TU Delft

Delft University of Technology

Search-Based Test Data Generation for SQL Queries

Castelein, Jeroen; Aniche, Maurício; Soltani, Mozhan; Panichella, Annibale; van Deursen, Arie

DOI
10.1145/3180155.3180202

Publication date
2018

Document Version
Final published version

Published in
Proceedings of the 40th International Conference on Software Engineering

Citation (APA)
Castelein, J., Aniche, M., Soltani, M., Panichella, A., & van Deursen, A. (2018). Search-Based Test Data Generation for SQL Queries. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 1220-1230) https://doi.org/10.1145/3180155.3180202

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Search-Based Test Data Generation for SQL Queries

Jeroen Castelein[1], Maurício Aniche[1], Mozhan Soltani[1], Annibale Panichella[1,2], Arie van Deursen[1]
[1]Delft University of Technology, [2]Snt-University of Luxembourg
jeroencastelein11@gmail.com,{m.f.aniche,m.soltani,a.panichella,a.vandeursen}@tudelft.nl

ABSTRACT

Database-centric systems strongly rely on SQL queries to manage and manipulate their data. These SQL commands can range from very simple selections to queries that involve several tables, subqueries, and grouping operations. And, as with any important piece of code, developers should properly test SQL queries. In order to completely test a SQL query, developers need to create test data that exercise all possible coverage targets in a query, *e.g.*, JOINs and WHERE predicates. And indeed, this task can be challenging and time-consuming for complex queries. Previous studies have modeled the problem of generating test data as a constraint satisfaction problem and, with the help of SAT solvers, generate the required data. However, such approaches have strong limitations, such as partial support for queries with JOINs, subqueries, and strings (which are commonly used in SQL queries). In this paper, we model test data generation for SQL queries as a search-based problem. Then, we devise and evaluate three different approaches based on random search, biased random search, and genetic algorithms (GAs). The GA, in particular, uses a fitness function based on information extracted from the physical query plan of a database engine as search guidance. We then evaluate each approach in 2,135 queries extracted from three open source software and one industrial software system. Our results show that GA is able to completely cover 98.6% of all queries in the dataset, requiring only a few seconds per query. Moreover, it does not suffer from the limitations affecting state-of-the art techniques.

CCS CONCEPTS

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

KEYWORDS

search-based software engineering, automated test data generation, SQL, databases.

ACM Reference Format:
Jeroen Castelein[1], Maurício Aniche[1], Mozhan Soltani[1], Annibale Panichella[1,2], Arie van Deursen[1]. 2018. Search-Based Test Data Generation for SQL Queries. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages. https://doi.org/10.1145/3180155.3180202

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 the author(s) 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.
ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180202

1 INTRODUCTION

SQL queries form the heart of database-centric applications, which can range from systems dealing with customer relations to applications managing medical data for hospitals. Software engineers are then required to test such SQL queries as intensively as they test program code. However, the complexity of generating test data required to fully test a SQL query grows together with the complexity of the query itself.

Consider a SQL query that joins two tables and contains two predicates:

```
SELECT items.* FROM invoice
  JOIN items ON invoice.id = items.invoiceid
 WHERE amount > 1000 OR taxFree = true
```

This SQL query returns all items of invoices that either have amount greater than 1000 or that are tax free. To test this query rigorously, the tester may want to follow some coverage criteria, such as to exercise all "branches" that can be executed in this SQL query. Thus, the tester needs to target 1) the JOIN relation, 2) the left predicate (`amount > 1000`) to be evaluated to true, 3) the right predicate (`taxFree = true`) to be evaluated to true. For that to happen, the two tables should contain the right data that satisfies each of these targets. While in this illustrative example, generating test data can be done by a human, in more complex cases, *e.g.*, queries with multiple JOINs, predicates, and subqueries, the number of targets grows, and *the generation of data that test all the branches of a SQL query becomes a difficult and time-consuming task*.

Researchers have proposed approaches to automatically generate test data to support developers testing SQL queries [4, 11, 21, 34]. These approaches transform the test data generation problem into a constraint satisfaction problem [36]. Subsequently, they use constraint solvers, such as Alloy [19] and Choco [20] to generate test data solving the constraints. However, such approaches suffer from two important problems that may prevent them from being used in large software systems: First, due to limitations of the existing constraint solver tools, these approaches commonly do not support strings and any kind of string manipulation. Secondly, mapping the entire SQL language to a constraint satisfaction problem is a highly complex task. As a consequence, these approaches commonly do not support JOIN expressions, subqueries, and specific database functions, such as date/time functions.

At the same time, SQL queries often contain JOINs, subqueries, strings, and database-specific functions. For example, 30% of the queries in SuiteCRM, a large open source web application that manages customer relationships, contain at least a single JOIN, and 28% of the queries in Alura, a closed-source e-learning web application, contain at least one subquery. Consequently, the aforementioned limitations of existing solutions clearly reduce their applicability to real software systems.

RIGHTSLINK

To overcome the identified limitations, we model the problem of test data generation for SQL queries as a *search-based problem*. We opted for search-based techniques since they have been successfully applied in various software testing scenarios (e.g., white-box unit testing [24] and regression testing [38]), handling complex data structures (e.g., Java objects [13]), and for string search problems [1].

Given a SQL query, its respective database schema and a collection of coverage requirements, we implement and evaluate three different search approaches, namely random search, biased random search, and genetic algorithms, to populate tables with test data meeting a given testing criterion. The random search explores a set of randomly generated data; the biased random search improves the pure random search by seeding constants that can be extracted from the SQL query under test. Finally, the genetic algorithm (GA) is guided by a fitness function based on data collected from the physical query plan generated by a fully-functioning instrumented database engine.

We provide an implementation of the three approaches in a tool, named *EvoSQL*. To evaluate the three approaches, we execute them on 2,135 queries extracted from 4 software systems, one of them being from an industry partner. Our results show that the GA is able to completely cover 2,106 queries (98.6%) of our dataset. On average, the GA takes 2 seconds to cover queries up to 10 different coverage targets, and 15 seconds to cover queries up to 20 different coverage targets. Interestingly, we observe that the GA does not get stuck in JOINs, subqueries, or string manipulation, and thus, shows advantages over the two other search algorithms.

Our study leads to the following four contributions:

- A formulation of the test data generation problem for SQL queries as a search problem together with the definition of three different search algorithms tailored to generate test data for SQL queries (Section 3).
- An open source Java implementation of the approach, namely EvoSQL (Section 4).
- An empirical study on the effectiveness, performance, and difficulties that the three approaches face on 2,135 queries extracted from four software systems (Section 5), demonstrating that the genetic algorithm reaches full coverage for almost all cases in 2-15 seconds, making the approach usable in a practical setting.
- A replication package containing the queries and schemas used in our evaluation that can help researchers in reproducing and improving our results [7].

