types with specific search operators. We needed to create the following new gene types:

- *AnyCharacter*: representing any character. A mutation on such gene would just change value into any other valid character.
- *CharacterClassEscape*: representing special characters, like only digits (e.g., $\backslash d$) or white spaces (e.g., $\backslash s$). Mutation replaces the current value with any other valid one. In case of digits, we apply a simple $v' = (v \pm 1)\%10$.
- *CharacterRange*: representing ranges of characters, like $[0 - 9]$ and $[a - z]$. Mutation replaces the current value with either the previous or next value in the range.
- *DisjunctionList*: representing a list of disjunctions, like $(a|b|c)$. Only one is active at a time. Mutation simply activate/select one of the other disjunctions.
- *Disjunction*: representing an element in a disjunction, which is composed of a list of terms (e.g., *AnyCharacter* and *CharacterClassEscape*). Mutation select one of such terms at random, and mutate it. Furthermore, a disjunction could match a whole string, or just represent a partial match. For example, $\backslash d$ would match strings with exactly 1 digit in a LIKE constraint, but would rather match any string with at least 1 digit in a \sim regular expression. In this latter case, one would need to write it as $^{\wedge}\backslash d\$$ to match strings with exactly 1 digit (as $^{\wedge}$ matches the begin of the string, whereas $\$$ matches its end). When a disjunction is a partial match, a mutation operator can add/remove a constant prefix and a constant postfix to the value of the gene. For example, “prefix_4_postfix” would be a valid value for a partial match $\backslash d$.
- *PatternCharacterBlock*: representing a constant block of characters, like the ‘-’ symbols in the $\backslash d\backslash d\backslash d\backslash d - \backslash d\backslash d - \backslash d\backslash d$ pattern. Such gene cannot be mutated.
- *Quantifier*: representing a pattern that can be repeated any arbitrary number of times, e.g., at least once ‘+’, zero or more times ‘*’, or between 3 and 5 times ‘{3,5}’. A pattern could hence be repeated between a *min* and a *max* number of times. For each repetition, we create a gene instance that can be evolved independently from the others. However, having an unbound number of repetitions would be very inefficient. For example, it would not be wise to create a string with two billion characters for a simple $\backslash d+$. Therefore, we reduce *max* to be at most *min* plus a constant (e.g., 2). In the case of $\backslash d+$, we would hence have *min* = 1 and *max* = 3. Initially, a random number of repetitions is chosen between *min* and *max*. Mutation then would either add/remove a repetition with low probability (e.g., 10%, but only as long as the

```

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

```

Fig. 6. Example of SQL INSERTs where a foreign key points to an auto-increment primary key added in a previous insertion.

min and *max* limits would remain satisfied), or mutate one of the repeated genes, selected at random.

- *Regex*: representing the whole regular expression. It directly contains a *DisjunctionList*.

Given such gene types, the regular expression $\backslash d\backslash d\backslash d - \backslash d\backslash d - \backslash d\backslash d$ would be represented by a *Regex* containing a *DisjunctionList*, containing a single *Disjunction*, composed of 10 terms being 8 *CharacterClassEscape* (representing the $\backslash d$) and 2 *PatternCharacterBlock* (for the '-').

For each type of regular expression language we want to support, we need a parser that is able to analyze the tree-structure of the regular expressions, and then build a representative *Regex* gene object for it. We built such parsers using Antlr [1]. To the best of our knowledge, this is the first attempt in the literature of search-based software testing to define mutation operators on instances of regular expressions. We support the most common types of regular expressions, although not 100% at the moment. There are some special cases in the regex grammars which we do not support yet, as they are so rare it was not a priority to handle for this work, as they are not used by any of the SUTs in the case study.

6.6 Insertions In Output Tests

Once the search is finished, EvOMASTER outputs 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

Table 2. The Domain Specific Language (DSL) supported by EvoMASTER to generate data directly into the SQL databases.

DSL Construct	Return Type	Description
<code>sql()</code>	<code>seq</code>	Create an empty sequence of SQL insertions.
<code>seq.insertInto(t, id)</code>	<code>stmt</code>	Append a new statement to insert a row to table <i>t</i> .
<code>stmt.d(c, v)</code>	<code>stmt</code>	Specify a concrete value <i>v</i> to be inserted to column <i>c</i> .
<code>stmt.r(c, id)</code>	<code>stmt</code>	Specify that the value to be inserted on column <i>c</i> is the one that will be generated after the execution of statement <i>id</i> .
<code>stmt.and()</code>	<code>seq</code>	Return the current sequence of SQL insertions.
<code>stmt.dtos()</code>	list of DTO	Output a list of DTOs (Data-Transfer-Object).

