Search-Based Test Data Generation for SQL Queries

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jeroen Castelein
born in Limmen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Search-Based Test Data Generation for SQL Queries

Author:	Jeroen Castelein
Student id:	4408721
Email:	`j.castelein@student.tudelft.nl`

Abstract

Software testing is an important, well-researched field. With the majority of modern-day applications using relational databases to manipulate their data, it is crucial that database interactions are tested as well. This is a complex task to perform manually, and thus researchers have been attempting to tackle this problem by means of automated test data generation. In their studies, they apply constraint-based techniques using SAT solvers to generate the test data. However, these techniques have known limitations such as solving subqueries.

In this thesis, we present a novel search-based approach that uses a Genetic Algorithm to generate test data for SQL queries, which overcomes the limitations of previous research. We provide an implementation of our approach, EvoSQL. In our implementation, we instrument a real database to extract all the information necessary for the fitness function. By doing so, we support all queries using standard SQL syntax. We evaluate our approach on $2,135$ queries from 4 real-world systems, of which EvoSQL is able to cover over 96% fully.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. F. Aniche, Faculty EEMCS, TU Delft
	M.Sc. M. Soltani, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bozzon, Faculty EEMCS, TU Delft
	Dr. P.A.N. Bosman, CWI

Contents

List of Figures

Chapter 1

Introduction

Database-centric applications are essential to modern-day society. Such applications range from companies managing customer relations, to hospitals managing medical data. It is therefore vital for software developers to carefully assure that these applications behave as expected. One way of doing this is by means of testing.

In practice, these applications manage and manipulate large amounts of data that are stored in a relational database. The program communicates with the database through SQL queries. These queries allow developers to retrieve, insert, update, and delete data in several different ways. As a consequence, important business rules are expressed as SQL queries. Therefore, SQL queries should be as properly tested as program code, so that developers can detect defects, perform regression testing, and apply safe refactoring.

To test SQL queries, developers need data that exercise all parts of a SQL query, such as predicates or subqueries. However, the combination of multiple tables together with query constraints, join operations, and subqueries makes the generation of such data a difficult and time-consuming task.

There has been some research that attempt to tackle this problem. For example, Tuya et al. [37] propose a coverage criterion for SQL queries that exercises all the different constraints of the query, similarly to a branch coverage criterion [15] in traditional software testing. Suárez-Cabal et al. [35] present an constraint-based approach to generating test databases that maximize the aforementioned coverage criterion. Shah et al. [32] take a different approach, and generate test databases with the purpose of killing SQL mutants, which are also proposed by the authors. Other researchers [10, 29] focus on maximizing program code coverage rather than SQL coverage, by means of generating test databases based on SQL queries that are executed by the program under test.

However, these approaches are unable to generate test databases for a wide range of SQL queries, such as queries with string constraints and subqueries. Their limitations come from the fact that they use SAT solvers that are unable to model these query constraints, such as Alloy [18] and Choco [20].

We present a novel search-based approach to generating test data for SQL queries that overcomes these limitations. Instead of relying on SAT solvers, our approach collects data from query executions in a fully-functioning database, and uses this data to guide a Genetic Algorithm. In practice, using a real database enables our approach to support all SQL constructs, such as string functions, subqueries, date/time operations, and aggregations.

We provide an implementation of our approach, named EvoSQL. Given a SQL query and its respective database schema, EvoSQL generates test data that covers each coverage rule. To evaluate our approach, we executed EvoSQL on $2,135$ queries extracted from 4 real-world systems, one of them being from an industry partner. EvoSQL achieves full coverage on $2,053$ of these queries, i.e., 96.2%, completely outperforming a random search baseline. For the queries that were not completely covered, we show that EvoSQL still achieves high partial coverage, and that this increases when we give it a larger time budget.

We make the following contributions in this thesis.

- A novel search-based approach for generating test data for a given SQL query, which supports all SQL constructs.
- EvoSQL, a Java tool that implements our approach, and it is available as open source.
- An empirical study on queries from 4 real-world systems showing that EvoSQL achieves full coverage for 96% of them, and high coverage for the other queries.
- A replication package containing the queries and schemas used in our evaluation, excluding one closed-source system.

The thesis is organized as follows. Chapter 2 contains background information. Chapter 3 explains our novel approach. Chapter 4 presents EvoSQL, our tool that implements the approach. Chapter 5 contains the details on the empirical study performed, as well as the results of the study. Chapter 6 contains discussions on the results and the state of EvoSQL. Chapter 7 contains the conclusions.

Chapter 2

Background

This section introduces the most important fields in background knowledge necessary with regards to this thesis. Section 2.1 introduces search-based software testing, the field of applying search-based algorithms to generate tests for software , in this thesis it is applied to generate test data for SQL queries. Section 2.2 explains the syntax of a SQL `SELECT` query, and how it applies constraints to data in the database. This definition is a crucial basis to the fitness function. Section 2.3 presents SQLFpc, a coverage criterion that tests SQL queries' constraints. In our implementation, SQLFpc is used to generate the coverage rules. Finally, Section 2.4 discusses related work in the field of SQL test database generation, as well as search-based test data generation.

2.1 Search-Based Software Testing

Search-based software engineering (SBSE) is an approach in which search-based optimization (SBO) algorithms are used to address software engineering problems [17]. SBO algorithms are used for search problems in which an optimal solution is sought in a search space of candidate solutions.

In the case of software engineering, a prime example is searching for input to a method so that a certain code branch is covered. This example brings us to the most researched field in SBSE, search- based software testing (SBST). In this subfield, the goal of the algorithm is automating a testing task [26]. Mostly, this involves generating test data, as is the goal of this research, as well as test cases. EvoSuite is an example of generating test cases with the goal of covering branches. More specifically, they generate whole test suites for Java applications that aim to have full branch coverage, with a minimal amount of test cases [12].

Before applying SBSE to a search problem, there are two prerequisites often referred to as the key ingredients of SBSE.

1. A representation for the problem's candidate solutions.

2. A *fitness function* that measures the 'goodness' of a candidate solution.

The representation of candidate solutions is often easily done in SBSE. If the goal is to generate a Java test case, then a possible representation is a Java method with some lines of code in it. The problem representation is also called the *encoding scheme*.

The fitness function is specific to the problem and must be defined so that a candidate solution can be measured deterministically. It must be able to distinguish more optimal from less optimal candidate solutions. For the Java test cases, an example fitness function would be to calculate whether a branch is covered, and if not, how close it is to being covered based on the evaluation of the *if statement*.

Coverage criteria

Branch coverage is a possible *coverage criterion* that the creator of a search algorithm might find important. A coverage criterion is the goal of the algorithm and is represented through the fitness function. Another example of a coverage criterion is execution time, where the search algorithm should aim to keep execution time of test cases as short as possible.

On its own, this criterion would find empty test cases to be optimal. Therefore, it should be combined with other criteria. This is called a multi-objective approach. In this approach, the optimal solution is identified using multiple fitness functions, one per criterion. A popular way of comparing the candidate solutions when using multiple fitness functions is using Pareto optimality. The Pareto optimal set contains all candidate solutions for which there is no way of improving any fitness values of one criterion without decreasing the fitness of another criterion [24]. In practice this means that when the search algorithm has found a test case covering the desired code branch, it will continue searching for test cases that have a shorter execution time.

In this thesis, we use a coverage criterion that focuses on covering constraints in a SQL query, presented in Section 2.3.

Algorithms

With the two main ingredients in place, the next step in SBSE is selecting an algorithm. The most widely used algorithms are Hill Climbing, Simulated Annealing, and Genetic Algorithms.

The first two are local search algorithms. Hill Climbing starts by picking a random spot in the search space. As the name suggests, it searches for a nearby spot to move to which is higher (fitter) than the current spot. This repeats until no higher spot can be found and the algorithm has found a local optimum. Because this may not be the global optimum, the algorithm can use a restarting mechanism to find fitter solutions. Simulated Annealing is similar to hill climbing, except that it is allowed to move to lower (less fit) spots. This behavior is controlled by a probability value, the temperature, which decreases with time. By doing this, some local optima can be avoided. Especially those that are not much fitter than their surroundings.

The Genetic Algorithms are global search algorithms. They are inspired by Darwinian evolution and the concept of survival of the fittest. Multiple candidate solutions are evaluated simultaneously. Each one of these is called an *individual*, and the set of individuals is called the *population*. Each individual represents a candidate solution and consists of genes. Each gene can be altered by the evolutionary operators: *crossover* and *mutation*. In crossover, two parent individuals are combined to create two new children individuals, where both children have a mix of their parents genes. In mutation, a single child individual has some genes altered.

In genetic algorithms, the population evolves generation after generation, with the aim of finding the optimal individual. In each generation, new individuals are created by selecting parent individuals from the population using a *selection* operator. This could be done randomly, or heuristically by giving fitter individuals a higher probability of being selected. After selecting two parent individuals, two children are created and the crossover and mutation operators are applied with some probability. After applying the evolutionary operators, the fitness of each of the new children is calculated. Finally, now that the new children are measured and added to the population, the next generation's population is decided. This is another selection process, where often the population's fitness is taken into account, hence survival of the fittest.

2.2 SQL Query Syntax

The goal of this thesis is to generate test data for SQL queries. A *query specification*, more commonly referred to as a `SELECT` query, is used to request some view of the data in a database. A `SELECT` query contains a `SELECT` keyword, a *select list*, and a *table expression* [1]. The table expression is constructed through a set of clauses (`FROM`, `WHERE`, `GROUP BY`, `HAVING`), and is a combination of data sources and predicates that may specify restrictions on the output. Using the output of the table expression, the select list defines how to write the data to output.

Thus, for the query to output at least one row, the table expression must output at least one row. In this section each of its clauses (`FROM`, `WHERE`, `GROUP BY`, `HAVING`) and how they affect the output is explained. First, the notion of SQL predicates is explained, as they are used throughout the clauses.

Predicates

In SQL, a predicate is a combination of an operator and some operands. There are two types of predicates: base and derived predicates. Base predicates are those that do not use any other predicates (e.g., $column_1 = 5$). Derived predicates have logical operators (`AND, OR, NOT`) and use other predicates in their operands.(e.g., $column_1 = 5$ `AND` $column_2 = 10$).

Each predicate can be evaluated returning a three-valued logic result; `TRUE`, `FALSE` or `UNKNOWN`. The `UNKNOWN` value occurs when the result is not defined by the operator (e.g., $5 >$ `NULL`).

Definition 1. The result of a predicate evaluation *pe* is a three-valued logic value. $\text{res}(pe) \in$ `TRUE,FALSE,UNKNOWN`.

FROM

The FROM clause specifies the table to take data from. It contains a FROM keyword followed by a list of *table references*. Each table reference has one *primary table* and optionally a list of *joined tables*. A joined table is a combination of a table and a join operator (FULL, LEFT, RIGHT, or INNER). The syntax for a JOIN clause is:

[FULL/LEFT/RIGHT/INNER] JOIN [<table>/(<subquery>)] AS <alias> ON *predicate*

Each join operator has a different functionality.

- **INNER** If the join operator is INNER, combining the table reference t up to this joined table jt, the output of this joined table o only contains the combinations of rows between t and jt for which the join predicate is satisfied. If the join predicate is never satisfied, o is empty.
- **FULL / LEFT** If the join operator is FULL or LEFT, and joins the joined table jt onto the table reference t up to this joined table, the output o contains at least one row for each row $r \in t$. If for a row $r \in t$ there are n rows in jt for which the join predicate is TRUE and $n > 0$, the output o contains n rows with r and each of the matching rows in jt.
- **FULL / RIGHT** If the join operator is FULL or RIGHT, a similar join is done as in with the LEFT join. The difference is that t is now joined onto jt, which means that the matching is done based from the rows in jt. Hence the output will have at least all rows from jt, plus the possible joined rows from t.

When all table references in the FROM clause are processed they are cross-joined onto each other into a single output. A cross join T,U has an output in which every row in T is present in combination with every row in U, known as the cartesian product. If one of the tables is empty, the output table is also empty.

WHERE

The optional WHERE clause contains a predicate and uses it to filter the output table from the FROM clause. Only the rows for which the predicate evaluates to TRUE remain in the output table.

GROUP BY

This optional GROUP BY clause specifies a grouped table by applying the given *grouping specification* to the result of the previously specified clause.

HAVING

The optional HAVING clause contains a predicate and uses it on the grouped table from the GROUP BY clause, and if there is none the implicit GROUP BY clause with an empty grouping specification. The clause eliminates groups that do not satisfy the predicate.

2.3 SQL Coverage Criteria

It is based on another criterion, Full Predicate Coverage [28], which is a form of masked MC/DC (Modified Condition / Decision Coverage) [8]. SQLFpc generates multiple **coverage rules**, given a SQL query and a database schema. Each coverage rule is represented as a SQL query. A coverage rule is covered by a database if there some output when the rule's query is executed on the database.

Definition 2. A coverage rule for the SQLFpc criterion is represented as a SQL query. It is covered by some database if, when executing the query on it, there is at least one row of output.