2 SQL TEST ADEQUACY

To enable the generation of test data for SQL queries, we must first select a test adequacy criterion. SQL queries contain different syntax and semantics that can be exercised, *e.g.*, joining, grouping, and aggregation. Consider the following SQL query:

```
SELECT *
  FROM Product
 WHERE Category = 'Toy'
```

It contains at least two different scenarios that could be tested: 1) when a row contains `Category = 'Toy'`, and 2) when a row contains `Category != 'Toy'`.

Tuya et al. [37] propose SQLFpc, a full predicate coverage criterion for SQL queries which takes into account logical operators, joins, grouping, aggregations, subqueries, case expressions and null values. Given a SQL query, SQLFpc produces coverage targets in SQL formats. Such SQL targets are satisfied when the database returns at least a single row after being populated with test data and then executed.

As an example, SQLFpc would generate two coverage targets for the query above:

(1) `SELECT * FROM Product WHERE (Category = 'Toy')`, which is, in this case, the same as the SQL under test, and
(2) `SELECT * FROM Product WHERE NOT(Category = 'Toy')`, which represents the negative counterpart.

A database that contains two rows (Row 1 = {'Toy'}, Row 2 = {'Car'}) would achieve 100% of coverage for the SQL under test, as row 1 satisfies target 1, and row 2 satisfies target 2.

In this paper, we adopt the coverage criterion by Tuya et al. [37]. Nevertheless, we devise our approach in a way that other coverage criteria can be used. Our approach only requires a coverage criterion that 1) produces a set of coverage targets in SQL format, and 2) each target is considered satisfied when a database returns non-empty results after executing it against the generated data.

3 APPROACH

Given an SQL query under test, our goal is to generate test data that satisfies a coverage criterion. The test data generation problem can be formulated as follows:

Definition 3.1 (Search Problem). Let $R = \{r_1, \ldots, r_m\}$ be the set of coverage targets for a query Q according to a coverage criterion. Find test data S that satisfies all coverage targets in R.

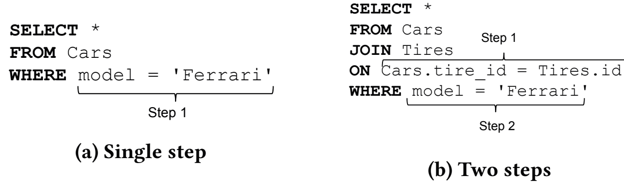
To allow the application of search-based techniques, we (i) design an encoding scheme for representing solutions of the problem in Definition 3.1; (ii) define a fitness function measuring the optimality of each solution; (iii) select three search algorithms to find optimal solutions and thus solving our problem. These aspects are described in detail in the next subsections.

3.1 Solution Representation

Given a coverage target for a SQL query, the search space is represented by all possible combinations of database tables whose rows satisfy the target. A database table has a specified (and previously defined in the supplied database schema) number of columns and can store any number of rows. The columns define the data type (*e.g.*, string or integer) that each cell in a row can store. As an example, a table `Product` has three columns: `name` which stores strings, `price` which stores doubles, and `size` which stores integers. Rows in this table always contain three elements: the first cell stores a string, the second stores a double, and the third stores an integer.

Our encoding scheme represents the set of database tables that are used in the SQL query, where each of them contains a list of non-empty rows. More specifically, a candidate solution is a set of tables $T = \{T_1, \ldots, T_n\}$, where each table T_i is composed of rows, *i.e.*, $T_i = \{R_1, \ldots, R_k\}$. Each row contains cells, *i.e.*, $R_j = \{V_1, \ldots, V_c\}$, where c is the number of columns in T_i.

RIGHTSLINK



(a) Single step **(b) Two steps**

Figure 1: Two examples of SQL queries with the corresponding steps in a *physical query plan*.

The tables and columns of a candidate solution are defined by the tables appearing in the target to solve. For example, let us consider the target `SELECT * FROM Product WHERE (price>100.00 AND size>10.00)`; a candidate solution for this target contains a list of rows for the previously defined table `Product`. As an example, a candidate solution for the query is:

$$Product = \{R_1 = \langle 'TV', 500.0, 60 \rangle\} \quad (1)$$

3.2 Fitness Function

To determine how close a candidate solution T is from covering a query target we need to define a proper fitness function. Usually, the fitness function is a distance measure $f : T \rightarrow [0, +\infty)$ that takes as input a candidate solution T and returns zero if and only if T satisfies the given target. In our case, the fitness function is computed by analyzing the execution of a SQL query in a database engine and its *physical query plan*, the plan that the database internally devises to process the query. Therefore, before describing our fitness function, we need to introduce and describe the physical query plan.

3.2.1 Physical query plan. For any SQL query to be executed, databases first convert the query into a *physical query plan* [17], which indicates the operations required to process the query as well as the order by which they need to be performed. The resulting physical query plan can be viewed as an ordered list of relational algebra operations in each of its nodes. As an example, nodes could contain a `JOIN` between two tables, or a predicate expressed in a `WHERE` statement. In the following, we refer to these relational algebra operations in the *physical query plan* as *steps* to satisfy.

Figure 1 depicts two examples of SQL queries and highlights the states in the corresponding *physical query plan*. Figure 1a contains a SQL query with a single step, which contains a single selection operation (`model = 'Ferrari'`). The SQL query in Figure 1b contains two steps: the physical query plan first commands the database to work on the `JOIN` (step one), and then, after that step is done, to work on the selection (step two). In the second case, if the `JOIN` does not produce an output (*i.e.*, the predicate `Cars.tire_id=Tires.id` is not `TRUE` for any row in both `Cars` and `Tires` table), the database realizes that the query will return an empty result, and stops its execution before proceeding to step two.

The order of the steps in the *physical query plan* as well as the order of operations in each step is automatically computed by the database engine that performs different cost calculations to come up with the best order to execute them [17]. The physical query plan also takes into account the priorities that an operation may have over another, *e.g.*, a subquery may need to be executed before a `JOIN`. Nevertheless, to cover a given query target r, a solution has to go through all the steps in the query plan, which will only happen if all predicates are evaluated to `TRUE` for at least once.

3.2.2 Fitness function definition. Executing a candidate solution against a given coverage target corresponds to executing all the steps in the plan. If a solution T fully satisfies all steps, *i.e.*, predicates in all steps, then it satisfies the coverage target under test. Given our coverage criterion, satisfying a coverage target means that the query under test, when executed in a database that contains the solution T, yields a non-empty result. If T does not cover the given target, it implies that some steps are not satisfied and the database engine terminates the execution before completing the last step in the query plan. In this case, we can measure the distance of T to cover the target by (1) counting the number of yet unsatisfied steps (*step level*), and (2) measuring how far T is to satisfy the step where the database engine stopped its execution prematurely (*step distance*).

Therefore, given a candidate solution T and a coverage target r, we design our fitness function as follows:

$$d(T, r) = step_level(T, r) + step_distance(T, L) \quad (2)$$

In the equation above, $step_level(T, r)$ measures the number of steps in the *physical query plan* not executed by the database engine when running T against the target r. The second function clause $step_distance(T, L)$ measures the distance of T in satisfying the first step L of the *physical query plan* that is not fully satisfied by T, *i.e.*, the *step distance* is computed for the step L where the database engine stopped its execution.