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 6. 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”). Table 2 provides a full detailed description of the DSL constructs. Such call tells the runtime to use the value of the primary key in a previous INSERT operation with *id* `insertId` (1*L* 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 *target* coverage improvement can our novel SQL heuristics achieve?
- RQ2:** What is the impact of our SQL heuristics on test bloat?
- RQ3:** Does generating SQL data directly improve performance of the search?
- RQ4:** Which configuration of our novel technique provides the best results?
- RQ5:** How does the search budget impact the performance of our novel techniques?
- RQ6:** How do our novel techniques fare on an industrial API?

7.1 Artifact Selection

In our experiments, we used the same case study from previous work on EvoMASTER [16, 17], 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. Besides those five open-source SUTs, in our analyses we also employed an industrial API provided by one of our industrial partners. We refer to such API with the arbitrary name *industrial*.

Data about these six RESTful web services is summarized in Table 3. Subject *catwatch* is a web application that fetches GitHub statistics; *features-service* is a REST API for managing feature models of software product lines; *proxyprint* is a platform for interaction between print-shops and

Table 3. 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	5
<i>scout-api</i>	75	7479	49	14	70
<i>industrial</i>	75	5687	20	2	18
Total	320	28170	191	43	249

Table 4. Detail of data types of RESTful web services used in the empirical study. We report number of boolean columns, integer columns (including bigint, integer, etc.), floating point columns (e.g. float4, double, etc.), timestamp columns, text columns (i.e. varchar columns), UUID, XML, date and JSON columns listed in all tables of each underlying database. Values in parenthesis in integer columns represent the number of automatically incremented integer columns.

Name	boolean	integer	float	timestamp	text	UUID	XML	date	JSON
<i>catwatch</i>	0	28(1)	0	3	14	0	0	0	0
<i>features-service</i>	1	11(3)	0	0	8	0	0	0	0
<i>proxyprint</i>	3	38(9)	7	4	40	0	0	0	0
<i>rest-news</i>	0	1(0)	0	1	3	0	0	0	0
<i>scout-api</i>	3	36(7)	1	8	22	0	0	0	0
<i>industrial</i>	0	0(0)	0	6	8	1	1	1	1

Table 5. Detail of constraints in the underlying database of each RESTful web services used in the empirical study. We report for each case study the number of primary keys, foreign keys, unique constraints and check constraints on each table. Each constraint might include several columns from the same table.

Name	primary keys	foreign keys	unique constraints	check constraints
<i>catwatch</i>	3	2	0	0
<i>features-service</i>	6	6	0	0
<i>proxyprint</i>	14	19	1	0
<i>rest-news</i>	1	0	0	0
<i>scout-api</i>	13	17	3	2
<i>industrial</i>	1	0	1	5

regular consumers; *rest-news* is an artificial REST API used in an university course on microservices; *scout-api* provides a REST API for interacting with “Scout”, a hosted monitoring service.

The six 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 [44]. 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, Derby or Postgres), with up to 15 tables and 92 columns (for the *proxyprint* SUT). A detailed account of the different SQL data types is shown in Table 4. In terms of database

Table 6. Average coverage results of covered branches, statements and HTTP status codes for *Base* and the 6 combinations of H_x and B_x . Highest coverage values per SUT are in bold. Ranking (from 1 to 7) is based on the relative performance of each configuration on each of the SUTs.

SUT	Base	B_1			B_2		
		H_1	H_2	H_3	H_1	H_2	H_3
<i>catwatch</i>	1019.5 (6)	1065.7 (3)	1114.4 (1)	1083.9 (2)	1063.7 (5)	1018.9 (7)	1063.8 (4)
<i>features-service</i>	644.5 (7)	671.8 (3)	728.6 (1)	671.7 (4)	657.9 (5)	678.6 (2)	647.5 (6)
<i>proxyprint</i>	1503.6 (4)	1534.1 (1)	1524.4 (2)	1447.7 (7)	1500.9 (5)	1473.3 (6)	1506.7 (3)
<i>rest-news</i>	280.2 (7)	284.9 (3)	289.0 (2)	283.4 (5)	294.9 (1)	283.6 (4)	281.2 (6)
<i>scout-api</i>	1814.8 (5)	1848.7 (3)	1827.5 (4)	1796.0 (7)	1868.7 (1)	1807.0 (6)	1855.6 (2)
Average Rank	5.8	2.6	2.0	5.0	3.4	5.0	4.2

constraints, Table 5 shows the different constraints found on each case study, being the *industrial* case study containing the highest number of custom check constraints.