To collect the coverage rules, SQLFpc provides a web service, hence an internet connection is required to use it. For the simple example query below, SQLFpc generates two coverage rules:

Query:

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

Coverage rules:

1. `SELECT * FROM Product WHERE (Category = 'Toy')`
2. `SELECT * FROM Product WHERE NOT(Category = 'Toy')`

To cover rule 1, there should be a row in the table `Product` in which the column `Category` is filled with the value `'Toy'`. To cover rule 2, there should be a row in which the same column is filled with any other value.

In practice, having a test suite in combination with a test database covering each coverage rule is useful because it exposes the full functionality of the query, by returning each combination of predicates that it can return, and similarly it is able to detect changes in the functionality of the query. If some rule should not return data but the query has changed and this data is now returned, the test suite is able to identify this, prompting the developer to change either the query or the test data.

In this thesis, the coverage rules are used to generate data. Because of how SQLFpc generates coverage rules, there are a few restrictions on what kind of queries appear in these rules. These restrictions reduce the scope of queries to cover for our implementation and removes ambiguities.

- **Join operator** For any join operator in a query, SQLFpc generates some rules such that the join must be satisfied using an `INNER JOIN` in the coverage rule. Alternatively, it generates rules such that the join must not be satisfied from either side using a `RIGHT JOIN` in one coverage rule, and a `LEFT JOIN` in another. To make sure they are not satisfied, SQLFpc adds predicates to the `WHERE-clause` that requires the values that are joined to be `NULL`. There may be some variations when there are multiple join operators in a query, or when the columns used in the join predicate are nullable, but this concept remains. This means that whenever there is an `INNER JOIN`, the data should satisfy this join. Likewise, whenever there is a `RIGHT/LEFT JOIN`, it should *not* satisfy this join.

- **Union operator** If two or more `SELECT` queries are concatenated by means of a `UNION` operator, SQLFpc generates rules for each `SELECT` query individually. This means that a coverage rule query can never contain `UNION` operators in the outer query.

2.4 Related work

Our approach aims to generate test data for SQL queries, using a search algorithm. To this end, we split the related work into two main subjects: Automated test data generation for SQL databases, and search-based test data generation. We leave a comparison between our proposed and related work to the discussion in Chapter 6 of this thesis.

2.4.1 Automated Test Data Generation for SQL Databases

Previous work on generating data for SQL databases follows different paths. Firstly, there are studies that generate test databases dependent on the database schema only.

Gray et al. [14] aim to generate a large databases quickly, by using parallel algorithms and execution. The authors accomplish this by generalizing sequential database generation to a point where it can be executed in parallel. They state that by doing this, they turn a two-day task into a one-hour task.

Bruno and Chaudhuri [5] present a flexible and scalable framework for database generation. They motivate their efforts as they see in research that data generators are often developed ad-hoc. They introduce a special language DGL (Data Generation Language) that can be used to generate synthetic data distributions. They conclude that the data generators run using DGL are efficient.

McMinn et al. [27] use a search-based approach to generate test data based only on a relational database schema. This test data serves to increase confidence in the schema's correctness by being able to catch mutations. They use a test generator based on Korel's Alternating Variable Method [23].

Secondly, in other studies, they focus on generating data based on one or more SQL queries, as well as the schema constraints.

Chays et al. [7] present the AGENDA tool set that generates tests for database applications. Based on the database schema, a query, and test heuristics selected by the user, it fills the test database with boundary values or other values following the heuristics. Using the expected behavior as stated by the user, they generate test data and tests to verify the behavior.

A similar approach is taken by Khalek et al. [21, 22] with their tool ADUSA. They use the Alloy SAT solver to populate test databases given a schema and a query. They also generate a test with the expected result of the qiven query. The goal of these tests is to test a database system, rather than a query as is done in this thesis. In their evaluation, they introduce bugs in real database systems to test, and they state their generated tests are effective at finding these bugs.

Another similar approach is taken by Binnig et al. [4] and their tool: QAGen. They use symbolic execution to define constraints; given a query, a database schema, and some preferences defined by the user. These constraints are then given to a constraint solver which generates a test database. In their evaluation, the efficiency of their approach is analyzed for different database sizes.

Shah et al. [32] use a constraint solver to generate test data with the goal of killing SQL mutants. They limit their research to queries with `WHERE`-clauses and join operations. The mutations considered are limited to the join type, comparison operators, and aggregation operators. They state that the datasets they generate are small and easily analyzable, while being able to effectively detect mutants.

Suárez-Cabal et al. [35] present an incremental approach to generating test databases for multiple queries simultaneously with their tool QAGrow. They use the SQLFpc coverage criterion, and incrementally generate data for each coverage rule. The constraints of generating new data are a combination of the current database state and the constraints in the next coverage rule. A constraint solver then generates the data which is added to the database. They evaluate their effectiveness on a set of simple and complex queries, for which they achieve 100% and 99%, respectively. However, they do not solve queries that contain subqueries, or string constraints.

Pai et al. [29] use Dynamic Symbolic Execution (DSE) to reduce manual effort in test generation for software applications. Their focus is on code coverage of the application, and how database interactions are handled. Their approach aims to cover two criteria, BVC and CACC, which implement using the DSE test-generation tool Pex[36]. In later research [30], they bypass the use of database states to run the tests by synthesizing database interactions in order to extract the environment constraints. They combine these with the program-execution constraints and use Pex to generate data. From their evaluations, they conclude their approach is able to achieve higher program code coverage than other approaches.

Binnig et al. [3] present a new take on generating test databases with their technique called Reverse Query Processing. Given a query, a database schema, and a query result, they generate a test database that produces this result. The goal of this test database to test a DBMS. To generate this data, they build a reverse relational algebra tree and use it to go backwards in the query flow, starting from the result table and ending up with the test database. In their evaluation, they generate databases with up to 86 million rows.

2.4.2 Search-Based Test Data Generation

In the area of search-based test data generation, research is mostly focused on testing program code. The goals of these tests range from straightforward code coverage, to causing or replicating program crashes.

Korel [23] introduces the concept of using actual values and execution of the program under test when generating test data. Function minimization search techniques are used to heuristically find a solution. These techniques are sped up by using dynamic data flow analysis.

Wegener et al. [38] apply evolutionary algorithms to the field of structural testing, with the goal of generating test data that maximizes a chosen coverage criterion (e.g., branch coverage). They introduce the idea of an approximation level, which indicates how many code branches still need to be executed as desired in order to reach the branch that is the current goal of the evolutionary algorithm. They provide a tool environment which provides test case automation for software. They evaluate their tool on some real-world examples and are able to achieve 100% coverage on all of them, outperforming a random testing approach.

Fraser and Arcuri [11, 12] present a search-based approach for generating whole test suites for Java applications, EvoSuite. They use a Genetic Algorithm that optimizes entire test suites towards a chosen criterion (e.g., branch coverage), while minimizing their size. They evaluate their whole test suite approach, in comparison with a single branch approach, on a case study on 19 open source libraries, with a total of 3,165 classes. They conclude that whole test suite generation achieves higher coverage, while producing smaller test suites.

Mao et al. [25] present a multi-objective search-based testing approach to exploratory testing for Android applications, called Sapienz. In their approach, they are able to instrument the app under test at multiple levels, depending on their accessibility. Their search algorithm uses NSGA-II [9] to find solutions, maximizing code coverage and fault revelation, while minimizing the size of the fault-revealing test sequences. They evaluate their approach on several open source apps, and state Sapienz always outperforms its competitors. Additionally, they evaluate the usefulness of their tool by executing it on the top 1,000 Google Play apps, in which they found 558 unique crashes in 329 of the apps.

Soltani et al. [34] use the EvoSuite approach to generate test cases that reproduce software crashes with their tool, EvoCrash. Given a crash stack trace, they use a Guided Genetic Algorithm to search for test cases that reproduce the crash and the stack trace. They evaluate their approach on 50 bugs from three open source projects. They state their approach is able to replicate 82% of these crashes, outperforming three other cutting-edge approaches.

Chapter 3

Approach

The goal of this approach is to generate test data that maximizes test coverage on SQL queries. Given a query and a database schema, data is generated to cover each coverage rule as defined by the coverage criterion. A coverage rule is covered by some data if there is output when executing the rule's query on the data. For each coverage rule, a single-objective Genetic Algorithm (GA) is used to find the data within a set time budget. The use of a GA is chosen over other search algorithms because of the size and complexity of the search space. After the GA is executed for each coverage rule, the test data for the given query are the datasets for each covered coverage rule. In the rest of this chapter, the GA is presented with the goal of finding data for a single coverage rule.

To find data that covers a rule, the GA uses evolutionary operators in combination with the two main ingredients of any heuristic search algorithm: 1) a problem representation (definition of an individual) and 2) a fitness function. As the objective is to find data such that a query has output, each individual in the GA's population represents a database state on which the query can be executed.

Definition 3. An individual in the GA contains a set of tables T. This set contains all tables used by the coverage rule's query. Each table $t \in T$ contains a non-empty set of rows. Each row in t has a typed value for each column, where the type is taken from the schema of table t in the supplied database schema. Nullable columns may also be set to `NULL`.

The overview of the GA is shown in Figure 3.1. The GA first initializes a population of random individuals, scattered throughout the search space. It then uses a novel fitness function that calculates how close an individual is to being a solution for the coverage rule. Using the output of the fitness function, the GA narrows down the search space by selecting fit individuals and altering them using crossover and mutation. The GA continues to narrow down the search space until a solution is found, or the time budget is depleted.

To speed up the process of finding solutions, the GA applies guidance techniques. Amongst these are seeding strategies, which use knowledge about the query to add likely values (seeds) to a seeding pool.

In Section 3.1, the initialization procedure is explained. In Section 3.2, the fitness function is presented. Then, the selection operators and the evolutionary operators are presented in Section 3.3 and Section 3.4, respectively. Finally, the guidance techniques used to speed up the GA are presented in Section 3.5.

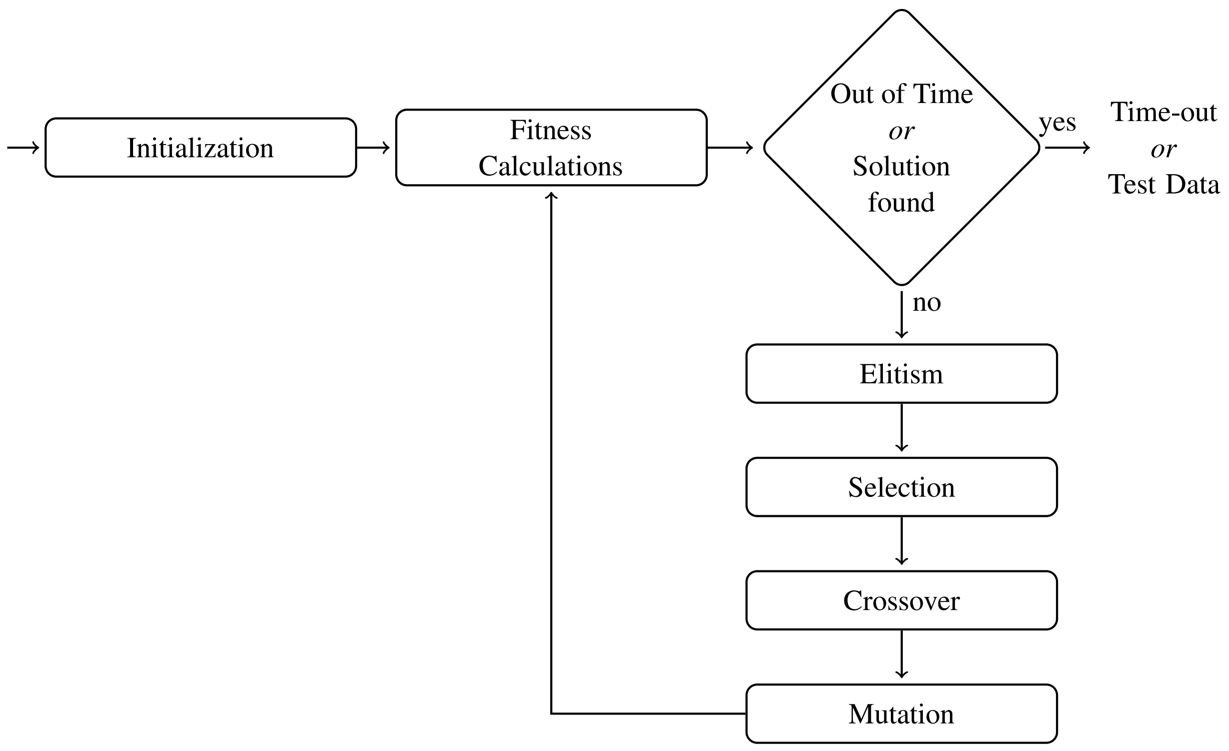

Figure 3.1: Overview of the GA

3.1 Initialization

At the start of the GA, the population is filled with the first generation of individuals. New individuals are generated table by table. For each table, a random number of rows, between 1 and 4, is generated. These boundaries were chosen as we observed that, in practice, no more than 4 rows are needed in one table. For many coverage rules, a single row per table is enough for the GA to find a solution. However, when aggregate operations are used, more than one row may be necessary for the solution to be found. Also, limiting the number of rows in a table improves the runtime of the fitness function.