A step L is satisfied by T if all relational algebra operations in L are satisfied. Therefore, let $O_L = \{op_1, \ldots, op_h\}$ be the set of operations in a given step L, the corresponding step distance for a solution T is defined as:

$$step_distance(T, L) = \phi \left[\sum_{i=1}^{h} \phi \left[dist(T, op_i) \right] \right] \quad (3)$$

where op_i is the i-th operation in L, $dist(T, op_i)$ denotes the distance of T to satisfy the relational operation op_i, and ϕ is the normalizing function $\phi(x) = x/(x + 1)$ widely used in search-based software testing [24]. According to Equation 3, the step distance results in values in the range $[0; +1)$ and it is obtained by summing up the distances $dist(T, op_i)$ for all operations in the step under analysis. All operations contribute in equal manner to $step_distance(T, L)$ as they are normalized before being aggregated with the sum operator.

The actual distance $dist(T, op_i)$ for each operation $op_i \in O_L$ is defined depending on the type of the values being involved. In particular, we identify the following operations and their corresponding operation distances:

(i) *Comparison operators.* The existing comparison operations in SQL (i.e., =, <>, >, >=, <, and <=) can involve either numbers, booleans, strings, dates, or the special value `NULL`. For numbers and boolean values, the distance $dist(T, op_i)$ corresponds to the standard *branch distance* defined by Korel [22]. For example, the distance for the equality `price=10` is $abs(price - 10)$ [22]. For strings, we use the *enhanced edit distance* defined by Alshraideh et al. [1] that combines the standard edit distance with the character distance.

RIGHTSLINK

We selected this distance as it performs best when dealing with string values in test data generation [1]. For dates, the distance is computed as the sum of the differences for each numerical calendar part: year, month, day, hour, minute, second, and millisecond.

(ii) *Logical operators.* For logical operations (e.g., AND) we used the traditional branch distance rules [22] that aggregate the distances of the logical expressions (predicates) involved in the operation. For example, the branch distance for the operation `price=100.00 AND size=10.00` is $d = [abs(\texttt{price} - 100.00) + abs(\texttt{size} - 10.0)]$.

(iii) *SQL operators.* There are five operators that are specific for SQL, which are: `BETWEEN`, `IS (NOT) NULL`, `IN`, `LIKE` and `EXISTS`. For these operators, we define the corresponding operation distances as described in the following.

The `BETWEEN` operator verifies whether a given value v is within the two given bounds, i.e., if $lb <= v <= ub$ with lb and ub being the lower and the upper bound respectively. In SQL, this is equivalent to $lb <= v$ `AND` $v <= ub$ and, therefore, we use the corresponding branch distance rule for the `AND` operator.

The `IS NULL` operator returns `TRUE` when the value v under inspection is `NULL`, and `FALSE` otherwise. The `IS NOT NULL` operator is equivalent to `NOT ( IS NULL )`. Therefore, it is equivalent to the boolean operator $v \neq$ `NULL` and it is treated as the `not equal` operator when computing the operation distance.

The `IN` operator verifies whether a given value lv (left term of the operation) is contained in the list on the right part of the operation. Such a list can be a constant list written in the query or the result of a subquery. This operator returns `TRUE` if at least one of the values in the list is equal to lv. Therefore, the operation distance corresponds to the minimum branch distance between each element e in the list and lv, i.e., $dist(T, op_i) = \min_{e \in list} abs(lv - e)$.

The `LIKE` operator performs pattern matching on a string. Therefore, its operation distance corresponds to the branch distance for pattern matching defined by Alshraideh et al. [1].

The `EXISTS` operator takes a subquery as a parameter. If the subquery (once executed) returns a value, then the operation is satisfied and its operation distance is zero. Otherwise, the operation distance corresponds to the operation distance of the subquery.

The `JOIN` operator merges two tables based on a list of *join conditions*, e.g., the condition `Cars.tire_id=Tires.id` for the SQL query in Figure 1b. Join conditions are equality operators between two columns, or between pairs of columns, to satisfy. Therefore, the operation distance for a join operation is computed based on the branch distance for equality operators: $dist(T, op_i)$ is equal to the sum of the branch distances applied to all *join conditions* in op_i.

3.3 Search Algorithms

In this paper, we consider three different search algorithms: (1) Genetic Algorithms (GAs); (2) random search; and (3) biased random search. The details of these algorithms are discussed below.

3.3.1 Genetic Algorithms. GAs are stochastic search algorithms inspired by the Darwinian natural selection. These algorithms evolve a pool of N solutions (called population) using the fitness function to measure the quality of each solution. Initially, the population consists of randomly generated solutions, that correspond (in our case) to set of solutions as described in Section 3.1 and filled with randomly generated values (e.g., random integers). The population is evolved via selection and reproduction through various iterations, called generations. In each generation, individuals in the population are evaluated as follows: each individual is inserted into an in-memory database engine and executed against the target under analysis. The in-memory database is instrumented to give the complete execution trace of the query, step-by-step. This execution trace is therefore used to computed the corresponding fitness function as described in Section 3.2. After evaluating all individuals in the populations, the fittest individuals (parents) are selected using the *tournament selection* [18, 33] and then recombined using crossover and mutation. These two genetic operators generate new individuals (offsprings) that inherit some properties (tables in our case) from their parents. Then, parents and offsprings compete with each other and the N fittest individuals (according to the fitness function for the target under analysis) are selected to form the population of the next generation (*elitism*). The selection-reproduction cycle ends if one individual in the current population achieves a zero fitness value meaning that it covers the given coverage target; otherwise the search is terminated when the maximum search budget allocated for the current target is reached.

The details of the genetic operators implemented for our problem are discussed in the next paragraphs.

Crossover. As explained in Section 3.1, in our case an individual (or solutions) is a set of tables $T = \{T_1, \ldots, T_n\}$. Therefore, the crossover operator has to generate two offspring by shuffling the tables in the parent solutions. To this aim, we used the *uniform crossover* [5]: given two parents $T = \{T_1, \ldots, T_n\}$ and $X = \{X_1, \ldots, X_n\}$, the uniform crossover uses a random binary vector (crossover mask) to decide which offspring inherits each table in T and X. The mask $b = \{b_1, \ldots, b_n\}$ has the same length as the parents (i.e., the number n of tables), one binary element for each table in the solutions. If the binary element b_i is one, then the first offspring inherits the table T_i (from the first parent) while the second offspring inherits X_i (from the second parent); otherwise, the first offspring inherits X_i while the second inherits T_i. Notice that parents and offsprings always have the same number of tables.

Mutation. When an individual $T = \{T_1, \ldots, T_n\}$ is mutated, there is a $1/n$ probability for each table $T_i \in T$ to be mutated, so on average, one table is mutated in T. Three types of mutations can be applied to each $T_i \in T$, which are in order of their application *delete, change, and insert.* These operations have the same probability p_m of being applied and are not mutually exclusive, *i.e.*, more than one mutation can be applied to the same table T_i. Let $T_i = \{R_1, \ldots, R_k\}$ be one table with k rows in a given solution T to mutate. The delete mutation consists in deleting one random row from T_i.