7.2 Experiment Settings

On each of the six SUTs, we ran EvoMASTER 30 times using the MIO search algorithm [16], with the search budget of 100k HTTP calls. We considered 13 different configuration settings:

- **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. This includes the 3 different settings H_1 , H_2 and H_3 (Section 5.3), combined with B_1 or B_2 (Section 5.4), for a total of $3 \times 2 = 6$ configurations.
- **SQL Heuristics & Generation (G+H)**: version in which we enable generating SQL data directly, as discussed in Section 6. As generating SQL data without a fitness function rewarding it would make little sense, in these experiments the SQL Heuristics are activated as well, with best H_x and B_x settings found in the experiments. We consider all combinations of 3 values for $n \in \{1, 3, 5\}$ (Section 6.1), and the configurations M_1 and M_2 (Section 6.4). This leads to $3 \times 2 = 6$ configurations.

For the five open-source SUTs, we had $5 \times 30 \times (1 + 6 + 6) = 1950$ independent searches, for a total of $1950 \times 100,000 = 195m$ HTTP calls. This large number of experiments required a cluster of computers, running for several days. However, for the *industrial* SUT, we only run it with 2 configurations, the *Base* one and the *Best* (out of 12) configuration found in the experiments on the open-source SUTs. The reason is that, due to technical constraints (e.g., the cluster not supporting Docker) and IP concerns, the experiments on the industrial case study required a dedicated machine.

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 7. Comparison of *Base* configuration with H_2B_1 .

SUT	Base	H	\hat{A}_{hb}	p-value
<i>catwatch</i>	1019.5	1114.4	0.67	0.037
<i>features-service</i>	644.5	728.6	0.75	0.001
<i>proxyprint</i>	1503.6	1524.4	0.48	0.762
<i>rest-news</i>	280.2	289.0	0.51	0.869
<i>scout-api</i>	1814.8	1827.5	0.53	0.741

7.3.1 RQ1: How much target coverage improvement can our novel SQL heuristics achieve? Table 6 shows the results of the experiments on the first 7 configurations, i.e. *Base* and the 6 combinations of H_x and B_x . Note that EvoMASTER does optimize for several different testing criteria at the same time, such as statement coverage, bytecode-level branch coverage and HTTP status coverage. It returns a total number of covered targets, and not a percentage.

In Table 6, the best configuration is H_2B_1 , as it has the lowest average rank (i.e., 2.0). Table 7 compares that configuration with *Base* in more details. To analyze such results, we followed the guidelines from [18]. 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. P-values are the probability of committing a Type I error (i.e., claiming two distributions are different when there is no difference). The Wilcoxon-Mann-Whitney U-test checks if two distributions \mathcal{A} and \mathcal{B} have the same stochastic order. Given a performance measure M , \hat{A}_{12} measures the probability that running algorithm \mathcal{A} yields higher M values than running another algorithm \mathcal{B} . If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A value of $\hat{A}_{12} = 0.8$ entails that we would obtain higher results 80% of the time when using \mathcal{A} instead of \mathcal{B} .

From Table 7, we can see moderate/strong improvement for two of the SUTs: *catwatch* ($A_{12} = 0.67$) and *features-service* ($A_{12} = 0.75$). For the other three SUTs, there was no statistically significant difference detectable with 30 runs (i.e., p-value > 0.05).

RQ1: SQL Heuristics provide statistically significant improvement on 2 SUTs out of 5 of up to 13% higher average coverage.

7.3.2 RQ2: What is the impact of our SQL heuristics on test bloat? The SQL heuristics do not change how EvoMASTER mutates the tests. They only impact the probability of the tests to reproduce. If a heuristic is not useful to smooth the search landscape (e.g., by removing local optima), it would not lead to an improvement. However, leading to worse results means that somehow they are impacting the search.

Longer test cases can be necessary when needing to setup the state of an application [14]. However, negative results could be obtained if the SQL heuristics lead to longer tests without an improvement in the coverage. Figure 7 shows how the average test case sizes do vary throughout the search with the different configurations of EvoMASTER. From those graphs, it is clear that those different configurations impact the test case size. In all those cases, the SQL heuristics lead to longer tests, even if with B_2 they are considered less important than the test length in the fitness function. *Bloat* is a very complex phenomenon, which is still not fully understood [27, 37].