Within each row, a typed value for each column is generated. This typed value is randomly generated. If the column is nullable, then the value generated may also be `NULL`. With a constant probability α, a randomly selected value from the seeding pool for the column's type is used instead. This probability is chosen so that it is high enough to be effective by seeding useful values, but not too high, which would cause an overrepresentation of seeded values, which may cause the GA to converge in a local optimum. The seeding techniques that fill the seeding pool are presented in Section 3.5.

3.2 Fitness Function

The fitness function serves to identify whether an individual is a solution, and to distinguish the fitter individuals in a population. A solution is found if the coverage rule's query has some output when executed on the individual's data. This is true when the last clause that potentially removes data (`FROM` (including `JOIN`s), `WHERE` or `HAVING`) is satisfied at least once, following the definition of a query specification (see Section 2.2). In this case, clause satisfaction means that, after applying all the predicates and other rules that come with this clause, there is at least one row of output.

Definition 4. An individual is a solution to a coverage rule if, when executing the rule's query on the individual's data, the following three statements are true:

- The `FROM` clause has some output. For this to happen, the tables, subqueries and joins in the `FROM` clause must have output.
- If there is a `WHERE` clause, the combination of its predicates must evaluate to `TRUE` for at least one row.
- If there is a `HAVING` clause, the combination of its predicates must evaluate to `TRUE` for at least one row.

This brings us to the definition of the fitness function.

Definition 5. Given an individual *i*, the fitness function for the current coverage rule is as follows:

$$f(i) = d_{FROM}(i) + d_{WHERE}(i) + d_{HAVING}(i) \tag{3.1}$$

The fitness of the individual is the sum of the fitness of the three important clauses in the query, where an individual with fitness 0 successfully covers the coverage rule. The fitness for each of these clauses represent how close the clause is to being satisfied with a number between 0 and 1. This way, none of the clauses can outweigh the other, and an individual satisfying more clauses is always fitter than an individual satisfying less.

The rest of this section describes how query executions are compared, starting with clause satisfaction in Section 3.2.1 and predicate satisfaction in Section 3.2.2.

3.2.1 Clause Satisfaction

The query is used by the fitness function to analyze how close individuals are to having output and thus satisfying the query. A clause in the query can only be evaluated if the previous clause has output. Therefore, if one individual has evaluated a clause further than another individual, it is closer to the solution, hence fitter. If both individuals evaluate up to the same clause, this clause's evaluation must be analyzed further to find out which individual is fitter. This subsection details how clause satisfaction is calculated for the three important clauses: `FROM`, `WHERE`, and `HAVING`.

FROM clause

If the `FROM` clause is not satisfied this means that the output table is empty. Because all table references (a combination of tables, subqueries and joins; see Section 2.2) in the `FROM` clause are cross-joined, each table reference must have output in order to satisfy the `FROM` clause. The more table references have output, the fitter the individual.

Definition 6. The fitness of the `FROM` clause execution is the normalized sum of the fitness of its table references. Normalization is done to bound the return value of $d_{FROM}(i)$ to 1. In the following formula TRS is the set of table references in the `FROM` clause, and d_{TR} is the fitness function for a table reference.

$$d_{FROM}(i) = \frac{1}{|TRS|} \sum_{tr \in TRS} d_{TR}(tr, i) \tag{3.2}$$

Table reference

The fitness of a table reference is based on it having some output, or otherwise how close it is to having output. The output of a table reference depends on its primary table and joined tables. Namely, the primary table and all inner join tables must have output. These tables are referred to as the *essential tables*, and each of these tables is either a base table, i.e., a table in the database schema, or a subquery. Another factor influencing the output of a table reference are the predicates of the inner join operations. If an inner join predicate does not evaluate to `TRUE` for any row, the table reference has no output. Thus, the fitness of a table reference depends on the fitness of its essential tables as well as the fitness of its inner joins' predicates.

Right join An exception to the rule is when there are right joins in the query. If there is a right join, all rows from the right join's table are always present in the output, joining the tables up to this right join onto them. This means that only the last right join and the joined tables afterwards can affect whether a table reference has output. Therefore, all tables up to the last right join are not important for the fitness of the table reference. Furthermore, the set of essential tables now starts from the last right joined table, and contains all inner joins after it.

Our approach to not using the tables up to a right join is inspired by how SQLFpc generates coverage rules (see Section 2.3).

Definition 7. In a table reference, the set of essential tables ET is a subset of the primary table and all joined tables. If there is a right join in the query, the essential tables are the last right joined table and all inner joined tables after it. If there are no right joins in the query, the essential tables are the primary table and all inner joined tables after it. The inner joined tables in the essential tables are referred to as the *essential inner joins* (EIJ).

For a table reference to have output, all essential tables must have output and the join predicates of all essential inner joins must be satisfied for at least one row. A table is not evaluated if the table before it has no output. Therefore, fitter executions evaluate more essential tables.

If two individuals evaluate up to the same table, the fitness of this last evaluated table is used to distinguish the fitter individual. If the evaluated table is a subquery, the fitness of the subquery execution is used. Also, if the evaluated table is part of an inner join, the fitness of the join's predicates is used. The fitness function for predicates is presented in Section 3.2.2.

Definition 8. The fitness of a table reference tr's execution for an individual i is a combination of the essential tables ET having output, and the predicates of the essential inner joins EIJ being satisfied. They are normalized to bound the return value of d_{TR} between 0 and 1.

$$d_{TR}(tr,i) = \frac{1}{2|ET|}\left(\sum_{et \in ET} d_T(et,i)\right) + \frac{1}{2|EIJ|}\left(\sum_{j \in EIJ} d_J(j,i)\right) \tag{3.3}$$

Definition 9. The fitness of a single table evaluation is one if it is not evaluated, because the previous essential table has no output. The fitness is zero if it is a base table, and thus always has output, and otherwise depends on the fitness of the subquery execution. The fitness function $f(i)$ is used recursively to calculate the fitness of the subquery, using only the clauses of the subquery.

$$d_T(et,i) = \begin{cases} 1, & \text{if } et \text{ is not evaluated} \\ 0, & \text{if } et \text{ is a base table} \\ \dfrac{f(i)}{1+f(i)}, & \text{otherwise } (et \text{ is a subquery}) \end{cases} \tag{3.4}$$

Definition 10. The predicates of an essential inner join are only relevant if they are evaluated, which only happens if the inner join's table has some output. If the predicates are not evaluated, the fitness is one. Otherwise, it is the fitness of the evaluation of the join's predicates (join-pred), where d_P is the fitness function for predicates.

$$d_J(j,i) = \begin{cases} 1, & \text{if join-pred is never evaluated} \\ d_P(\text{join-pred}, i), & \text{otherwise} \end{cases} \tag{3.5}$$

WHERE & HAVING clauses

A `WHERE` or `HAVING` clause is not satisfied if its combination of predicates does not evaluate to `TRUE` for any row. The fitness of these clauses' evaluations depends on whether it is evaluated at all, and otherwise on the fitness of the predicates.

Definition 11. The fitness of the `WHERE` clause execution for an individual i is one if the `FROM` clause is not satisfied, zero if there is no `WHERE` clause, or the fitness of the `WHERE`-predicates otherwise. Again, d_P is the fitness function for predicates.

$$d_{WHERE}(i) = \begin{cases} 1, & \text{if } \texttt{FROM} \text{ clause has no output} \\ 0, & \text{if there is no } \texttt{WHERE} \text{ clause} \\ d_P(\texttt{WHERE}\text{-predicates}, i), & \text{otherwise} \end{cases} \tag{3.6}$$

The fitness of the HAVING clause execution for an individual i is identical to the WHERE clause except for requiring all clauses up to it to be satisfied.

$$d_{HAVING}(i) = \begin{cases} 1, & \text{if FROM/WHERE/GROUP BY clauses have no output} \\ 0, & \text{if there is no HAVING clause} \\ d_P(\text{HAVING-predicates}, i), & \text{otherwise} \end{cases} \tag{3.7}$$

3.2.2 Predicate Satisfaction

The calculation of a predicate's fitness is based on its evaluations. A predicate is evaluated once for each row in the table the predicate applies to. Each evaluation has an operator, values of its operand(s) and a three-valued logic result (TRUE, FALSE, UNKNOWN, see Section 2.2). In this section, we define the fitness function for predicates by means of a distance function.

The purpose of the distance function is, given a predicate evaluation, to return how close it is to evaluating to TRUE. If the predicate evaluation result is TRUE, the predicate is satisfied and there is no need for a distance. After the predicate has been evaluated on all rows, the fitness of the predicate corresponds to the fittest row's evaluation, i.e., with the lowest distance.

To calculate the distance of derived predicates (AND, OR, or NOT, see Section 2.2), the distance of their children is recursively calculated. These distances are then combined differently depending on the operator, which we explain near the end of this section.

A derived predicate may have a NOT operator. If such a predicate's child evaluates to TRUE then the predicate itself evaluates to FALSE and thus, it is not satisfied. In this case, the distance of the NOT to being TRUE is equal to the distance of its child to being FALSE. Therefore, to be able to give a correct distance to the NOT predicate, the distance function must also return a distance value when a predicate evaluates to TRUE. This way, if the parent of a predicate evaluating to TRUE has a NOT operator, the parent will have the inverted value of its child, so that the value represents how close this parent predicate is to evaluating to TRUE.

To summarize, the result of the distance function, given a predicate evaluation, resembles a two-way distance. If it is less than zero, the outcome of the evaluation is TRUE and the distance function returns the distance to the evaluation outcome to being FALSE. On the other hand, if it is greater than zero, the outcome of the evaluation is FALSE and the distance function returns the distance to the evaluation outcome to being TRUE. The distance is never zero, as negating it would have no effect on the boolean result value it represents.

Definition 12. The distance function dist takes a predicate evaluation pe and returns a number representing how far the result is from being TRUE or FALSE. Here, $\text{res}(pe)$ returns the result of the predicate evaluation, as described in Section 2.2.

- If $\text{res}(pe) = \text{TRUE}$, $\text{dist}(pe) < 0$ and the more negative it gets, the further the predicate is from being FALSE.

- If $\text{res}(pe) = \texttt{FALSE}$, $\text{dist}(pe) > 0$ and the higher it gets, the further the predicate is from being `TRUE`.

Definition 13. The fitness of a predicate given an individual i is optimal if the distance of one of the predicate evaluations PE is negative, i.e., the result of one of the evaluations is `TRUE`. If not, the lowest distance of the evaluations is the best. This distance is normalized to be bound between 0 and 1.

$$d_P(predicate, i) = \begin{cases} 0, & \text{if } \exists pe \in PE \ \text{dist}(pe) < 0 \\ \min\limits_{pe \in PE} \dfrac{\text{dist}(pe)}{1 + \text{dist}(pe)}, & \text{otherwise} \end{cases} \tag{3.8}$$

In case a base predicate is evaluated to `UNKNOWN`, e.g., when evaluating whether a numeric value is larger than `NULL`, the distance of the operation is set to `MAX_DISTANCE`.

Definition 14. Given a base predicate evaluation bpe, if $\text{res}(bpe) = \texttt{UNKNOWN}$, then $\text{dist}(bpe) = \texttt{MAX\_DISTANCE}$.

Comparison operators

The comparison operators in SQL are EQUAL (=), NOT EQUAL (<>), GREATER THAN (>), GREATER THAN OR EQUAL (>=), LESS THAN (<) and LESS THAN OR EQUAL (<=). Each of these operators compares two values. *lv* and *rv* represent the left and right value, respectively. If at least one of the two values is `NULL`, the result is `UNKNOWN` and the distance returned is `MAX_DISTANCE`.

Each of the comparison operators rely on a function calculating the difference between the two operands. The difference function is defined for four types: number, boolean, string, and date.

Definition 15. The difference function $\text{diff}(lv, rv)$ returns a number representing the difference between two values *lv* and *rv* of the same type. If the difference is negative, *lv* is smaller than *rv*. If the difference is positive, *lv* is greater than *rv*. If the difference is zero, they are equal.

- **Number.** The difference between two numbers is straightforward; the left value minus the right value.

- **Boolean.** Boolean values can only be 0 (`FALSE`) or 1 (`TRUE`). Their difference is the same as with numbers.

- **String.** The difference between two string values is the absolute difference in length plus the absolute ordinal character difference per position in the two strings. If the left value is lexicographically smaller, the difference is negated to be a negative number.

- **Date.** The difference between two date values is the sum of the difference for each numerical calendar part: year, month, day, hour, minute, second, millisecond.

The following formulas calculate the distance per comparison operator using the difference function.