The change mutation modifies each row r in a table T_i with a probability p_c. A row is modified by randomly changing one of its values. Let r_j be the row to mutate; each cell in r_j is modified with a $1/c$ probability, where c is the number of columns in T_i. Elements are mutated depending on their types: floating-point numbers are modified using the *polynomial mutation* [10], which is standard for real numbers; integers are mutated by using a *delta mutation*, i.e., by adding or removing a *delta* value randomly generated from the interval $[-10; 10]$; a date is mutated by applying the same *delta mutation* to all its calendar parts (e.g., days); strings are mutated by adding, removing or replacing characters [1]; finally, booleans

RIGHTSLINK

are mutated by flipping their values (e.g., from true to false). If the column to mutate is `nullable`, one of its value is set to `NULL` with a probability p_{null}.

The last mutation operator adds one row to a given table T_i. It either duplicates one existing row in T or it adds a newly randomly generated row (i.e., a row containing randomly generated values).

In a given query under test, not all columns of the table are exercised. Therefore, we limit its search space by ignoring columns that are not used by any predicate, and instead, considering the other columns to which we refer to as *mutable columns*. The approach uses a naive technique to identify the mutable columns: if a column is used anywhere in the query's important clauses (`FROM`, `WHERE`, `GROUP BY`, `HAVING`), it is added to the list of mutable columns.

Seeding strategies. Seeding is the technique of inserting values from a seeding pool into the population with some probability. The values in the seeding pool are extracted using the knowledge taken from the query under test. Our GA uses a seeding pool containing all constants (e.g., strings and integers) appearing in the query. This is because a query may contain comparisons with constant values. Each constant in the query is extracted and added to the *column seeding* pool, which will be used to be seeded into columns of the generated individuals, when mutating a row.

We use a further seeding strategy specific for join operations. A typical SQL predicate for joining two tables together is an equality between two columns. Each predicate of the form $column_1 = column_2$ adds a logical link between the two columns. Therefore, the seeding is applied by copying some values from $column_1$ into column $column_2$, and vice versa.

Post Processing. Readability of the generated data is important when it comes to comprehending the `SQL` query and writing test cases for it. To improve the readability of the generated data, when a solution is found, we apply a naive *row minimization* technique: for each table T_i in the generated solution T, we remove all rows that do not contribute to covering the coverage target under analysis.

3.3.2 Random Search.

As for GAs, random search is executed against each target to cover independently. It iteratively generates random solutions, and this process is repeated until the coverage target is satisfied (*i.e.*, the coverage target, when executed against the generated data, yields a non-empty result) or the maximum search budget allocated for the current target is reached.

Random search is a simple algorithm that does not evolve existing solutions. However, it is often used in the literature as baseline the to test the complexity of the problem to solve and to assess the need for more advanced algorithms (e.g., GAs). Moreover, random search has been shown to outperform other search algorithms when solving specific problems [3, 30].

3.3.3 Biased Random Search.

In addition to the simple random search algorithm, in this paper we used another variant of random search that uses the same seeding strategy used in GAs. Therefore, randomly generated solutions contain (with a given probability) values seeded from the constants appearing in the query (i.e., from the *column seeding*) or obtained by applying the seeding strategy specific for join operations.

3.3.4 Search budget allocation.

All search algorithms described above can optimize only one single coverage target at one time. To solve multiple coverage targets they have to be executed several times, one run is executed independently for each coverage target. Therefore, the total search budget SB given to test each query Q is divided in equal manner among all coverage targets of the SQLFpc coverage criterion [37]. In other words, the local budget assigned for each coverage target is $SL = SB/m$, where m is the total number of targets for Q. If one of the coverage targets is satisfied without fully consuming the local budget SL, the saved search time is used to dynamically increase the budgets for the remaining uncovered targets.

The coverage targets generated for a query differ from each other for only few spare operations and predicates (i.e., they are very similar). Therefore, test data covering one target may be very close to covering other similar targets (according to their fitness functions). For this reason, some solutions generated when optimizing previous coverage targets are used to seed the initial population in the GA.

4 OUR IMPLEMENTATION: EVOSQL

We provide *EvoSQL*, a tool implementing the database engine instrumentation, the fitness function and the three search algorithms described in the previous section. EvoSQL takes as input a query, a database schema, and a time budget, and returns test data for each SQLFpc coverage target. Our implementation is available open source and can be found in our GitHub [6] as well as in our appendix [7].

EvoSQL internally uses HSQLDB[1], a Java relational database engine that supports the latest SQL standards and is able to run in-memory. We modified the source code of the database engine to instrument the *physical query plan* generated during a query execution, which is used to calculate our fitness function.

To extract the coverage targets, our tool uses the webservice that is made available by Tuya *et al.* [37]. The webservice receives as an input the SQL query and the database schema and returns a list of coverage targets in SQL format.

We disabled internal optimizations of HSQLDB as they would reduce the amount of information we could collect. In particular, we disabled (1) *indexing*, as it excludes rows that do not satisfy predicates of indexed columns without individually evaluating them, and (2) *lazy* AND/OR optimizations, *i.e.*, short-circuit evaluation, which makes the engine potentially not evaluate all predicates.

5 EMPIRICAL STUDY

The *goal* of this study is to evaluate the effectiveness of the three different search algorithms when generating test data for SQL queries. More specifically, we investigate the following research questions:

- **RQ$_1$:** *What is the coverage achieved by the proposed search-based algorithms?*
- **RQ$_2$:** *What is the performance of the proposed search-based algorithms?*
- **RQ$_3$:** *What causes the different approaches to not achieve 100% coverage?*

[1]HSQLDB - http://hsqldb.org/

RIGHTSLINK

Application	Total Queries	Queries w/o bad syntax	Unique Queries	Final Queries
Alura	554	494	258	**249**
EspoCRM	151	149	40	**40**
SuiteCRM	709	704	280	**279**
ERPNext	18,454	17,761	1,631	**1,567**
Total	19,868	19,108	2,209	**2,135**

Table 1: Queries collected per application

Property	0	1/ 2	3/ 4	5/ 6	7/ 8	9/ 10	11/ 15	16/ 20	21+
Predicates	58	1389	495	100	33	11	27	16	6
Joins	1890	189	32	3	17	2	-	1	1
Subqueries	2052	78	3	1	-	-	1	-	-
Functions	1796	291	12	16	2	6	12	-	-
Columns	60	1369	457	127	43	26	20	13	20
Targets	-	656	382	408	346	114	107	51	71

Table 2: Number of queries according to their different properties

5.1 Context of the Study

We evaluate the different search algorithms on 2, 135 queries taken from four software systems:

- Alura[2] is a closed source Java e-learning platform that uses Hibernate as layer between application and database. Hibernate generates SQL queries based on incoming data requests.
- EspoCRM[3] is an open source web application to manage customer relationships (CRM). It uses a REST API backend written in PHP which communicates with a MySQL database.
- SuiteCRM[4] is another open source CRM. It is a fork of the SugarCRM Community Edition, and is written in PHP. The database it uses can be either MySQL, MariaDB or SQL Server. For our evaluation, we used MySQL.
- ERPNext[5] is an end-to-end business solution that manages business information (ERP stands for Enterprise Resource Planning). It is built on top of the Python & JavaScript framework *Frappé* and uses MariaDB.