For three SUTs, we had no significant change in coverage (Table 7), although there is an increase in test size (Figure 7). It could well be that we have two contrasting effects at play here, but that balance out each other in the end. In other words, a plausible explanation could be that any benefit provided by the SQL heuristics would be lost due to an increase in average test size that leads to

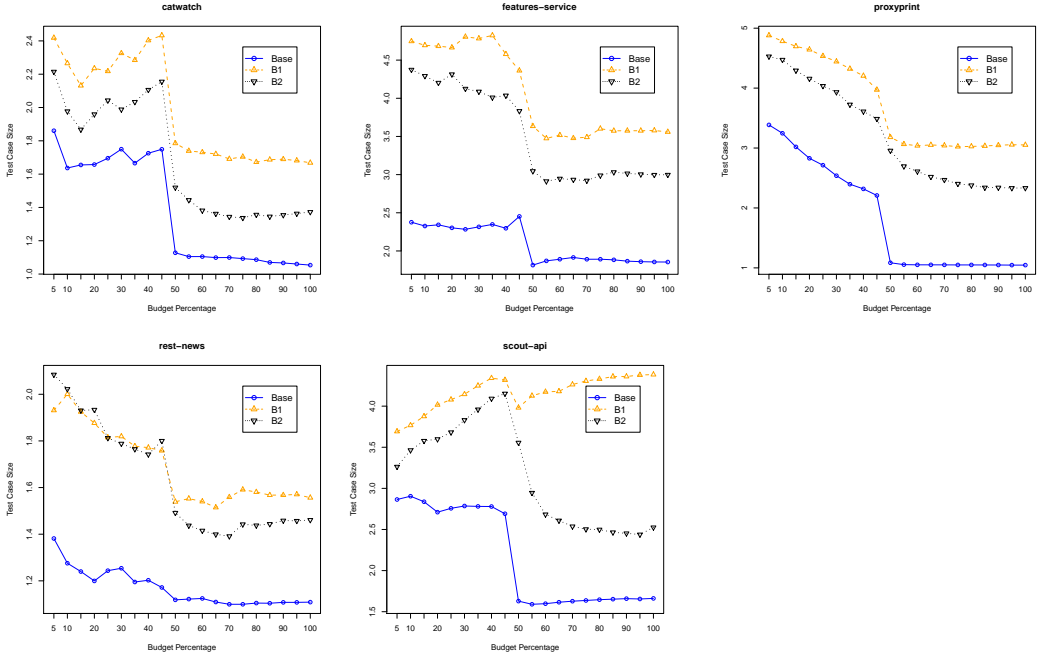


Fig. 7. Average test case size (based on the number of HTTP actions) for uncovered targets, throughout the search, using H_2 for both the B_1 and B_2 configurations.

Table 8. Average coverage results for *Base* and the 6 combinations of n and M_x , using H_2B_1 . Highest coverage values per SUT are in bold. Ranking (from 1 to 7) is based on the relative performance of each configuration on each of the SUTs.

SUT	Base	M_1			M_2		
		$n = 1$	$n = 3$	$n = 5$	$n = 1$	$n = 3$	$n = 5$
<i>catwatch</i>	1019.5 (7)	1144.6 (6)	1164.0 (4)	1155.4 (5)	1174.4 (2)	1169.9 (3)	1188.0 (1)
<i>features-service</i>	644.5 (7)	747.1 (1)	677.8 (6)	722.0 (3)	686.7 (5)	701.8 (4)	733.8 (2)
<i>proxyprint</i>	1503.6 (7)	1533.3 (4)	1530.0 (5)	1523.9 (6)	1582.6 (2)	1544.1 (3)	1627.1 (1)
<i>rest-news</i>	280.2 (7)	316.1 (2)	308.6 (5)	308.0 (6)	315.4 (3)	316.8 (1)	314.7 (4)
<i>scout-api</i>	1814.8 (2)	1777.3 (6)	1767.0 (7)	1815.2 (1)	1792.7 (5)	1805.2 (3)	1793.2 (4)
Average Rank	6.0	3.8	5.4	4.2	3.4	2.8	2.4

fewer fitness evaluations. When dealing with secondary objectives such the SQL heuristics, novel better techniques could hence be designed to avoid harmful bloat in the evolved test cases.

RQ2: The SQL heuristics have negative impact on the test length.

7.3.3 RQ3: Does generating SQL data directly improve performance of the search? Table 8 shows the results of EVO MASTER when enabling the generation of SQL data directly into the databases.