EQUAL If the two values are equal, the outcome is `TRUE` and the distance is minus ε. Here, ε is an arbitrarily small value, used to avoid zero. Otherwise the distance is the absolute difference between the two values.

$$\mathrm{dist}(lv = rv) = \begin{cases} -\varepsilon, & \text{if } \mathrm{diff}(lv, rv) = 0 \\ \left|\mathrm{diff}(lv, rv)\right|, & \text{otherwise} \end{cases} \tag{3.9}$$

NOT EQUAL If the two values are not equal, the outcome is `TRUE` and the distance is the negative absolute difference. If they are equal the distance is ε.

$$\mathrm{dist}(lv <> rv) = \begin{cases} -\left|\mathrm{diff}(lv, rv)\right|, & \text{if } \mathrm{diff}(lv, rv) \neq 0 \\ \varepsilon, & \text{otherwise} \end{cases} \tag{3.10}$$

GREATER THAN If the difference is positive, the outcome is `TRUE` and the distance is the negative difference. Otherwise the difference is negative or zero, and the distance is ε plus the absolute difference.

$$\mathrm{dist}(lv > rv) = \begin{cases} -\mathrm{diff}(lv, rv), & \text{if } \mathrm{diff}(lv, rv) > 0 \\ \varepsilon + \left|\mathrm{diff}(lv, rv)\right|, & \text{otherwise} \end{cases} \tag{3.11}$$

GREATER THAN OR EQUAL If the difference is positive or zero, the outcome is `TRUE` and the distance is the negative difference minus ε. Otherwise the difference is negative and the distance is the absolute difference.

$$\mathrm{dist}(lv >= rv) = \begin{cases} -\mathrm{diff}(lv, rv) - \varepsilon, & \text{if } \mathrm{diff}(lv, rv) \geq 0 \\ \left|\mathrm{diff}(lv, rv)\right|, & \text{otherwise} \end{cases} \tag{3.12}$$

LESS THAN If the difference is negative, the outcome is `TRUE` and the distance is the same as the difference. If the difference is positive or zero, the outcome is `FALSE` and the distance is ε plus the difference.

$$\mathrm{dist}(lv < rv) = \begin{cases} \mathrm{diff}(lv, rv), & \text{if } \mathrm{diff}(lv, rv) < 0 \\ \varepsilon + \mathrm{diff}(lv, rv), & \text{otherwise} \end{cases} \tag{3.13}$$

LESS THAN OR EQUAL If the difference is negative or zero, the outcome is `TRUE` and the distance is the difference minus ε. If the difference is positive, the outcome is `FALSE` and the distance is the same as the difference.

$$\mathrm{dist}(lv <= rv) = \begin{cases} \mathrm{diff}(lv, rv) - \varepsilon, & \text{if } \mathrm{diff}(lv, rv) \leq 0 \\ \mathrm{diff}(lv, rv), & \text{otherwise} \end{cases} \tag{3.14}$$

Subqueries in predicates

The operands predicates (lv, rv) may be subqueries. These subqueries can have two possible outcomes: 1) the subquery returns a single value, that is used by the predicate, 2) the subquery returns no values, i.e., a `NULL`.

For the former case, the distance of the predicate evaluation is calculated as aforementioned. For the latter case, as the subquery returns `NULL`, the distance of the predicate evaluation would be `MAX_DISTANCE`; however, the fitness function of the subquery can be used to give a more representative distance to these predicate evaluations. Therefore, the distance is calculated as the `MAX_DISTANCE` plus the fitness of the subquery execution.

This way, subqueries that do return values are fitter than subqueries that do not. Also, by using the fitness function of the subquery, the distance of a predicate evaluation is lower (i.e., closer to be evaluated to `TRUE`) when the subquery evaluation is fitter.

Definition 16. If either side of a comparison operator is a subquery, and this subquery has no output, the distance of the predicate evaluation is based on the fitness of the subquery evaluation.

$$\text{dist}(pe) = \texttt{MAX\_DISTANCE} + f(i), \quad \begin{array}{r}\text{where } f(i) \text{ is the fitness function for the subquery,}\\ \text{given the current individual } i\end{array} \tag{3.15}$$

SQL operators

There are some special SQL operators that the distance function takes into account. More specifically: `BETWEEN`, `IS (NOT) NULL`, `IN`, `LIKE` and `EXISTS`.

BETWEEN The `BETWEEN` operator returns `TRUE` when with value v and bounds lb and ub (lower and upper, respectively), $lb <= v <= ub$. In SQL this is equivalent to $lb <= v$ `AND` $v <= ub$.

$$\text{dist}(v \ \texttt{BETWEEN} \ lb \ \texttt{AND} \ ub) = \text{dist}(lb <= v \ \texttt{AND} \ v <= ub) \tag{3.16}$$

IS NULL The `IS NULL` operator returns `TRUE` when the value v under inspection is `NULL`, and `FALSE` otherwise.

$$\text{dist}(v \ \texttt{IS NULL}) = \begin{cases} -1, & \text{if } v \text{ is } \texttt{NULL} \\ 1, & \text{otherwise.} \end{cases} \tag{3.17}$$

IS NOT NULL The `IS NOT NULL` operator is equivalent to `NOT` (`IS NULL`).

$$\text{dist}(v \ \texttt{IS NOT NULL}) = \text{dist}(\texttt{NOT} \ (v \ \texttt{IS NULL})) \tag{3.18}$$

IN The IN operator compares a single value on the left lv with a list of values on the right RVS. This list of values may be a constant list written in the query or a subquery result. It returns `TRUE` if at least one of the values in RVS is equal to lv. If the list is empty due to an empty subquery result, the fitness of the subquery evaluation defines the distance. If the list is not empty and the result is `FALSE`, the distance is the minimum absolute difference between lv and all values in RVS. This difference is normalized so that it is always lower than the case of an empty subquery result.

$$\text{dist}(lv\ \texttt{IN}\ RVS) = \begin{cases} -1, & \text{if } \exists rv \in RVS \mid rv = lv \\ 1 + f(i), & \text{if } |RVS| = 0, \text{ where } f(i) \text{ is the fitness of the} \\ & \qquad \text{subquery for the current individual } i \\ \min\limits_{\forall rv \in RVS} \dfrac{|\text{diff}(lv, rv)|}{1 + |\text{diff}(lv, rv)|}, & \text{otherwise} \end{cases} \tag{3.19}$$

LIKE The LIKE operator performs pattern matching on a string. The pattern supports using wildcards: % for matching a string of any length (including zero) and _ for matching a single character. If the pattern matches the string completely, it returns `TRUE`. Otherwise, it performs a recursive calculation of the distance between the string and the pattern. This distance is the sum of characters left in both the string and the pattern after the furthest possible matching.

$$\text{dist}(v\ \texttt{LIKE}\ pattern) = \begin{cases} -1, & \text{if pattern is matched, and } v \text{ has no characters left} \\ n + m, & \text{otherwise, pattern has } n \text{ characters left to match, and } v \text{ has } m \text{ left} \end{cases} \tag{3.20}$$

EXISTS The EXISTS operator takes a subquery as a parameter. This subquery is executed and if the subquery returns a value the operator evaluates to `TRUE`. Otherwise it evaluates to `FALSE` with a distance equal to the fitness of the subquery execution for the current individual i.

$$\text{dist}(\texttt{EXISTS}(subquery)) = \begin{cases} -1, & \text{if subquery has output} \\ f(i), & \text{otherwise} \end{cases} \tag{3.21}$$

Logical operators

SQL has three logical operators, `AND`, `OR` and `NOT`. The predicates using these operators are derived predicates as they have 2, 2, and 1 child predicates respectively. To calculate the predicate evaluation's distances, they use the distance(s) of their children.

AND An AND evaluation has two children, c_1 and c_2. If both evaluate to `TRUE` (with negative distances) then the AND evaluates to `TRUE`. The distance to become `FALSE` is the distance closest to zero (smallest negative) of the children, as only one of them has to be `FALSE`. If the AND evaluates to `FALSE` or `UNKNOWN`, the distance is the sum of distances of the children evaluating to `FALSE` or `UNKNOWN`, as they must all be `TRUE`.

$$\text{dist}(c_1 \;\texttt{AND}\; c_2) = \begin{cases} \max(\text{dist}(c_1), \text{dist}(c_2)), & \text{if res}(c_1) = \texttt{TRUE} \text{ and res}(c_2) = \texttt{TRUE} \\ \text{dist}(c_2), & \text{if res}(c_1) = \texttt{TRUE} \text{ and res}(c_2) \neq \texttt{TRUE} \\ \text{dist}(c_1), & \text{if res}(c_1) \neq \texttt{TRUE} \text{ and res}(c_2) = \texttt{TRUE} \\ \text{dist}(c_1) + \text{dist}(c_2), & \text{otherwise} \end{cases} \tag{3.22}$$

OR An OR evaluation has two children, c_1 and c_2. If both evaluate to `FALSE` (with positive distances) then the OR evaluates to `FALSE`. If neither evaluate to `TRUE` and at least one evaluates to `UNKNOWN`, the OR evaluates to `UNKNOWN`. In either of these cases, the distance to become `TRUE` is the distance closest to zero of the children, as only one of them has to be `TRUE`. If the OR evaluates to `TRUE`, the distance is the sum of distances of the children evaluating to `TRUE`.

$$\text{dist}(c_1 \;\texttt{OR}\; c_2) = \begin{cases} \min(\text{dist}(c_1), \text{dist}(c_2)), & \text{if res}(c_1) \neq \texttt{TRUE} \text{ and res}(c_2) \neq \texttt{TRUE} \\ \text{dist}(c_2), & \text{if res}(c_1) \neq \texttt{TRUE} \text{ and res}(c_2) = \texttt{TRUE} \\ \text{dist}(c_1), & \text{if res}(c_1) = \texttt{TRUE} \text{ and res}(c_2) \neq \texttt{TRUE} \\ \text{dist}(c_1) + \text{dist}(c_2), & \text{otherwise} \end{cases} \tag{3.23}$$

NOT A NOT evaluation has a single child c. The distance of the NOT is the inverted distance of the child, unless the child result is `UNKNOWN`, in which case the NOT result will also be `UNKNOWN` and the distance is the same as that of the child.

$$\text{dist}(\texttt{NOT}\; c) = \begin{cases} \text{dist}(c), & \text{if res}(c) = \texttt{UNKNOWN} \\ -\text{dist}(c), & \text{otherwise} \end{cases} \tag{3.24}$$

3.3 Selection Operators

The initial population may not contain a solution. In this case, as Figure 3.1 shows, the GA evolves a new population until a solution is found, or the time budget is depleted. To make sure new generations improve their fitness, selection operators are applied to select which individuals may stay and reproduce.

At the beginning of each new generation, the old (parent) population and the new (children) individuals are redivided to the new population, so that the population size remains the same as the original population size N. *Elitism* performs the division, with the aim of maintaining a fit population. With this (partially) new population, *Tournament Selection* is applied N times to select parent individuals for reproduction.

It is important to balance these operators so that the convergence rate is not too slow, causing the GA to take longer at finding the solution, nor too fast, causing the GA to converge prematurely to a local optimum [33]. To this end, the two selection operators combined must ensure that while converging to a solution, thus prioritizing fitter individuals, they must also make sure to give the lesser fit individuals a fighting chance.

3.3.1 Elitism

Given an old population of size N and an equally sized set of new individuals, elitism decides which individuals survive. The elitism applied by our GA selects the N fittest individuals from the combined set of new individuals and the old population. Since all old and new individuals are included, elitism is ensured [9].

This approach to elitism differs from typical implementations, where elitism makes sure to keep some of the fittest individuals from the old population in the new population. By using our proposed approach, the fittest individuals are never lost.

3.3.2 Tournament Selection

To select parent individuals for reproduction, the GA applies *Tournament Selection*. In this operator, a tournament with size n selects n individuals randomly from the population to compete in a tournament competition [13]. The individuals are paired up, and the fitter one wins each pairing. The winner of the tournament is the fittest individual in the n selected individuals, and is added to the mating pool, to reproduce with other individuals in the mating pool. This is repeated until the mating pool is filled with N individuals, as many as are in the population.

We chose Tournament Selection over other selection operators (e.g., Rank Selection) for its simplicity and its ability to slow down the convergence rate. This balances out with the elitism applied by the GA.

3.4 Evolutionary Operators

Using the individuals selected for reproduction, N new individuals (children) are created using the *crossover* and *mutation* evolutionary operators. From the mating pool, two parent individuals are taken at a time. The GA may apply the crossover operator to the parents, generating two children that have some combination of their parents data. If the crossover operator is not applied, the parents are copied to create two children that are identical to their relative parent. Afterwards, the GA applies mutation to both new children.

3.4.1 Crossover

The crossover operator generates two children individuals C_1 and C_2 based on two parents P_1 and P_2. Given each individuals has a set of tables T, a random number x is chosen from $\left[1,|T|\right]$. The first child C_1 is then created containing all tables but the xth from the first parent P_1, plus the xth table from the second parent P_2. Likewise, the second child C_2 contains all tables but the xth from parent P_2, plus the xth table from parent P_1.

By swapping tables between individuals, predicates comparing columns between the swapped table and the other tables could be closer to evaluating to `TRUE`. Such predicates typically appear in SQL join operations.

3.4.2 Mutation

The mutation operator is applied to each individual. In this operator, mutation is applied at table level, as well as row level. Each table in an individual is mutated with a probability based on the amount of tables in the individual. If a table is mutated, there are four actions that may be applied to the table, in the given order.