We chose these systems because they are database-centric, *i.e.*, they make intensive use of databases and, thus, contain a large number of SQL queries. In addition, these systems are written in three different programming languages (Java, PHP, and Python), giving us a more diverse sample of projects and queries, as the way a developer writes a SQL query might be influenced by the overall ecosystem of the chosen language (*e.g.*, Java developers typically used Hibernate to generate SQL queries). Apart from the industrial system, all others can be found on GitHub, enabling other researchers to reproduce our results.

[2] Alura- http://www.alura.com.br/.
[3] EspoCRM - https://www.espocrm.com.
[4] SuiteCRM - https://suitecrm.com.
[5] ERPNext - https://erpnext.com.

The queries were collected by executing the test suites of each system and mining the generated database logs. Table 1 shows the total number of queries collected per system as well as the number of queries selected in our empirical evaluation. For the query selection, we analyzed all extracted queries to filter out non-executable and duplicated queries. In particular, we filtered out **HSQLDB and SQLFpc non-compliant SQL**, i.e., queries containing SQL constructs that are not supported by either HSQLDB or SQLFpc. We also removed **duplicated queries**, i.e., similar queries differing by some constant values. For example, the queries `SELECT * FROM t WHERE a = 1` and `SELECT * FROM t WHERE a = 2` are similar, as their only difference is a constant (1 and 2). As there is no difference in solving these queries, we exclude duplicated queries. Finally, we ignored **queries with no predicates or other constraints**, *i.e.*, , queries with no conditional branches to be exercised, and thus, without coverage targets.

In Table 2, we report the characteristics of the queries in our dataset. As expected, the number of coverage targets of a query is strongly correlated with the number of its properties (Spearman correlation=0.95, p-value < 0.01). As queries with fewer properties could be easier to solve than (complex) queries with more properties, we control our results by the number of coverage targets.

5.2 Configuration of the Search Parameters

The performance of search algorithms is influenced by a large number of parameters. To identify the best configuration of parameters for the applied Genetic Algorithm (GA), we executed the algorithm in a training set that contained 100 queries. We carefully devised this training set to contain all SQL constructs that are supported by HSQLDB, such as `JOIN`s, `WHERE`s, subqueries, and string functions. The training set is available in our online appendix [7]. To evaluate various possible configurations, we opted for a set of values and thresholds that are commonly used in applying evolutionary search algorithms on similar software engineering problems [14, 32]. The exercised configurations for different probabilities (in a total of 108 different combinations) are as follows: (1) `NULL` mutation (p_{null}) = $\{0.01, 0.10, 0.50\}$, (2) inserting, deleting or duplicating a row (p_m) = $\{1/3, 1/6\}$, (3) row change mutation (p_c) = $\{1/n, 1\}$, where n is the number of rows in the mutated table, (4) seeding = $\{0.01, 0.10, 0.50\}$, and (5) crossover = $\{0.0, 0.6, 0.75\}$.

For each combination of probabilities, we executed the approach 10 times and averaged the execution time of the complete training set. At the end, we selected the configuration with the smallest execution time. The best configuration we identified is as the following:

- Population size = 50.
- Tournament size = 4.
- `NULL` mutation (p_{null}) = 0.1.
- Inserting, deleting, and duplicating (p_m) = 1/3.
- Row Change Mutation (p_c) = 1.
- Seeding = 0.5
- Crossover = 0.75.
- Cloning from previous target population = 0.6.

The time budget for the search process is 30 minutes. Finally, to compare the biased random search with the GA, we set the same probability for seeding individuals which is 0.5.

RIGHTSLINK

5.3 Experimental Procedure

To answer RQ_1, we executed the three approaches on the entire dataset. Given the randomized natural of the algorithms, we execute each of them 10 times. We perform 10 runs, the minimum suggested in literature [2] and a common choice for expensive experiments [15, 28], which is our case. For each query, we average their coverage across the 10 executions. We consider a query to be *successfully covered* if its average coverage is 100%, meaning that all its targets were covered in all 10 executions. If any target was not covered in any of the executions, that query is considered to be *not fully covered.* Such definition gives us the "worst-case view" on the results.

Throughout the research, we have experienced that some of the coverage targets that are produced by SQLFpc are infeasible. In most of these cases, this is due to some combination of predicates that cannot be satisfied, *e.g.*, `A > 10 and A < 10`. Such infeasible targets may affect the analysis of our results, as a query would then be marked as not succesfully covered. We manually analyzed each target produced by SQLFpc, and removed them from our analysis. In the end, we removed 127 out of the 12, 991 total coverage targets.

To answer RQ_2, we analyze the average execution time per query in each approach. Once more, we control the queries by their coverage targets to analyze whether this has an effect on the execution time. We provide descriptive statistics of each approach.

Finally, to answer RQ_3, we train and understand a J48 decision tree [31] that classifies whether a coverage target is likely to be covered or not based on the results in RQ_1, using R's `RWeka` package. We use the following SQL constructs as features to the model: number of tables used, number of base predicates (does not include `AND`, `OR`, and `NOT` operators), number of inner joins, number of left joins, number of right joins, number of subqueries, number of aggregate functions (`MIN, MAX, SUM, AVG, COUNT`), number of non-aggregate functions (e.g., `DATENOW, IFNULL`), number of columns, number of `WHERE` clauses, number of `GROUP BY` clauses, number of `HAVING` clauses, number of string equality predicates, number of date equality predicates, number of EXIST predicates, number of LIKE predicates, number of IFNULL functions, and the SQL query's total number of coverage targets.

As classes (failed and successful coverage targets) are not evenly balanced (there are more failing cases in the random search and more successful cases in the biased and GA search), we apply SMOTE (Synthetic Minority Over-sampling TEchnique) to generate extra data points for the smaller class [8]. This way, both classes have equal size, which prevents the classifier from generating a biased model. Finally, to enable the model to be understood by a human, we limit the number of leaves in the tree by 10. We also report the accuracy of the generated models, *i.e.*, , the percentage of instances in our dataset that are correctly classified by the model. The accuracy gives us an estimate of how much we can trust on the model.

Replication package. We provide an open source replication package [7] that contains (1) our implementations of the three search-based algorithms, (2) the R scripts used to generate the analysis, and (3) the queries and schemas from all systems but the closed-source application Alura, which enables researchers to replicate and further compare with other tools.