In the case of *catwatch*, improvements in target coverage were even up to $\frac{1188.0 - 1019.5}{1019.5} = +16.5\%$. The *Base* configuration is not the best (i.e., rank 1) on any of the SUTs. It is actually the worst (i.e., rank 7) in all SUTs but *scout-api*. The best configuration is $H_2B_1M_2n_5$, with average rank 2.4.

Table 9. Coverage comparison of *Base* configuration with SQL heuristics H (H_2B_1), and with SQL data generation $G + H$ ($H_2B_1M_2n_5$). Values in bold are statistically significant at $\alpha = 0.05$ level.

SUT	Base	H	G+H	\hat{A}_{hb}	\hat{A}_{gb}	\hat{A}_{gh}
<i>catwatch</i>	1019.5	1114.4	1188.0	0.67	1.00	0.80
<i>features-service</i>	644.5	728.6	733.8	0.75	0.74	0.57
<i>proxyprint</i>	1503.6	1524.4	1627.1	0.48	0.83	0.83
<i>rest-news</i>	280.2	289.0	314.7	0.51	0.93	0.84
<i>scout-api</i>	1814.8	1827.5	1793.2	0.53	0.46	0.44

Table 10. Fault comparison of *Base* configuration with SQL data generation $G + H$ ($H_2B_1M_2n_5$). We report the average number of faults that were found, with in parenthesis their maximum value out of 30 runs.

SUT	Base		G + H		\hat{A}_{gb}	p-value	Max Diff.
<i>catwatch</i>	9.4	(14)	12.3	(14)	0.95	< 0.001	0
<i>features-service</i>	16.2	(17)	18.9	(21)	1.00	< 0.001	4
<i>proxyprint</i>	31.1	(32)	32.6	(34)	0.93	< 0.001	2
<i>rest-news</i>	1.0	(1)	1.8	(2)	0.90	< 0.001	1
<i>scout-api</i>	52.3	(56)	51.8	(55)	0.44	0.560	-1

It is interesting to notice that the best configuration M_2n_5 is the one in which the SQL genes have least impact on the mutation probability of the HTTP actions. It would appear that, although generating SQL data is very useful, it should have no negative impact on the mutation of the HTTP actions. But a larger n (e.g., 5 vs. 1) could lead to less readable tests. Taking this insight into account, it could be possible to design even better mutation rates than M_2 while trying to minimize n .

RQ3: Generating SQL data significantly improves coverage, even up to +16.5%.

7.3.4 RQ4: Which configuration of our novel technique provides the best results? Table 9 shows the coverage results of the best configuration $H_2B_1M_2n_5$ with the *Base* one and with the SQL heuristics only H_2B_1 . When generating SQL data directly, results are much stronger. Improvements are statistically significant on all but one of the SUTs, with very strong effect-sizes. An effect-size like 1.00 for *catwatch* means that, in every single of the 30 runs, we got better results than in any of the other 30 runs without SQL generation. This is the strongest possible achievable \hat{A}_{12} effect-size. For *scout-api* there is no statistical difference detectable with 30 runs.

Table 10 compares $H_2B_1M_2n_5$ with *Base* in more details, looking at which faults are automatically found (these are based on 500 HTTP status codes per endpoint, taking into account the last executed instruction in the SUT). On average, $H_2B_1M_2n_5$ finds more faults, with 7 new unique ones identified (although those are not found in all of the 30 runs).

Some of the new found faults are directly related to the improvement on code coverage. For example, in the case of *rest-news*, this is due to the fact that at the moment EvoMASTER fails to generate any data into the database by just using the API HTTP endpoints, e.g., by using a POST request. This is due to input validation in the source code of the HTTP controller in the SUT, which currently EvoMASTER is not able to handle (i.e., generate the right body payload and query parameters for the that POST request). That API also has a PUT endpoint in which a specific field of a resource can be updated. However, if EvoMASTER fails to generate the resource, then that endpoint would just correctly return a 404 HTTP status, e.g., by executing:

```
if (!crud.existsById(id)) {
```

```

@Test
public void test_1_with500() throws Exception {
    List<InsertionDto> insertions = sql()
        .insertInto("NEWS_ENTITY", 0L)
        .d("ID", "-401889144")
        .d("AUTHOR_ID", "\"evoma{ter_4050_input\"")
        .d("COUNTRY", "\"evomaster_4051_inputV\"")
        .d("CREATION_TIME", "\"2099-01-15 02:10:34\"")
        .d("TEXT", "\"evomester_3931_inhmt\"")
        .dtos();
    controller.execInsertionsIntoDatabase(insertions);

    given().accept("application/vnd.tsdes.news+json;charset=UTF-8;version=2")
        .contentType("text/plain")
        .body("evomastej_3933_inpu")
        .put(baseUrlOfSut + "/news/-401889144/text")
        .then()
        .statusCode(500) // org.tsdes.spring.examples.news.api.NewsRestApi_233
        .assertThat()
        .contentType("application/vnd.tsdes.news+json")
        .body("'status'", numberMatches(500.0))
        .body("'error'", containsString("Internal Server Error"))
        .body("'path'", containsString("/news/-401889144/text"));
}