1. **Delete.** Delete a random row from the table. This action aims to remove unnecessary rows, so that further mutation is more likely to be applied to the rows that are necessary.

2. **Duplicate.** Duplicate a random row in the table. This action is useful when the query applies a grouping, and needs multiple rows for a group.

3. **Add.** Add a newly generated row to the table. This action keeps the GA from getting stuck in local optima.

4. **Mutate.** Mutate a row in the table. This action serves to change the data in a row to satisfy the query's predicates. The paragraph below describes the process of mutating a row.

Row mutation When mutating a row, each mutable column is mutated with a probability based on the amount of mutable columns in the row. If there are seeded values available for this column it will seed a value with probability β. Column values that are `NULL` mutate to a random value for its type. If the column is nullable it mutates to `NULL` with probability γ. Like in initialization (see Section 3.1), the probabilities are chosen so that they are effective, but do not cause convergence to a local optimum. Finally, if none of the previous mutations are done, a linear typed mutation is applied.

Typed column mutation The typed mutation changes a column value to a new value nv that is close to the old value ov such that the difference $\text{diff}(ov, nv)$ between these two values is small. This is small to prevent oscillations around optima, and to ensure that fitter individuals are closer to the solution. Like the difference function, column mutation is defined for four types: number, boolean, string, and date.

- **Number.** If the number is an integer, a random value between 0 and 10 is added or subtracted. If the number is a real number, a small value from a gaussian distribution with mean 0 is multiplied with *ov*. The result of this multiplication is added to *ov*. Finally, the real number is truncated to a random amount of decimals between 0 and 16.
- **Boolean.** For boolean values, the new value is the negated old value.
- **String.** For string values, each character is mutated with some small probability. For each character, one of three actions is taken.
  1. A random character is added after the character
  2. The character is removed
  3. The character's ordinal value is changed. A random value between 0 and 10 is added or subtracted.
- **Date.** For date values, each calendar part is mutated with some small probability. A calendar part is mutated by adding or subtracting a random value between 0 and 10.

3.5 Guidance Techniques

To speed up the GA, domain knowledge is applied to guide the GA to finding fitter individuals sooner. Before the initialization phase, static query analysis is performed extracting metadata for the GA to use.

Mutable columns Each individual contains data for each column in each table used in the query. However, not all columns are relevant for finding the solution. The GA can limit its search space by ignoring columns that are not used by any predicates. A naive approach is used to decide which columns are used: 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 knowledge about the coverage rule's query. The GA uses two seeding pools, one for values to seed into columns, and the other for seeding individuals from the previous GA's population.

Constants In a query, there may be predicates comparing a column value to a constant value. Each constant in the query is added to the column seeding pool for the value's type.

Column equalities 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. When seeding for $column_1$, the values in $column_2$ are added to the seeding pool, and vice versa.

Previous population The coverage rules generated for a query are all similar to the original query and to each other. This indicates that the data covering the rules may look similar as well. Therefore during the initialization phase of the GA, some individuals from the previous coverage rule's execution are cloned before the new individuals are generated. If at least one solution was found for the previous rule, one of them is always cloned. The rest of the previous population is cloned with some probability. This random sampling of the previous population is chosen over weighted sampling because it is less susceptive to local optima.

Chapter 4

EvoSQL

We provide an implementation of the approach, EvoSQL. EvoSQL takes a query, a database schema, and a time budget, and returns test data for each coverage rule of the query. First, the tool collects the coverage rules for this query and schema by invoking the SQLFpc web service. EvoSQL then distributes its time budget to each obtained coverage rule fairly, and proceeds to execute the GA on each coverage rule consecutively. Each coverage rule's GA executes until either a solution is found, or its time budget is depleted. If a solution is found for one coverage rule, the remaining time budget is distributed across the coverage rules that have not yet been solved. This continues until all coverage rules are covered, or the time budget is depleted.

Our approach requires the query for each coverage rule to be completely interpreted and executed against an individual. Such functionalities are already present in any SQL relational database. Therefore, we instrument an existing, open source database and follow its execution flow to extract all the necessary data for the fitness function.

In this chapter, we present how we instrumented our database of choice, HSQLDB, and what challenges we encountered while instrumenting it. In Section 4.1 we describe the implementation of the fitness function into the database. In Section 4.2 we present database optimizations techniques that we encountered and disabled. Finally, in Section 4.3 we present how we handle exceptions that are thrown by the database, and how this may influence the fitness function.

4.1 Implementation of the Fitness Function

The database we instrument is HSQLDB[1], a Java relational database engine that supports the latest SQL standards. HSQLDB can be executed completely in-memory, which is ideal as an individual in EvoSQL has little data and the fitness for it is only calculated once, after which the data is removed from the database.

Firstly, because EvoSQL uses the SQLFpc criterion, it can benefit from using the characteristics of SQLFpc coverage rules. In Section 4.1.1, we present how the fitness function is adapted to make better use of these characteristics. Secondly, the way HSQLDB executes

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

queries is slightly different from how our fitness function interprets queries. In Section 4.1.2 we clarify these differences and how we adapted our implementation to match.

4.1.1 Coverage Rule Restrictions

Our instrumentation takes advantage of how SQLFpc handles join operations. In general, if there is a left or right join in a query, it is complex to decide whether the join should be satisfied or not. Although these joins are not part of the fitness function, it could be that some predicate in the `WHERE` clause requires the join to be satisfied, or the opposite. Without knowing which to do, the best option for our implementation would be to try both satisfying the join and not. This causes a split in the search space which may cause convergence in a local optimum.

Fortunately, as presented in Section 2.3, the coverage rules SQLFpc generates always have inner joins if the join must be satisfied, and left or right joins when it must **not** be satisfied. This is enforced by a predicate in the `WHERE` clause that requires the joined table's columns to be `NULL`, meaning the join must not be satisfied. In our implementation we explicitly aim to never satisfy any left or right join operations by attaching negative fitness if such a join is satisfied.

4.1.2 HSQLDB Query Execution Flow

To implement the fitness function, we use the HSQDLB query execution flow as a guideline. In HSQLDB, predicates from the `WHERE` clause are evaluated while processing the `FROM` clause, rather than after evaluating the entire `FROM` clause. When a table in the `FROM` clause is processed, HSQLDB evaluates all predicates that can be evaluated using the data that is gathered so far, and have an individual effect on removing rows. For example, in the query below, the predicate `t1.value = 50` is evaluated before the inner join operation.

```
SELECT *
FROM t1
INNER JOIN t2 ON t1.id = t2.id
WHERE t1.value = 50
```

For the fitness function this means that the fitness calculations of the `FROM` and `WHERE` clauses are intertwined, and that when calculating the fitness for a table reference, the predicates evaluated by HSQLDB are also taken into account.

Our implementation of the fitness function adds the fitness of these predicates to the fitness calculation of the corresponding essential table, and removes their fitness from the fitness calculation of the `WHERE`-clause. If the predicate belongs to a table that is not an essential table, these can only be predicates that require left or right joined table's columns to be `NULL`, as discussed in the previous section.

4.2 Database Optimizations

Typically, databases are optimized to minimize the processing power used when executing a query. HSQLDB is no exception. To assert no data evaluations are skipped when executing a query, we disabled these optimizations.

Indexing Databases allow tables to be indexed so that some queries may be optimized. The index is used to reduce the set of rows that will be evaluated by the query, as it is able to exclude the rows that do not satisfy predicates of indexed columns without individually evaluating them. Excluding rows in the instrumented database causes the execution to be incomplete. Therefore, we disable indexing so that all rows are always evaluated.

AND/OR optimization In many programming languages programmers can rely on lazy evaluations of the logical operators `AND` and `OR`. In lazy evaluation, when the left expression in an `AND` operation evaluates to `FALSE`, the right expression will not be evaluated. Likewise with an `OR`, when the left expression evaluates to `TRUE`. In languages such as Java, this is often used to check if objects exist before attempting to access them. In SQL there is no such thing as objects and lazy evaluation is solely an optimization. If lazy evaluation is used it prevents the fitness function from extracting all predicate evaluations. Therefore, we disable lazy evaluations in the instrumented database so that both sides of `AND` and `OR` operations are always evaluated.

4.3 Exception Handling

Databases may throw exceptions when executing a query. Most exceptions are either database or query related. These indicate that the database cannot execute the query and for the GA, there is no point in trying other individuals, as the individual is not the cause of the exception. In this case, our implementation informs the user of the exception.

There are also exceptions that are related to the data not matching the query. Thus, they are caused by the individual, and are important when calculating a fitness value. Namely, if an individual causes an exception to be thrown, it is worse than any other individual that does not throw an exception.

To handle these exceptions, we expand the fitness function so that individuals that cause exceptions to be thrown are less fit than those that don't:

$$f(i) = \begin{cases} 4, \text{ if an exception is thrown} \\ d_{FROM}(i) + d_{WHERE}(i) + d_{HAVING}(i), \text{ otherwise} \end{cases} \tag{4.1}$$

The exceptions caused by the individual that we encountered and handle accordingly are the following two:

- **Division by zero** If the query being executed contains a division, in which the denominator depends on at least one column value, there is a chance that the data in this column causes the denominator to evaluate to 0. When this happens a division by zero occurs, throwing an exception. Because this exception is caused by the data in the column, and other data may not cause this exception, the individual is unfit.

- **Cardinality violation** A cardinality violation occurs when there is a subquery in the query which should return a single value, but returns multiple. This could be a field in the `SELECT` clause, or one of the values in a comparison operator. This requires the subquery to select a single column which is not aggregated, and is valid (yet unsafe) SQL syntax as the subquery may be designed to only return 1 row of data. A cardinality violation is the result of such an unsafe query in combination with data that causes the subquery to return multiple values. Therefore, the individual is unfit for this query.

Chapter 5

Empirical Study

The goal of this study is to evaluate the effectiveness of EvoSQL. To do this we have collected $2,135$ queries from 4 software systems, one of them being from one of our industry partners. We investigate the query coverage that EvoSQL is able to achieve across these systems. We compare our results to a pure random search algorithm as the baseline in this study. To evaluate our approach we formulate four research questions.

- **RQ$_1$:** *What is the difference in the query coverage achieved by EvoSQL and the baseline?* In this research question, we evaluate the effectiveness of EvoSQL compared to the baseline.

- **RQ$_2$:** *What are the causes when EvoSQL does not achieve 100% coverage?* In this research question, we search for key attributes that are difficult for EvoSQL to achieve full coverage on.

- **RQ$_3$:** *What is the performance of EvoSQL?* In this research question, we analyze the performance of EvoSQL on our evaluation set.

- **RQ$_4$:** *How do different time budgets impact the effectiveness of EvoSQL?* In this research question, we investigate how much coverage can be achieved with different time budgets. Also, we investigate whether there are queries for which EvoSQL cannot improve the coverage, and what may be the cause.

The rest of this chapter is organized as follows. We describe how the queries are gathered and analyze some of their SQL properties in Section 5.1. In Section 5.2 we explain what the random search baseline is. We describe the experimental procedure for each research question in Section 5.3. Finally, we present and discuss the results in Section 5.4, followed by the threats to validity in Section 5.5.

5.1 Context of the Study

We evaluate our approach on $2,135$ queries taken from four real world software systems:

1. Alura[1] is a closed source e-learning platform. It uses Hibernate as a layer between application and database. Hibernate generates SQL queries based on the data requested by the application.

2. EspoCRM[2] 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.

3. SuiteCRM[3] 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.

4. ERPNext[4] is an end-to-end business solution that manages business information (ERP: enterprise resource planning). It is built on top of the Python & JavaScript framework *Frappé* and uses MariaDB.

To collect the queries for evaluation, we execute the test suites of each system, and extract the queries directly from the database logs. Table 5.1 shows the total amount of queries collected per system. In this table, we also describe the amount of queries as we prepare the set for evaluation. This preparation is a cleaning procedure ensuring that each query is executable, and that no repetitive work is done.

1. **Usable queries.** Not all queries can be executed, as some may have bad syntax, or syntax that is specific to the system's database. The usable queries are the queries that both SQLFpc as well as HSQLDB can execute.

2. **Unique queries.** Many of the queries are very similar to each other, only differing by some constant values. This can be the result of a method being executed with different parameters. 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. There is no difference in solving these queries for the GA, as column `a` is either randomly filled, or taken from the seeding pool, which changes depending on the constant. To get the unique queries, we group all usable queries by the query strings excluding constants and randomly select one query per group.

3. **Evaluation queries.** Finally, some of the unique queries may have no predicates or other constraints. These are queries that request some view on all data from a table. For such a query, there are no coverage rules and there is nothing for EvoSQL to do. The queries for which SQLFpc generates no coverage rules are removed to form the final set of queries being used in the evaluation, the *evaluation queries*. We evaluate $2,135$ queries in total.

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

Application	Total queries	Usable queries	Unique queries	Evaluation 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 5.1: Queries collected per application

5.1.1 Query properties

The query properties of the evaluation queries are presented in Table 5.2. The first four properties represent the amount of occurrences of this property in the query. The used columns property is the amount of columns used by the predicates of a query, as defined in Section 3.5. Finally, the coverage rules property is the amount of coverage rules that SQLFpc generates for a query.