6 RESULTS

6.1 RQ_1: What is the coverage achieved by the proposed search-based algorithms?

In Figure 2, we show a boxplot of average coverage that each search-based approach achieved. In addition, in Table 3 we present the number of queries each approach was able to completely cover, controlled by the number of coverage targets. From this data, we observe that:

The random search proved to be highly inefficient. Among the 2,135 queries, the random baseline was able to completely cover only 140 (6.5%) of them. It also achieved a partial median coverage of only 33.75% among the remaining queries, and completely failed (*i.e.*, achieved 0% coverage) in 605 queries (28.33%). Its ability to find a solution quickly decreases as the number of coverage targets increases. In practice, random search was only able to completely cover queries with less than 8 coverage targets. Still, the number of times it fails is relatively high even in queries with less targets, *e.g.*, the random search was not able to solve any of the 21 queries with 1 or 2 coverage targets in SuiteCRM, and only 34 out of 263 queries with 3 or 4 coverage targets in ERPNext.

The biased search presents good efficacy in queries with less than 10 coverage targets. The biased search was able to completely cover 1,923 (90%) queries, had a partial median coverage of 79.76%, and completely failed only in 11 queries (0.05%). For queries with less than 10 coverage targets, the biased search could only not completely cover 66 out of 1,906 queries (3.46%). However, its effectiveness also decreases as the number of coverage targets increases. In particular, it rapidly decreases after 10 targets: it solved only 83 out of the 229 queries (36.24%) with more than 10 targets, and only 5 out of the 71 queries with more than 20 coverage targets.

The GA is the most effective approach. It completely covered 2,106 queries (98.64%), had a partial median coverage of 86.66% among the remaining ones, and did not completely fail in any of the queries in the dataset. The GA achieved high efficacy in queries with less than 10 coverage targets, as it did not completely covered only 3 out of 1,906 queries. However, we still observe that its performance decreases as the number of coverage targets increases. Still, the decrease is much smaller than in the previous approaches. Even in queries with more than 20 coverage targets, the GA was able to solve 53 out of the 71 existing queries (74.64%).

6.2 RQ_2: What is the performance of the proposed search-based algorithms?

In Table 4, we show descriptive statistics of the average runtime of each approach. In addition, in Figure 3, we show the average query coverage given a time budget for the biased and GA approaches, respectively. We do not present data for the random search as it fails in most cases, and thus is not a competitor; the full data can be found in our online appendix [7]. We observe that:

The biased search is the fastest for queries up to 5-6 coverage targets. Both the GA and the biased search are able to solve queries up to 6 coverage targets in less than one second. However, the biased search is even faster than the GA for such cases, *e.g.*, for queries

RIGHTSLINK

# of targets	1 - 2 (656 queries)			3 - 4 (382 queries)			5 - 6 (408 queries)			7 - 8 (346 queries)		
	Random	Biased	GA	Random	Biased	GA	Random	Biased	GA	Random	Biased	GA
Alura	2/14	13/14	14/14	11/51	50/51	51/51	0/33	33/33	33/33	0/38	35/38	37/38
SuiteCRM	0/21	21/21	21/21	7/66	66/66	66/66	1/117	115/117	117/117	1/45	45/45	45/45
ERPNext	61/621	621/621	621/621	34/263	262/263	263/263	21/240	230/240	240/240	2/260	242/260	259/260
EspoCRM	0/0	0/0	0/0	0/2	2/2	2/2	0/18	13/18	18/18	0/3	3/3	3/3
# of targets	9 - 10 (114 queries)			11 - 15 (107 queries)			16 - 20 (51 queries)			21+ (71 queries)		
	Random	Biased	GA	Random	Biased	GA	Random	Biased	GA	Random	Biased	GA
Alura	0/24	18/24	24/24	0/46	25/46	46/46	0/25	8/25	24/25	0/18	3/18	15/18
SuiteCRM	0/19	17/19	18/19	0/7	5/7	6/7	0/2	0/2	2/2	0/2	0/2	2/2
ERPNext	0/70	53/70	70/70	0/53	33/53	50/53	0/21	4/21	18/21	0/39	2/39	29/39
EspoCRM	0/1	1/1	1/1	0/1	1/1	1/1	0/3	2/3	3/3	0/12	0/12	7/12

Table 3: Number of completely successfully covered SQL queries vs their total number.

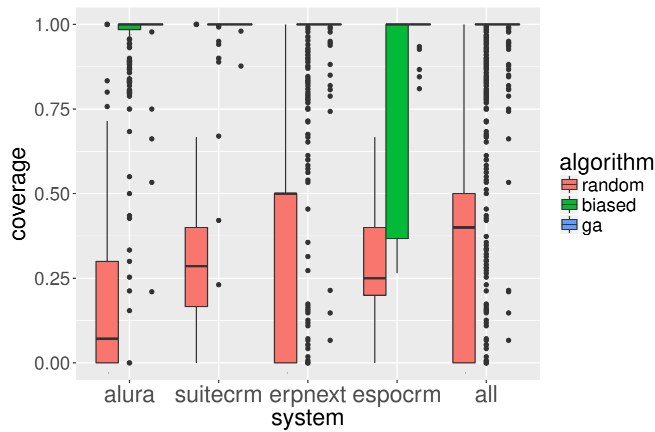


Figure 2: The average coverage for a SQL query achieved by each search-based approach, after 10 executions. Figure better visualized in colors.

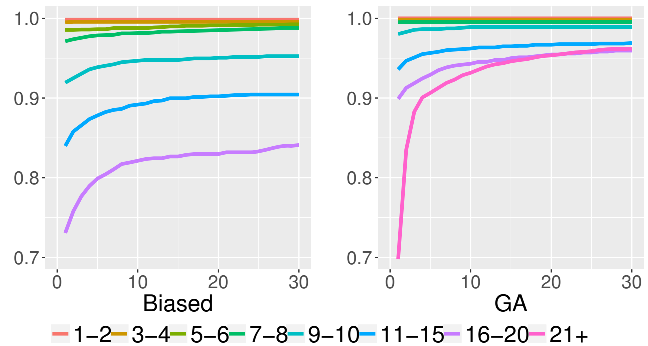


Figure 3: The average coverage of a SQL query (Y axis) in a given time budget (X axis, in minutes) controlled by the number of coverage targets. Figure better visualized in colors.

with 1-2 coverage targets, the biased search has a median runtime of 0.05 seconds, compared to 0.22 of the GA (a difference of 0.17s).

The GA becomes much faster than the biased search as the number of coverage targets increase. As soon as the number of coverage targets starts to increase, both the biased search and the GA become slower. However, the runtime of the biased search grows rapidly, while the GA's runtime growth is less aggressive. As an example, the median runtime for the biased search in queries with 9 to 10 coverage targets is 5.95 seconds while GA presents a median of 1.48 seconds; for 11 to 15 coverage targets, the biased search takes, on average, 74.04 seconds (a 12x increase when compared to 9-10 targets) while the GA takes 3.65 seconds (an increase of 2.4x).

A time budget of one minute is enough for the GA to completely cover simple queries and to cover at least 70% of complex queries. A time budget of one minute is, on average, sufficient for the GA to completely cover queries with 6 coverage targets or less; for the biased search, the same occurs for queries with 4 or less coverage targets. However, for complex queries, the given 30 minutes budget does not seem enough for the biased search. In particular, even after 30 minutes, queries with more than 20 coverage targets achieve only 51% median coverage (not even appearing in the Figure, due to our scale). On the other hand, the GA needs a single minute to cover at least 70% of queries with more than 20 targets, and 90% in queries ranging from 16 to 20 targets. To achieve 90% of coverage in queries with more than 20 targets, the GA needs, on average, 4 minutes.