```

Fig. 8. Example of test case generated for the *rest-news* case study where the use of our novel techniques leads to find a new fault.

```

return ResponseEntity.status(404).build()
}

```

where `crud.existsById(id)` will execute a SQL SELECT command to check if the given resource identified by a given `id` exists in the database. The WHERE clause of such parametric query is `news_entity.id=?`.

By using our novel techniques, EvoMASTER in this case will detect that such SELECT returns no data. Adding data to that specific table `news_entity` will hence become part of the search, where the field `id` is numeric. Then, a basic branch distance on a numeric equality provides a clear gradient to generate input data that satisfies the constraint `news_entity.id=?`. A test case like the one in Figure 8 was then generated by EvoMASTER.

In that case, a new entry is generated with `id -401889144`, and the same `id` is used in the URL of the PUT request (i.e., `/news/-401889144/text`). With such data, the `crud.existsById(id)` command returns true, and the else branch of that if statement is taken. However, the endpoint crashes in the code that follows, specifically at line 233, due to a faulty catch statement (which tries to catch the wrong type of exception).

RQ4: The configuration $H_2B_1M_2n_5$ gives the best results, with strong statistical significance in 4 SUTs out of 5, finding 7 new faults in these SUTs.

7.3.5 RQ5: How does the search budget impact the performance of our novel techniques? When running a search algorithm, the longer we keep it running the better results we can expect. However,

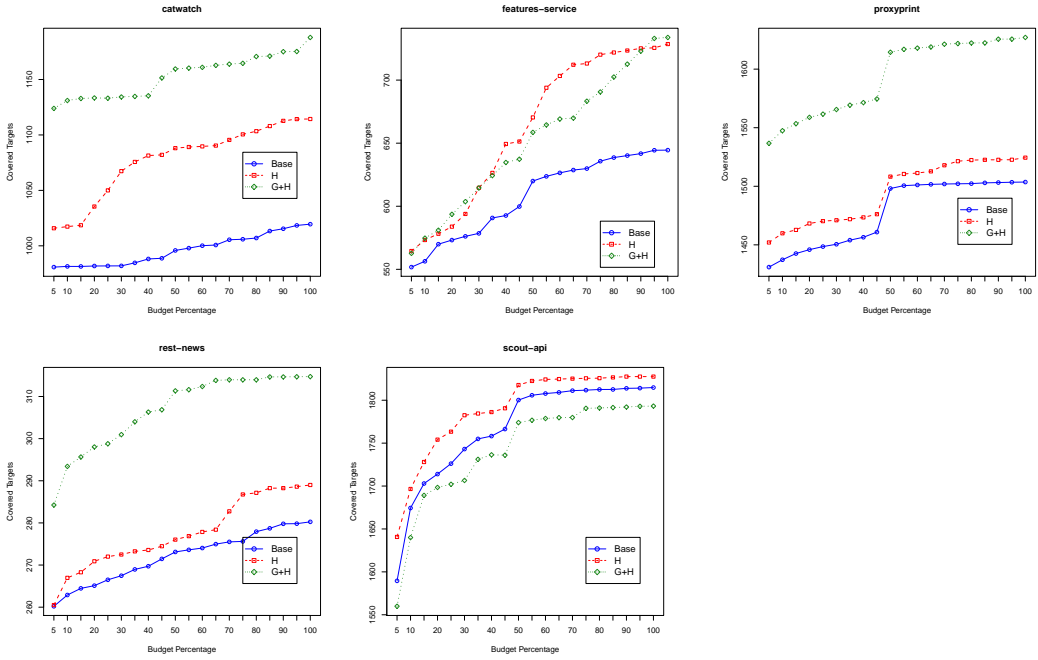


Fig. 9. Average coverage throughout the search for the configurations compared in Table 9, reported at 5% intervals of the budgeted time allocated for the search.

after a certain point, the algorithm's performance could plateau when reaching a complex local optimum in the search landscape. Based on the trade-off between *exploration* and *exploitation* of the search landscape, it could well be that a greedier algorithm (e.g., Hill Climbing and (1+1) Evolutionary Algorithm [40]) could achieve higher coverage faster, but then plateau to a worse final coverage. When comparing algorithms (and new configurations), the search budget (measured in time) does hence have an important role. Low search budgets would favor greedier algorithms, whereas high search budgets would favor algorithms that put more focus on the exploration of the search landscape.