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	-	-
Used columns	60	1369	457	127	43	26	20	13	20
Coverage rules	-	656	382	408	346	114	107	51	71

Table 5.2: Number of queries with a certain amount of a property

There is a correlation between the coverage rules and the other query properties, meaning that the amount of coverage rules is a good measure of the amount of constraints to solve, and thus a measure of general complexity of a query. To show the correlation, we compare amount of coverage rules with the sum of the other query properties in Figure 5.1. In this figure, we also show a regression line making the correlation clear. The Spearman correlation is 0.95 (p-value < 0.01). Most of the data points are in the left bottom corner with few coverage rules and properties. From this distribution, we may conclude that most of the queries generate few coverage rules, hence have few constraints to solve.

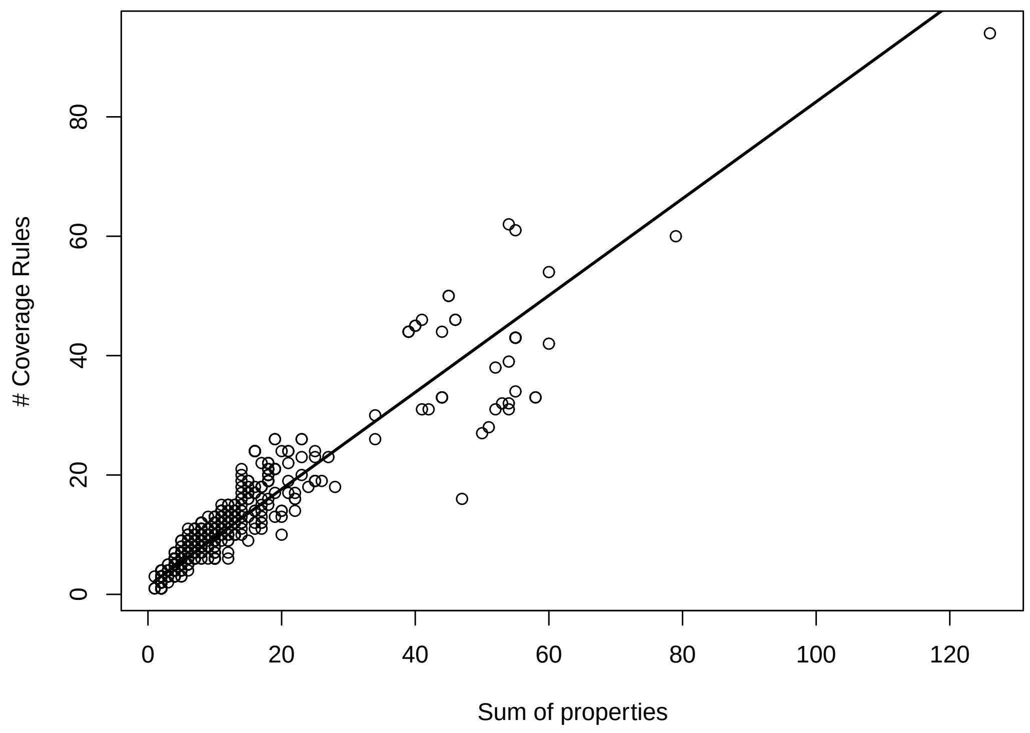

Figure 5.1: Scatterplot of the correlation between the amount of SQLFpc coverage rules and the sum of the other query properties.

5.2 Baseline

To be able to quantify the effectiveness of EvoSQL, we compare it to pure random search. This is a common randomized search heuristic to compare against in which each generated individual is completely random [19]. This means that while adhering to the encoding scheme, the data in each column is a random value. Random search is commonly used as a baseline to compare to more advanced algorithms with the aim of beating it, although in areas of testing it has been shown that random search can be more effective than some advanced algorithms [16]. To make the comparison fair, this algorithm has the same time-based termination criterion as EvoSQL.

5.3 Experimental Procedure

To answer the research questions, both EvoSQL and the baseline are executed multiple times. The systems are split up into many processes, of which 5 are executed concurrently.

The machine the experiment runs on has 40 cores @ 2.30GHz from 4 Intel Xeon E5-2650 v3 CPUs, and 125GB RAM.

The parameter settings used to run EvoSQL are based on default parameter values used in the field of evolutionary algorithms [12, 31]. These default values are shown to give reasonable results for the entire search space [2].

- **Population size.** We use a population size of 50.
- **Tournament size.** We use a tournament size of 4.
- **Mutation.** We mutate any table $t \in T$ in an individual with probability $1/|T|$. For each table mutation the probability is $1/6$ for the mutation actions; deleting, duplicating, and adding a row. Because we have found in practice that rows in one table are not often combined, row mutation is always applied.
- **Row mutation.** In row mutation, each mutable column $mc \in MC$ in the row is mutated with probability $1/|MC|$.
- **NULL.** The probability of setting a column value to `NULL` is 0.01.
- **Crossover.** The probability of applying crossover is 0.75.
- **Population cloning.** The probability of an individual from the population of the previous execution (i.e., the previous coverage rule) to be cloned into the next execution's population is 0.6.
- **Seeding.**
  - The probability of seeding on initialization is 0.5.
  - The probability of seeding during mutation needs to be very low as otherwise it is likely to overrule the smaller mutations. It is set to 0.01 so that there is some small probability of getting a value from the seed pool.
- **Termination criterion.** Besides stopping when all coverage rules are covered, the algorithm has a time budget of 30 minutes for one query.

To answer RQ$_1$, both EvoSQL and the baseline are executed on all evaluation queries 10 times. We execute them 10 times to have higher confidence in the results, given the randomized nature of the algorithms. Collecting the rules from SQLFpc requires a constant internet connection as well as the SQLFpc web service to be online. To ensure no errors occur due to a faulty connection or SQLFpc downtime, we store the coverage rules beforehand. During evaluation, we mock the web service so that the correct coverage rules are returned whenever they are requested.

Throughout the research, we have experienced that some of the coverage rules that SQLFpc returns are infeasible. In most of these cases this is caused by some combination of predicates that cannot be satisfied. Because detecting these programmatically is out of scope for this study, we manually analyze each rule that EvoSQL can not solve across all

10 executions, which are 429 rules from 101 queries. All coverage rules that are found to be infeasible are removed from the results.

Across the 10 executions, the query coverage is averaged for each query. We say a query is successfully covered if the average query coverage is 100%, which means that all coverage rules were covered in each execution. If one rule was not covered in any of the 10 executions, we say that EvoSQL *failed* to cover that query. For each system, we investigate the query coverage achieved, as well as the number of successes and failures. By using the failed queries as they are defined, we have a worst-case view on the results. Because the coverage rule distribution is skewed, we also analyze the queries grouped by the amount of coverage rules. This shows how the algorithms handle tougher queries with more coverage rules.

To answer RQ_2, we use the results from RQ_1 and analyze the coverage rules that EvoSQL failed to cover at least once in the 10 executions. Similar to the queries, we use the terminology of a success only if a rule is covered in all 10 executions.

We programmatically extract query properties from each rule and use them to fit a decision tree classifying whether a rule is likely to fail or not. The properties we extracted and their descriptions are listed in Table 5.3. We chose these properties as they are important SQL constructs. The last few properties that are more specific are chosen from our experience with EvoSQL and solving queries with these properties. Each of these properties are extracted from the `FROM` clause onwards, as the items in the `SELECT` clause are not relevant to covering a coverage rule (see Section 2.2). We use Weka[5], a machine learning toolkit, to create the decision tree. From the decision tree, we extract the important query properties to conclude what makes EvoSQL fail a coverage rule.

We take the following steps to select useful data and generate a decision tree that classifies whether a coverage rule is likely a failed rule or a successful rule, given its properties:

1. Because our classes, failed rules and successful rules, are not evenly balanced, we apply SMOTE (Synthetic Minority Over-sampling TEchnique) to generate extra data points for the smaller class [6]. This way, both classes have equal size for a better classification.

2. We select the top 5 properties ranked by information gain to the class. This limits the properties that the decision tree will use to those that are important, which prevents overfitting.

3. In the last step we create a decision tree on the top 5 properties using the J48 classifier, which generates a pruned C4.5 decision tree. We use 10-fold cross-validation to calculate the accuracy of the decision tree.

To answer RQ_3, we analyze EvoSQL's execution time per query. Once more, we group the queries by their coverage rules to analyze whether this has an effect on the execution time. For the execution time, we analyze the mean, the standard deviation, as well as the quantiles.

[5]Weka - http://www.cs.waikato.ac.nz/ml/weka/

Property	Explanation
Tables	Number of tables used
Predicates	Number of base predicates, this does not include `AND`, `OR`, and `NOT` operators
Inner Joins	Number of inner joins
Left Joins	Number of left joins
Right Joins	Number of right joins
Subqueries	Number of subqueries
Aggregate Functions	Number of aggregate functions (`MIN, MAX, SUM, AVG, COUNT`)
Other Functions	Number of non-aggregate functions (e.g., `DATENOW, IFNULL`)
Used Columns	Number of used columns, as described in Section 3.5
WHEREs	Number of `WHERE` clauses
GROUP BYs	Number of `GROUP BY` clauses
HAVINGs	Number of `HAVING` clauses
StringEQs	Number of string equality predicates
DateEQs	Number of date equality predicates
EXISTs	Number of EXIST predicates
LIKEs	Number of LIKE predicates
CASEs	Number of CASE expressions
IFNULLs	Number of IFNULL functions
Outcome	Success if the coverage rule was covered each evaluation, Failure otherwise

Table 5.3: Query properties retrieved from the coverage rules' queries

To answer RQ_4, we re-evaluate the failed queries with a time budget of two hours instead of half an hour. Like in RQ_1, this evaluation is executed 10 times. We compare the coverage rates between the two evaluations. Then, we manually inspect the coverage rules of queries that did not improve, to inspect why EvoSQL may be unable to solve them.

We then combine the results of the evaluations run with a half an hour time budget and a two hour time budget. We use the average time budget used per coverage rule in a query to calculate how many coverage rules would be covered on average, given a different time budget. If a query was re-evaluated with a two hour time budget, those results are used in favor of the evaluation with a half an hour time budget. This is done because we have more information about these queries, and the queries that were not evaluated in this re-evaluation were already fully covered by EvoSQL in half an hour.

We calculate the query coverages per time budget from 1 to 120 minutes (i.e., 2 hours).

Using this data, we present the average query coverage progression against an increasing time budget grouped by coverage rules as done in Section 5.1.1.

Replication package

We provide an open source replication package[6]. This package contains our implementations of both algorithms, EvoSQL and the baseline. It also contains the queries and schemas from all systems but the closed-source application Alura, which opens up for a comparison with other tools.

5.4 Results

We concluded from the manual analysis on these rules that 127 out of the total $12,991$ coverage rules are infeasible, and they are excluded from the results. Throughout the results, we group queries by their number of coverage rules. For comparison reasons, we still group queries based on their initial number of coverage rules, as we showed the correlation between them and the other query properties in Section 5.1.1. For example, a query which had 9 coverage rules of which 8 are feasible still appears in the group with 9 - 10 coverage rules.

RQ$_1$. What is the difference in the query coverage achieved by EvoSQL and the baseline?

The high level results of both algorithms are shown in Figure 5.2. Overall, EvoSQL achieves 100% coverage for the vast majority of the queries. The only system for which this is not true is EspoCRM. We show a more detailed insight into the results in Table 5.4 and Table 5.5. In these tables, we grouped queries based on the amount of coverage rules like in Section 5.1.1. For each system and algorithm (B = baseline, E = EvoSQL) we show two metrics. The first table contains the number of successful queries n_s vs the number of failed queries n_f written as n_s/n_f. As mentioned before, a query is only successful if all its rules are covered in all executions. The second table contains the average query coverage of the queries which the approaches could not fully cover at least once.

From this data we conclude that EvoSQL outperforms the baseline. With regards to successfully covering queries, the baseline has low effectiveness with few rules, and never succeeds with more than 8 rules. This contrasts with EvoSQL, where up to 8 rules it gets very few failed queries. While the number of successful queries starts dropping for EvoSQL after 10 rules, the average coverage of the failed queries remains high. The lowest being 57% on the group of 12 EspoCRM queries with at least 21 coverage rules. This means that on average, many coverage rules are still covered for these complex queries. Notably, baseline achieves close to 25% average coverage on the same EspoCRM queries. The relatively high coverage by the baseline for this amount of rules only occurs with EspoCRM queries.