	1st quantile		Median		3rd quantile	
	Biased	GA	Biased	GA	Biased	GA
1-2	0.03	0.15	0.05	0.22	0.07	0.37
3-4	0.09	0.2	0.15	0.28	0.36	0.46
5-6	0.17	0.3	0.34	0.41	0.56	0.58
7-8	1.23	0.59	2.25	0.88	7.64	1.45
9-10	1.26	0.9	5.95	1.48	100.80	2.49
11-15	7.47	2.21	74.04	3.65	844.10	8.80
16-20	83.1	7.33	844.00	15.69	1539.00	41.79
21+	1027	81.36	1688.00	155.90	1800.00	581.90

Table 4: Descriptive statistics of the biased and GA approaches' average runtime (in seconds).

RIGHTSLINK

6.3 RQ$_3$: What causes the different approaches to not achieve 100% coverage?

The decision tree models generated after the random search, biased search, and GA results, achieved an accuracy of 85.93%, 90.27%, and 91.88%, respectively. In the following paragraphs, we present and discuss them:

The random search can not deal with JOINs and strings. The model classifies any query with JOINs as a failing one. In addition, even in queries with no JOINs, the existence of a single string equality also makes the model to classify the target as failing.

The biased search suffers from queries with many predicates. The model classifies any query with more than 6 predicates as a failing one. On the other hand, queries with 4 predicates or less are classified as successful. When the query contains between 4 and 6 predicates, the model uses the number of joins as a way to differentiate between them: the model classifies the targets with the existence of a single JOIN as a failing one. Interestingly, the strings feature does not appear in the model; we discusss more about this in Section 7.

The size of the query impacts the GA. The classifier classifies any query with less than 5 predicates or with more than 5 predicates and less than 60 columns as successful. No other feature seemed relevant to the model, meaning that the number of columns in a query is what impacts the performance of the algorithm.

7 DISCUSSION

7.1 Biased search vs Genetic Algorithm

Our results indicate that biased search, although highly efficient for simple queries, suffers in queries with many predicates. The numbers also show that, for simple queries, its runtime performance can be even better than that achieved by the GA. This can be explained in two ways: First, the experimental dataset contains several string comparisons and non-complex LIKE commands, both of which can be solved by seeding. Our dataset, however, did not contain more complex string manipulation functions, such as *concatenations, left, right, length, reverse*, or combinations of them. Although our data set suggests that these operations do not often appear in SQL queries, developers can use such functions.

After further investigation with manually created cases that did actually contain such operations, we observed that the GA is the only approach that is able to find solution in such cases. For example, the queries `SELECT * FROM product WHERE length(name) = 12 AND left(name, 5) = 'REFRI' AND right(name, 7) = 'GERATOR` and `SELECT * FROM product WHERE reverse(name) = 'ROTAREGIRFER'` are not solved by the biased search. On the other hand, given the instrumentation that we perform in the database engine to guide the fitness function, the GA approach solves this particular query in a few seconds. Although these queries are just examples, they illustrate how the biased search is unable to solve complex cases where guidance is needed.

Second, although the GA implementation has an initialization step that is similar to what happens in the biased search, at every iteration of the evolution, the GA spends time calculating the fitnesses, and applying the search operators. All these calculations do not happen in the biased search. Thus, for less complex queries, these extra actions that are taken by the GA explain the small difference in the runtime performance.

It is also worth noticing that the inefficiency of GA can be due to the single-target search strategy: GA is re-executed multiple times, once for each coverage target. However, such a search strategy often leads to an inefficient allocation of the search budget[14] as the order by which the targets are optimized (over independent runs) by GA strongly impacts the search effectiveness [14]. To address this limitation, various multi-target evolutionary algorithms have been proposed literature in the context of white-box unit testing [14, 28, 29]. Therefore, better results may be obtained when using these multi-target strategies, whose evaluation is part of our future agenda.

7.2 Implications

We designed and implemented EvoSQL with the working developer in mind, and anticipate the following usage scenarios:

(i) **Query unit testing**: As we saw in our experimental dataset, queries can be highly complex, containing many predicates and subqueries. EvoSQL provides developers with test data that completely exercises the query, enabling them to verify whether the behavior of the query is exactly as expected.

(ii) **Query regression testing**: Support developers in refactoring or evolving their queries, using data sets generated from an earlier version as an oracle to ensure that the queries preserve the desired behavior.

(iii) **Integration testing**: Support developers in writing integration tests, helping them to create the right data sets enabling them to trigger and test interesting interactions between code and queries.

The use of genetic algorithms for generating SQL test data opens up interesting areas for future research. Our current approach, based on HSQLDB, only exercises queries expressed by the standard SQL92 Specification. Different databases (*e.g.*, MySQL, Oracle, Postgres) provide their own functions, data types, and constructs, which were not included in this study. Oracle, for example, provides a data type to store XML. For each database-specific function or data type, our proposed fitness function can be optimized with tailored branch distances, mutation operators, and random data generation.

From an integration perspective, dependencies on external infrastructures, such as databases and files, currently hinder the power of automated test generation tools [15]. Researchers have worked on achieving high branch coverage of source code that interacts with such infrastructure by either generating database test data from scratch or using previously existing databases [9, 12, 23, 27, 35]. However, we discuss in Section 8, they suffer from different limitations. Our GA approach has no such restrictions, suggesting that the interplay of our technique and source code test generation could increase test coverage substantially.

Finally, we did not consider local search or memetic algorithms (the combination of local and global search) [16]. Future research may explore the impact of combining local search with the global one on the coverage rate and performance.

RIGHTSLINK

7.3 Threats to Validity

With respect to internal validity, (1) we collected the 2,135 from the four systems after executing their existing test suites and extracting the SQL queries from the database logs. Thus, our dataset is limited by the queries that are actually exercised by their test suites. However, as we see in Table 2, the final dataset has shown to be diverse, ranging from simple queries with few constructs to large and complex queries with more than 20 columns and coverage paths. (2) The analysis of the infeasible targets that were generated by SQLFpc was performed manually. To reduce the risk of classifying a feasible query as infeasible, the analysis were conducted by the first two authors of this paper. (3) We defined the internal probabilities of the GA approach after performing experimenting different combinations of probabilities in a set of 100 tests. As testing all possible probabilities would imply in an explosion of combinations, we experimented a set of well-known intervals in the field [14, 32]. Nevertheless, there might be better configurations, and future work needs to be conducted on this matter.

With respect to external validity, we analyzed 2,135 from four different systems, three of them being open source and one of them being an industry system. Although systems were different in their programming languages as well as in their nature, more research needs to be conducted to generalize our results to SQL queries in any software system.