In this paper, we chose the arbitrary stopping criterion of 100,000 HTTP calls. Depending on the hardware and the SUT, each run would take around 20 – 60 minutes, which could be considered a reasonable amount of time for an engineer using EvOMASTER. However, being system testing, an engineer could want to run EvOMASTER for much longer (e.g., over the night, or over the weekend). But too long execution times would make running several experiments non-viable. We simply chose a reasonable trade-off between execution time (100k HTTP calls), number of SUTs in the case study (6), number of repetitions (30) and number of compared configurations (13).

To analyze the effects of the search budget on the final results, we collect the average coverage at every 5% intervals of the search budget. Figure 9 shows such results. In the case of *catwatch*, *proxyprint* and *rest-news*, generating SQL data leads to a large improvement already since the beginning of the search. Such improvement seems to remain fairly constant throughout the search. This could be explained if generating SQL data make some testing targets very trivial to cover. Those will be covered fairly quickly.

Table 11. Coverage comparison of *Base* and $H_2B_1M_2n_5$.

SUT	Base	G+H	\hat{A}_{hb}	p-value
<i>industrial</i>	268.8	271.1	0.50	0.990

On the other hand, in *features-service*, $G + H$ starts with just slightly better results than *Base*, and then it gradually improves over time, with increasing gap. This could be explained if there is the need to generate SQL data with very specific values, which might take some time to evolve. But, once evolved, they lead to much better results.

Finally, *scout-api* presents a quite intriguing scenario. Although the SQL heuristics H lead to better coverage, generating SQL data (i.e., $G + H$) gives worse performance than *Base*. This is interesting because $G + H$ includes the SQL heuristics as well. On the one hand, it would appear that on such SUT it is possible to generate the right data via the HTTP endpoints, and so the SQL heuristics help in evolving the right URI query parameters and body payloads in the HTTP requests. On the other hand, generating SQL data directly seems to have some side-effects that is countering the benefits of the SQL heuristics.

RQ5: *Our novel techniques provide improvements throughout the whole search.*

7.3.6 RQ6: *How do our novel techniques fare on an industrial API?* Open-source repositories, such as GitHub, do host many libraries and tools. But web services are comparatively not so common. Furthermore, there is a threat to validity that open-source web services might not be representative for enterprise applications developed in industry. To cope with this potential issue, we applied EvoMASTER on one of the APIs of one of our industrial partners.

Due to the need of running such API on a dedicated machine, we only compared two configurations: *Base* vs. the best $H_2B_1M_2n_5$. Table 11 shows the results. Between the two configurations there was no difference in performance.

A more in-depth analysis of the *industrial* API pointed to a clear explanation. Coverage is low, with most HTTP calls failing on input validation. For example, most of the endpoints have a URL query parameter of type string, which then is used as a date (e.g., in the code by calling the method `LocalDate.parse(input)`). However, such information was missing from the Swagger schema. It would be extremely unlikely that a random string would match a valid date, and so that method `parse` does throw an exception. Unfortunately, heuristics like the *branch distance* would not provide any gradient to evolve a string representing a valid date.

Heuristics and techniques to handle the SQL databases are of no use if a database is never accessed by the generated test cases. Before we can evaluate how effective our techniques are on such industrial case study, it is clear that tools like EvoMASTER need to improve their input generation algorithm. In this particular case, there is a need for more sophisticated techniques to handle input validation. A possibility could be to use some kinds of *testability transformations* [32], where methods such as `LocalDate.parse(input)` are replaced with ones that can give gradient to the search.

Even if the achieved coverage was low, it was possible for EvoMASTER to automatically find 19 faults in this case study (faults mainly due to incorrect input validation, crashing with a 500 error status instead of a 4xx). However, such faults were found regardless of the use of our novel techniques to handle SQL databases.

RQ6: *SQL heuristics do not help the search if the SUT code dealing with SQL is not even reached due to failed input validation.*

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 online at <http://www.evomaster.org>.