[6]Available as soon as this thesis is published as a paper (est. December 2017)

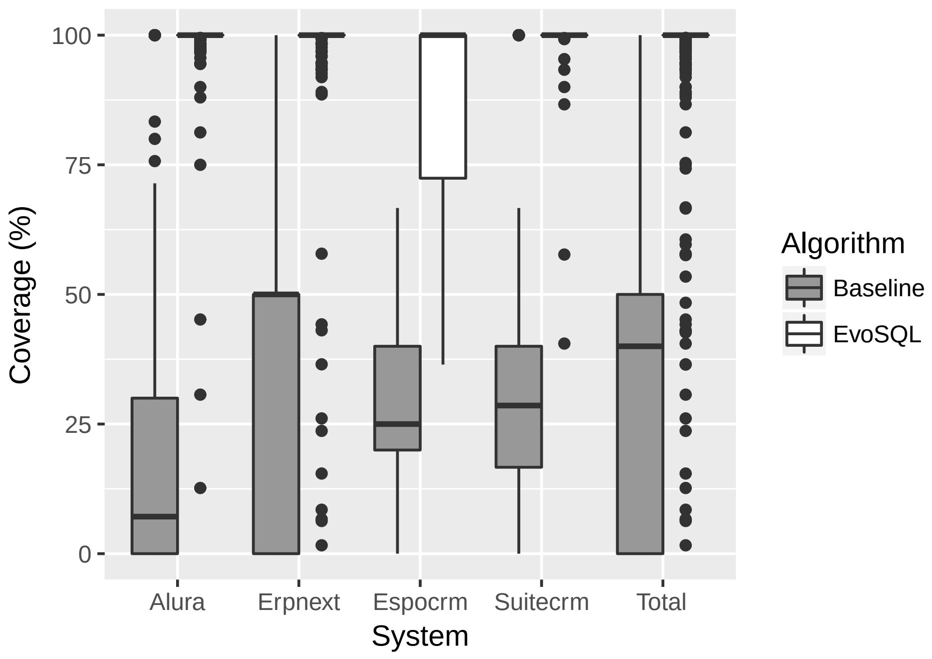

Figure 5.2: Boxplot of the average query coverage per system, baseline vs EvoSQL

Coverage Rules	1 - 2		3 - 4		5 - 6		7 - 8	
	B	E	B	E	B	E	B	E
Alura	2/12	14/0	11/40	51/0	0/33	33/0	0/38	37/1
EspoCRM	0/0	0/0	0/2	2/0	0/18	18/0	0/3	3/0
SuiteCRM	0/21	21/0	7/59	66/0	1/116	116/1	1/44	45/0
ERPNext	61/560	621/0	34/229	263/0	21/219	240/0	2/258	260/0
Coverage Rules	**9 - 10**		**11 - 15**		**16 - 20**		**21+**	
	B	E	B	E	B	E	B	E
Alura	0/24	23/1	0/46	35/11	0/25	19/6	0/18	8/10
EspoCRM	0/1	1/0	0/1	1/0	0/3	3/0	0/12	0/12
SuiteCRM	0/19	18/1	0/7	3/4	0/2	0/2	0/2	2/0
ERPNext	0/70	70/0	0/53	49/4	0/21	13/8	0/39	18/21

Table 5.4: Successful / failed queries, grouped by the amount of coverage rules, per algorithm and system. B: Baseline, E: EvoSQL

Coverage Rules	1 - 2		3 - 4		5 - 6		7 - 8	
	B	E	B	E	B	E	B	E
Alura	45.8%	-	25.8%	-	13.4%	-	17.0%	75.0%
EspoCRM	-	-	66.7%	-	30.0%	-	4.8%	-
SuiteCRM	50.0%	-	31.2%	-	35.1%	86.7%	6.3%	-
ERPNext	50.0%	-	62.6%	-	18.6%	-	2.0%	-

Coverage Rules	9 - 10		11 - 15		16 - 20		21+	
	B	E	B	E	B	E	B	E
Alura	17.2%	88%	8.7%	98.3%	4.2%	97.0%	7.9%	74.5%
EspoCRM	11.1%	-	18.2%	-	26.5%	-	24.9%	57.1%
SuiteCRM	9.8%	90.0%	6.6%	86.4%	8.9%	70.0%	4.5%	-
ERPNext	10.3%	-	5.2%	63.5%	6.8%	88.1%	3.8%	67.0%

Table 5.5: Average coverage of the failed queries, grouped by the amount of coverage rules, per algorithm and system. B: Baseline, E: EvoSQL

RQ$_1$: EvoSQL is more effective than the baseline. It is able to fully cover 2,053 out of 2,135 queries, while the baseline only covers 140 queries. When not fully covering a query, EvoSQL still achieves high coverage.

RQ$_2$: What are the causes when EvoSQL does not achieve 100% coverage?

Out of the 12,864 coverage rules in total, 932 were not covered by EvoSQL at least once. Throughout all coverage rules, the top 5 properties ranked by information gain are: Used Columns, Predicates, StringEQs, Inner Joins, and Tables. In Figure 5.3, we show the decision tree generated using these properties. If the leaf value is `FAILURE`, it classifies coverage rules with these properties likely to fail. If the leaf value is `SUCCESS` the coverage rules are likely to be covered successfully. The percentage in a leaf is the percentage of failing coverage rules that are classified by this leaf's properties. Using 10-fold cross-validation, the weighted average of the accuracy is 92.0%.

The Used Columns property does not appear as a decision variable. From the tree, we can conclude on common properties of failed coverage rules:

- **At least 6 predicates.** Only a small portion of the failing rules appears in the group with less than 6 predicates.

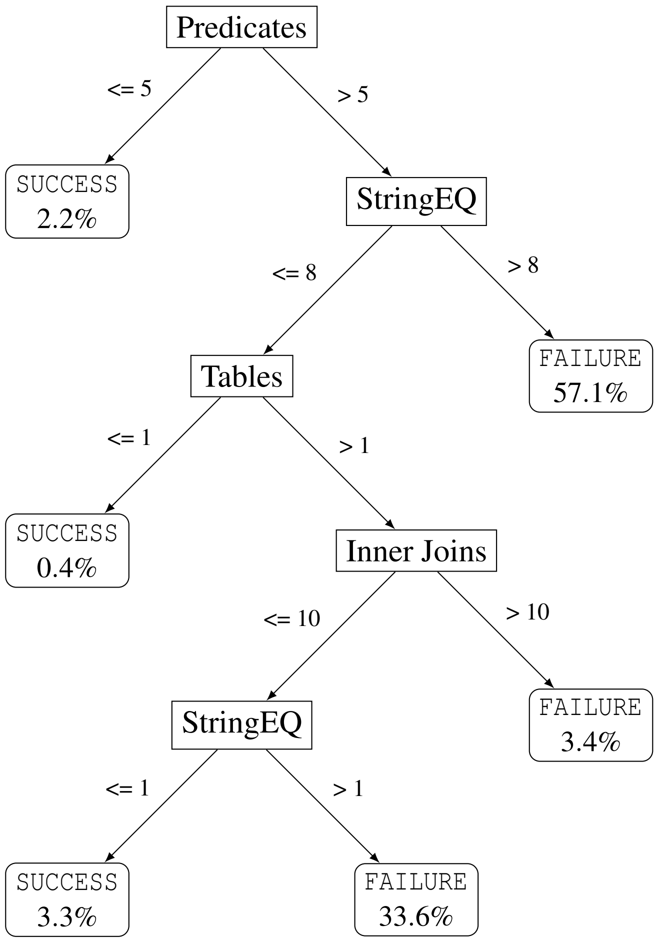

Figure 5.3: Decision tree generated by Weka. The percentage in the leafs indicates the fraction of failed coverage rules that this leaf classifies. The percentages sum up to 100%.

- **Many string equality predicates.** It appears in two decision rules, where the larger-than side classifies as failure for a large portion of the failing coverage rules.

- **Many inner joins.** While only appearing near the bottom of the tree, a portion of the failing coverage rules can be distinguished based on having many inner joins.

RQ$_2$: Coverage rules with few predicates are almost always successfully covered. The difficulties for EvoSQL lie in satisfying coverage rules with many string equality predicates, as well as inner joins.

Coverage Rules	1 - 2	3 - 4	5 - 6	7 - 8	9 - 10	11 - 15	16 - 20	21+
Mean	0.1	0.2	43.8	41.7	194.1	501.3	710.9	1,412.5
Standard deviation	0.1	0.3	273.6	253.5	537.9	732.7	756.1	612.4
0% (Minimum)	0.0	0.0	0.0	0.1	0.2	0.2	8.3	13.4
25%	0.1	0.1	0.1	0.6	1.1	5.1	31.3	964.3
50% (Median)	0.1	0.1	0.2	1.2	2.9	24.3	192.8	1,800.0
75%	0.1	0.2	0.4	1.2	10.8	954.2	1,604.5	1,800.0
100% (Maximum)	1.5	3.4	1,800.0	1,800.0	1,800.0	1,800.0	1,800.0	1,800.0

Table 5.6: EvoSQL execution time per query in seconds; mean, standard deviation, and 4 quantiles, grouped by the amount of coverage rules.

RQ$_3$: What is the performance of EvoSQL?

In Table 5.6 we show characteristics of the EvoSQL execution time in seconds per query, grouped by the amount of coverage rules of the query. In these results, the maximum execution time of a query is 30 minutes (1,800 seconds). Infeasible coverage rules are included in these results, to represent the time that users would have to wait, as they do not know beforehand whether a query will be infeasible.

For queries with 5 to 20 coverage rules, the median execution time is clearly lower than the mean. This is the result of failed queries in the set. These take the maximum execution time (i.e., 1,800 seconds), dragging up the mean. This is especially true for queries with up to 10 coverage rules, where the 75% is a lot lower than the 100%. As there are no failed queries in the queries with 1 to 4 coverage rules (see Table 5.4), the mean is closer to the median, and the maximum execution time is low.

For queries with at least 21 coverage rules, the mean is lower than the median. This is because the median of the execution time of these coverage rules is actually the maximum execution time for a query, as the majority of these queries do not achieve 100% coverage, and thus deplete the time budget.

RQ$_3$: EvoSQL has good performance for the majority of the queries, taking no longer than 11 seconds for 75% of the queries with up to 10 coverage rules. EvoSQL takes longer for queries with more coverage rules.

RQ$_4$: How do different time budgets impact the effectiveness of EvoSQL?

In Table 5.7 we show the coverage delta achieved by EvoSQL after running the 82 failed queries from the first evaluation with a larger time budget. The time budget is increased from half an hour to two hours. In general, the increased time budget increases coverage,

Coverage Rules	1 - 2	3 - 4	5 - 6	7 - 8	9 - 10	11 - 15	16 - 20	21+
Alura	-	-	-	±0.0%	+12.0%	+0.5%	+1.2%	+7.2%
EspoCRM	-	-	-	-	-	-	-	+22.1%
SuiteCRM	-	-	+5.0%	-	±0.0%	+11.0%	+22.1%	-
ERPNext	-	-	-	-	-	+12.2%	+9.0%	+25.7%

Table 5.7: Average coverage deltas between the failed queries of RQ_1 and the new evaluation, in which the time per query is increased to two hours.

especially for queries with more coverage rules. For the Alura queries it seems to have the least effect, except for those with 9 to 10 or at least 21 coverage rules.

EvoSQL achieved 100% coverage for 12 of the queries. There are 19 queries where no improvement is made (with a delta < 2%). The average coverage of these queries is 94.7%. Manual inspection of the coverage rules in these queries shows that for 14 out of the 19 queries, the cause is a combination of complex predicates that need multiple rows as output from the `FROM` clause. These coverage rules are hard to cover for EvoSQL, because the fitness function satisfies the `FROM` clause with a single row of output. We discuss these cases in Section 6.2. The remaining 5 queries had no distinguishing features, but seem to suffer from local optima.

In Figure 5.4, we show the average query coverage that EvoSQL would achieve for queries with at least 9 coverage rules, set out against the available time budget. In this figure, infeasible coverage rules are not excluded, as they also consume the time budget.

As for the queries with up to 8 coverage rules, that are not in the figure, all have between 98% and 100% average coverage, even on a time budget of one minute. For these queries, the time budget hardly impacts query coverage. The only group that benefits from a slightly larger time budget (1 $\rightarrow$ 3 minutes, 98.1% $\rightarrow$ 99.5%) is the queries with 7 to 8 coverage rules.

In Figure 5.4 we can clearly see that, for the queries with 11+ coverage rules, coverage does increase with a higher time budget. The coverage achieved does flatten out eventually. The only group of queries that has not shown to flatten out yet, is those with at least 21 coverage rules.

RQ_4: For failed queries with more than 10 coverage rules, the time budget has a substantial effect on query coverage. With a time budget of 2 hours, EvoSQL is able to increase coverage of these queries from 76.6% (in half an hour) to 90.2%.

5.5 Threats to Validity

While performing the study, we observed multiple threats to the validity of our study. In this section we present these threats, and what approach we took to mitigate them.