8 RELATED WORK

Several approaches [4, 11, 21, 34] have been introduced by researchers with similar goals to EvoSQL: generating test data or test databases based on one or more SQL queries. These approaches are based on constraint solving, which transform SQL queries to constraints and apply constraints solvers to satisfy them. Therefore, these approaches suffer from two main limitations, namely: 1) they are not able to describe the entire SQL syntax as constraints, and 2) solvers may not be able to satisfy certain constraints, e.g., when SQL queries involve common constructions such as subqueries and String predicates (in our evaluation set, 84.1% of queries contained such constructions).

EvoSQL takes a different approach and benefits from using an existing, fully functioning SQL database engine. As a consequence, all queries using standard SQL syntax are supported. Unfortunately, to the best of our knowledge, none of the tools discussed in this section are available for download and re-implementating them would be a highly demanding task. This prevented us from doing an empirical comparison between them and EvoSQL. In the following, we discuss the existing approaches [4, 11, 21, 34] based on what is reported in their papers, and highlight the differences between our approach and them.

QAGrow, presented by Suárez-Cabal et al. [34], generates test databases for a set of queries, using SQLFpc [37] as coverage criterion. The approach generates test databases by formulating the problem of generating data for a query as a constraint satisfaction problem, in which the current database state is also taken into account. They then use a SAT solver, *Choco* [20], to generate the test data. They evaluated their approach on 215 queries taken from a closed-source system. On these queries, they achieved 99.0% SQLFpc coverage, in about two minutes time. However, QAGrow does not solve queries which contain strings and subqueries.

Emmi et al. [11] describe an algorithm to automatically generate test input for database applications. Their goal, however, is not to test the SQL query itself, but maximizing branch coverage of the program code under test. The approach also uses a constraint solver, where the constraints are a combination of the path constraints in the program and the database constraints. Although the approach handles string constraints (equality, inequality and LIKE), it supports only `FROM` and `WHERE` clauses, but no `JOIN`s.

Khalek et al. [21] present ADUSA, a tool that generates test data for SQL queries. The paper uses the generated data to find faults in database systems, such as MySQL and HSQLDB. Thus, they generate repeated test data for the same SQL query. The approach models SQL queries into *Alloy* [19] specifications. ADUSA supports `FROM`, `WHERE`, `GROUP BY` and `HAVING` clauses. However, ADUSA only supports natural joins (an implicit join between two tables, using common column names) and cross joins (which have no predicates). In addition, the Alloy solver is unable to solve string constraints.

The QAGen tool, presented by Binnig et al. [4], also generates test databases with the goal of testing a DBMS. The approach uses a constraint solver to generate test data. The model is built using symbolic query processing, their extension of symbolic execution. QAGen does not support subqueries, and solely supports `JOIN`s that use foreign key constraints in the `JOIN` predicate.

Finally, McMinn *et al.* [25, 26] have worked on generating tests for the integrity constraints (*e.g.*, "must not be NULL" constraint) that may exist in a database schema using a search-based approach. Their work complements ours as, in practice, developers must completely test their databases: this includes both the integrity of the schemas as well as executed SQL queries.

9 CONCLUSION

The goal of this paper is to understand how to automatically generate test data systematically covering realistic SQL queries. To achieve this goal, we model test data generation for SQL queries as a search-based problem. We devise and evaluate three different approaches, based on random search, biased random search, and genetic algorithms. We define a fitness function that can be used to steer such algorithms towards an optimal solution containing the appropriate data to reach a given coverage target.

We offer an implementation of our approach in the open source tool EvoSQL, which we use to evaluate the applicability of our approach in practice. Using the 2,135 queries collected from real life systems, we demonstrate that (1) EvoSQL can handle the full SQL standard, including subqueries, JOINs, and string handling; (2) achieves 100% coverage for 98% of our queries; and (3) manages to do so in 2-15 seconds in most of the cases.

Our work paves the way for systematic and automated unit, integration, and regression testing of SQL queries. Furthermore, we anticipate interesting future research in terms of further optimizing the performance and effectiveness of the genetic algorithm, taking EvoSQL beyond standard SQL, making use of all integrity constraints of the schema to speed up the search, and letting the genetic algorithm take information obtained from the system's source code into account as well.

RIGHTSLINK

REFERENCES

[1] 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* 16, 3 (2006), 175–203.

[2] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[3] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13 (Feb. 2012), 281–305.

[4] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 341–352.

[5] Edmund K Burke, Graham Kendall, et al. 2005. *Search methodologies*. Springer.

[6] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. EvoSQL. https://github.com/SERG-Delft/evosql. (2018). The precise version used in this paper can be found at https://github.com/SERG-Delft/evosql/releases/tag/icse.

[7] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-Based Test Data Generation for SQL Queries: Appendix. https://www.zenodo.org/record/1166023. (2018). https://doi.org/10.5281/zenodo.1166023

[8] Nitesh V Chawla. 2003. C4. 5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure. In *Proceedings of the ICML*, Vol. 3.

[9] David Chays, John Shahid, and Phyllis G Frankl. 2008. Query-based test generation for database applications. In *Proceedings of the 1st international workshop on Testing database systems*. ACM, 6.

[10] Kalyanmoy Deb and Debayan Deb. 2014. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing* 4, 1 (2014), 1–28.

[11] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 151–162.

[12] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 151–162.

[13] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.

[14] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[15] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[16] Gordon Fraser, Andrea Arcuri, and Phil McMinn. 2015. A memetic algorithm for whole test suite generation. *Journal of Systems and Software* 103 (2015), 311–327.

[17] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. 2000. *Database system implementation*. Vol. 654. Prentice Hall Upper Saddle River, NJ:.

[18] David E Goldberg and Kalyanmoy Deb. 1991. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms* 1 (1991), 69–93.

[19] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (2002), 256–290.

[20] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. 2008. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*. 1–10.

[21] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O Laleye, and Sarfraz Khurshid. 2008. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 238–247.

[22] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.

[23] Chengkai Li and Christoph Csallner. 2010. Dynamic symbolic database application testing.. In *DBTest*.

[24] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[25] Phil Mcminn, Chris J Wright, and Gregory 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.

[26] Phil McMinn, Chris J Wright, Cody Kinneer, Colton J McCurdy, Michael Camara, and Gregory 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.

[27] Kai Pan, Xintao Wu, and Tao Xie. 2011. Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. ACM, 4.

[28] A. Panichella, F. Kifetew, and P. Tonella. 2017. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1.

[29] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 1–10.

[30] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 254–265.

[31] Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA.

[32] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.

[33] S.N. Sivanandam and S.N. Deepa. 2007. *Introduction to genetic algorithms*. Springer Science & Business Media.

[34] María José Suárez-Cabal, Claudio de la Riva, Javier Tuya, and Raquel Blanco. 2017. Incremental test data generation for database queries. *Automated Software Engineering* (2017), 1–37.

[35] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: Automated test generation for database applications via mock objects. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 289–292.

[36] Edward Tsang. 1993. *Foundations of constraint satisfaction*. Academic Press.

[37] Javier Tuya, María José Suárez-Cabal, and Claudio De La Riva. 2010. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability* 20, 3 (2010), 237–288.

[38] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

RIGHTSLINK