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 analyze the results.

Threats to external validity come from the fact that only six 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. To cope with the issues of using only open-source projects, one of those APIs was provided by one of our industrial partners. Furthermore, experiments on system testing are quite time consuming. Given the same time budget to run the experiments, having more SUTs would have required either less repetitions (instead of 30), and/or less compared configurations (instead of 13). Although those six 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

In this paper, 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 six different RESTful APIs show that our novel techniques can improve coverage significantly, up to a +16.5% improvement. Furthermore, 7 new faults were automatically detected. However, there are several possibilities for improvement. For example, our results show that handling SQL heuristics as secondary objectives can have negative effects on test bloat. Novel techniques could be designed to minimize such negative side-effects.

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 [23]. Furthermore, it will also be important to support other kinds of databases, like NoSQL ones such that the popular MongoDB [6].

EvoMASTER is freely available online as open-source, accessible at <http://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.]. Antlr. <https://www.antlr.org/>.
- [2] [n.d.]. EclipseLink. <http://www.eclipse.org/eclipselink/>.
- [3] [n.d.]. Hibernate. <http://hibernate.org>.
- [4] [n.d.]. JDBI. <http://jdbi.org/>.
- [5] [n.d.]. JOOQ. <https://www.jooq.org/>.
- [6] [n.d.]. MongoDB. <https://www.mongodb.com/>.
- [7] [n.d.]. P6Spy. <https://github.com/p6spy/p6spy>.
- [8] [n.d.]. RestAssured. <https://github.com/rest-assured/rest-assured>.
- [9] [n.d.]. Spring Framework. <https://spring.io>.
- [10] [n.d.]. SQL. <https://www.iso.org/standard/63555.html>.
- [11] 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.
- [12] 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.
- [13] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability (STVR)* 16, 3 (2006), 175--203.
- [14] Andrea Arcuri. 2011. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Transactions on Software Engineering (TSE)* 38, 3 (2011), 497--519.
- [15] 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.
- [16] A. Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology (IST)* (2018).
- [17] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
- [18] 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.
- [19] 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.
- [20] Andrea Arcuri and Juan P Galeotti. 2019. SQL data generation to enhance search-based system testing. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 1390--1398.
- [21] A. Baresel and H. Sthamer. 2003. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation Conference (GECCO)*. 2442--2454.
- [22] 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.
- [23] C. Cadar and K. Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82--90.
- [24] 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.
- [25] M. Emmi, R. Majumdar, and K. Sen. 2007. Dynamic test input generation for database applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 151--162.
- [26] 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.
- [27] Gordon Fraser and Andrea Arcuri. 2013. Handling test length bloat. *Software Testing, Verification and Reliability (STVR)* 23, 7 (2013), 553--582.
- [28] G. Fraser and A. Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering (TSE)* 39, 2 (2013), 276--291.
- [29] 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.
- [30] Maryam Abdul Ghafoor, Muhammad Suleman Mahmood, and Junaid Haroon Siddiqui. 2019. Extending symbolic execution for automated testing of stored procedures. *Software Quality Journal* (2019), 1--35.
- [31] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.
- [32] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering (TSE)* 30, 1 (2004), 3--16.

- [33] 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.
- [34] 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.
- [35] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. 2009. Automatic creation of SQL injection and cross-site scripting attacks. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 199--209.
- [36] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* (1990), 870--879.
- [37] William B Langdon and Riccardo Poli. 1998. Fitness causes bloat. In *Soft Computing in Engineering Design and Manufacturing*. Springer, 13--22.
- [38] Y. Li and G. Fraser. 2011. Bytecode Testability Transformation. In *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings (Lecture Notes in Computer Science)*, Myra B. Cohen and Mel Ó Cinnéide (Eds.), Vol. 6956. Springer, 237--251. https://doi.org/10.1007/978-3-642-23716-4_21
- [39] 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.
- [40] Phil McMinn. 2004. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.* 14, 2 (2004), 105--156. <https://doi.org/10.1002/stvr.294>
- [41] 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.
- [42] 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.
- [43] 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.
- [44] S. Newman. 2015. *Building Microservices*. " O'Reilly Media, Inc."
- [45] 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.
- [46] 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.
- [47] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability (STVR)* 26, 5 (2016), 366--401.
- [48] J. Thomé, A. Gorla, and A. Zeller. 2014. Search-based security testing of web applications. In *International Workshop on Search-Based Software Testing (SBST)*. ACM, 5--14.
- [49] 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.