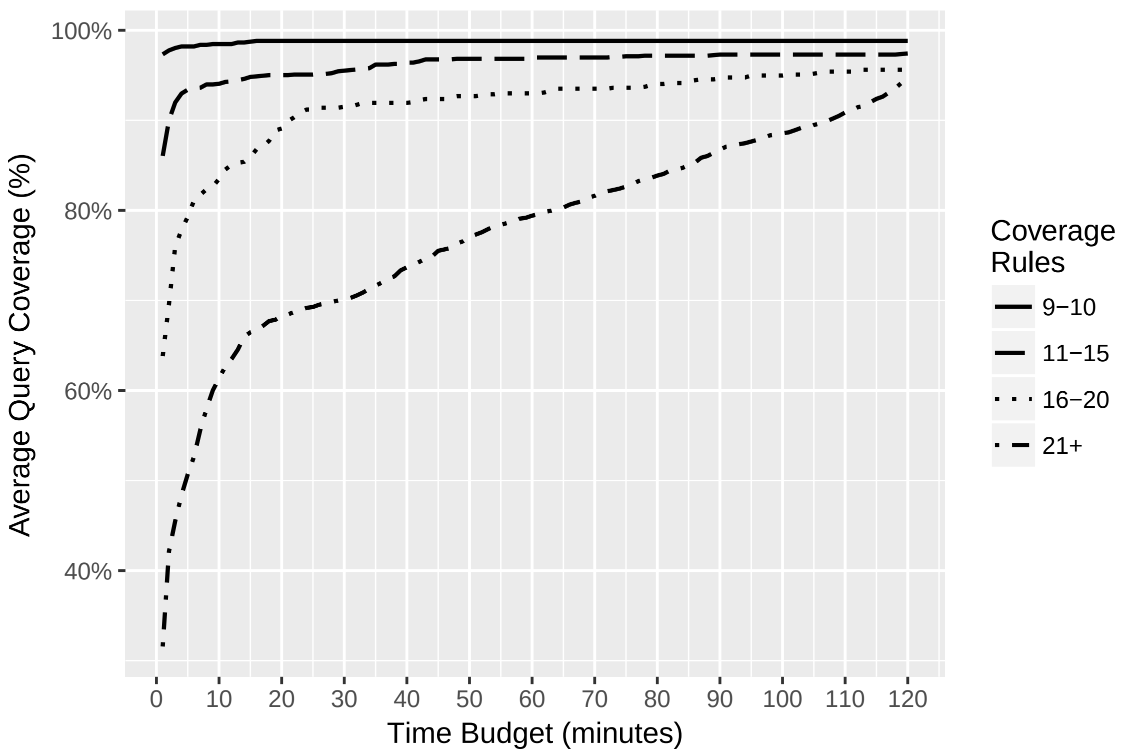

Figure 5.4: Average query coverage achieved by EvoSQL, dependent on time budget, for queries with at least 9 coverage rules.

External

With respect to external validity, the biggest threats are related to the representability of the study, and the correctness of the results. The evaluation queries were selected from four different applications to increase the diversity of the queries. Still, more research needs to be conducted to generalize our results to any SQL query possible.

The queries were collected through the applications' test suites, so that the queries are real queries that are used by the applications. This may mean that not all queries in the system are extracted, if their test suites do not execute all queries.

A limitation of our study is that not all queries from the original set of extracted queries were executable by SQLFpc or HSQLDB. Database specific syntax prevented this. This means that our evaluation set does not contain all of the business logic from the applications. Because the number of unusable queries was not very high, we argue that the evaluation set is still representative.

Both EvoSQL and the random baseline rely on the random function to generate data. A single evaluation may be 'lucky' and get higher results than the average evaluation. While there are already many queries that are evaluated, we also run the evaluation set 10 times.

The last external threat is that the coverage rules we use are generated by the SQLFpc web service. It could be possible that for some queries, the web service returns the wrong coverage rules, which implies that the data we generate does not cover the SQLFpc criterion.

Internal

With respect to internal validity, the first threat we observe is the infeasibility analysis performed on the results. This analysis on the 429 coverage rule queries is performed manually. Common SQLFpc constructions were identified to more easily decide whether a query is infeasible. To reduce the risk of classifying a feasible query as infeasible, two researchers analyzed the queries.

Another internal threat is the correctness of our implementation of EvoSQL and our instrumentation of HSQLDB. For EvoSQL, we maintain a test suite that preserves the correctness. For HSQLDB, we made sure not to harm any existing database functionalities by constantly executing its original test suite against our changes.

Finally, when we generate the decision tree for RQ_2, we start with many properties extracted from the coverage rules. A threat could arise when these properties can not successfully describe the class. Therefore, we included as many crucial parts of a SQL query we could think of, and then added other properties that we thought may be important, such as string equalities and `EXIST` expressions. Using all these properties, we selected the top 5 by applying an information gain evaluator. These 5 attributes have shown that they are good at classifying the coverage rules.

Chapter 6

Discussion

Throughout the study, we have encountered ideas and opportunities for future work. These derive from findings in the results, to choices that we made in the approach. In this section we discuss how we could do things differently, and what effects these different approaches may have. We also compare EvoSQL to other tools that share the goal of generating test data for SQL queries.

First, we evaluate the findings from the results. We discuss what makes string equality predicates hard, and how EvoSQL could improve on them in Section 6.1. We also discuss how we may have to generate multiple rows per clause in Section 6.2. In Section 6.3 we discuss for which coverage rules the baseline is good enough, and whether EvoSQL may benefit from incorporating the baseline for these coverage rules. Next, we compare EvoSQL to other tools with similar goals to ours in Section 6.4. We discuss how EvoSQL may be improved by tuning the GA in Section 6.5. Further, in Section 6.6 we argue that our approach is not limited to the SQLFpc coverage criterion, and what may change when using other criteria. We discuss how EvoSQL may be used in Section 6.7. Finally, the future work in Section 6.8 contains our ideas on improving and extending EvoSQL.

6.1 String Equality Predicates

In the results, we saw that string equality predicates are one of the most important factors of EvoSQL failing to cover a coverage rule in a decent time-frame. There are two forms of string equality predicates that we may see in queries:

1. `<column1> = 'value'`.
2. `<column1> = <column2>`, where both columns have a string type.

The first is more likely to appear in a `WHERE` predicate, while the latter is more likely to appear in a `JOIN` predicate. Examples of the effect of these predicates are the EspoCRM queries with more than 20 coverage rules (see Table 5.4). These queries each had about 7 inner joins with string equality predicates.

The reason that string equalities are so hard, is that the search space is enormous. Even though the fitness function gives fitter values to individuals that are closer to the solution, it is a long search and there are a lot of different mutations that can be applied to one string value. There is however one thing that both of the predicates above have in common, they are both added to the seeding pool. A possible solution could be to tune the probability of seeding values during mutation. This may however have an adverse effect on queries that do not need a higher seeding probability. Another solution could be to have a higher probability of generating small strings. String equality predicates are easier to solve if both are short strings, as less mutations are possible. This could help when solving joins, but may have an adverse effect if there are predicates requiring long string values.

6.2 Clause Multi-Satisfaction

The fitness function we present focuses on each clause having some output. Once there is at least one row of output, it is satisfied. However, as we have seen in the results of RQ_4, some coverage rules require a clause with difficult constraints to return multiple rows, as a later clause may contain predicates that need this. An example of such a query is as follows. To successfully cover such a coverage rule, there must be multiple `t1.id` values that match a `t2.id` value.

```
SELECT *
FROM t1
INNER JOIN t2 ON t1.id = t2.id
HAVING COUNT(DISTINCT t1.id) > 1
```

In our current implementation a single match will be made, at which point the `FROM` clause is satisfied. Then, when the fitness of `HAVING` clause is calculated it compares the number of distinct `t1.id` values (i.e., 1) to be greater than 1.

This results in the GA getting stuck in a local optimum where it believes the `HAVING` clause is close to being satisfied, however the `FROM` clause should be targeted to return multiple rows.

Although the amount of queries this affected is small (0.7% of our evaluation set) and the query coverage achieved on these queries is still 95% on average, this issue should be investigated in future work.

6.3 Performance of the Baseline

Although we concluded that EvoSQL is superior to the baseline, the baseline still achieves decent average coverage for some queries. We will now analyze these queries to see why some of their coverage rules are easily covered, and whether the baseline could outperform EvoSQL on them.

Coverage rules						
1	2	3	4	5	6	7
98.2%	50.5%	68.0%	33.3%	22.0%	38.6%	5.7%

Table 6.1: Average coverage for the baseline

Queries with 1 to 7 coverage rules

Table 6.1 contains the average coverage for the baseline for queries with 1 to 7 rules. An interesting pattern can be seen here, namely that for queries with 2 rules, 1 is covered on average. For queries with 3 rules, 2 are covered on average. With more rules, the coverage gradually goes down, only peaking up once at 6 rules. From 7 rules onwards, the average coverage remains low. The reason behind this pattern is the way SQLFpc generates coverage rules.

All queries with 1 coverage rule have syntax like: `SELECT MAX(<column>) FROM <table>`. The rule generated for such a query tests the column aggregation, and requires some data such that there are at least two equal values and at least one other value in the used column. This is not hard to achieve randomly.

The queries with 2 or 3 rules are almost all queries with a single predicate (874/881). If the predicate *pred* uses a comparison operator and both values (*lv, rv*) are numeric, SQLFpc generates three rules with boundary values.

1. $lv = rv + 1$
2. $lv = rv$
3. $lv = rv - 1$

Each of these rules is easy to cover for the baseline, as long as this is the only predicate to satisfy. If they are not numeric or using another operator, SQLFpc generates two rules.

1. *pred* is `TRUE`
2. *pred* is `FALSE`

These two rules together cover the entire search space except for the smaller case resulting in `UNKNOWN`. If one of them is difficult to cover, the other is easy to cover. This is because almost any data that does not satisfy one rule, will satisfy the other rule. This is where the 50% comes from for queries with two rules. If the predicate uses a column *c* that is nullable, SQLFpc adds one rule.

1. *c* is `NULL`.

This rule is easy to cover, as the `NULL` value will likely appear in one of the random individuals. With this rule added, we can see that the 68% for queries with 3 coverage rules is due to a large amount of them having this rule in combination with the prior two.

For the queries with 4 to 6 coverage rules, the rules are a combination of the rules above, but often with multiple predicates. With more predicates, there are more rules in which a base predicate (as it was in the original query) must be `TRUE` and fewer in which it must be `FALSE` or `UNKNOWN`. Therefore, the average coverage goes down. The upwards spike for 6 rules is caused by 20 out of the 142 queries having two base predicates that are easy to solve by the baseline (e.g., $c_1 <= 1$ `AND` $c_2 >= 6$). These all achieve 100% coverage, lifting the average coverage.

EspoCRM queries with $10+$ coverage rules

In Table 5.5 the baseline achieves decent coverage on the EspoCRM queries with more than 10 coverage rules, especially when compared to the other systems. Specifically, it gets up to around 25% average coverage on queries with at least 16 coverage rules. The reason for this is similar to the cases with few coverage rules. Namely, some of the generated coverage rules are easy to cover. Each of these EspoCRM queries have many inner joins in the original query. A portion of the coverage rules generated by SQLFpc transform these joins into combinations of left and right joins. This could for instance be a right join followed by many left joins. Because by definition of SQLFpc none of these joins have to be satisfied, it is very easy for the baseline to cover, as almost any data can.

To conclude, the random baseline is only able to solve trivial coverage rules, and predicates such as $c_1 =$ `'value'` can hardly ever be satisfied. For the trivial coverage rules, we could choose to apply the random search rather than EvoSQL, to reduce complexity and execution time. The advantage here would be that there is no overhead of applying the fitness function on the initial population of the GA, as the GA will also very likely find the solution in the first population. The downside is that each coverage rule would have to be analyzed in order to detect whether random search can be applied, or alternatively start each coverage rule in queries with few rules with a short execution of random search. We argue that the advantage does not outweigh the downsides, so there is no need to ever apply random search instead of EvoSQL. Still, for a conclusive answer, future research needs to be conducted.

6.4 Comparison to Tools

In the research field, multiple tools have been introduced with similar goals to EvoSQL, generating test data or test databases based on one or more SQL queries. Each of these tools use constraint solvers to generate data. Because they use solvers, they have to describe the SQL query so that the solver can generate data for it. This introduces limitations in their solutions, namely that they have to be able to describe the entire SQL syntax (which they have not done so far), and the solver itself may not be able to satisfy the necessary constraints. Common limitations include subqueries and string predicates, which are common constructions in SQL queries (in our evaluation set, 84.1% of queries contained such constructions). EvoSQL benefits from using an existing, fully functioning SQL database. This way, 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. This prevents us from doing an empirical comparison between us and them. In the following subsections, we discuss existing approaches and highlight the differences with our approach.

QAGrow

The QAGrow tool presented by Suárez-Cabal et al. [35] generates test databases for a set of queries, using the SQLFpc coverage criterion. QAGrow generates test databases rather than test data, making it a more complete solution for testing purposes.

- **Approach:** 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*, to generate the test data. They evaluate their tool on 215 queries taken from a closed-source system, having $1,339$ coverage rules. From these coverage rules, they state they have removed the infeasible ones before execution. On these queries, they achieved 99.0% SQLFpc coverage, in about 2 minutes time.

- **Limitations:** QAGrow is not able to solve string constraints, although they state they generate integers when the column type is string. They also do not solve queries with subqueries. The queries that they evaluate are not available.

- **Comparison to EvoSQL:** EvoSQL is able to cover more types of SQL queries, most importantly string constraints. Unfortunately, besides the known restrictions, the QAGrow evaluation set is not defined in fine detail, nor is it publicly available. This prevents us from doing a real comparison.

ADUSA

The ADUSA tool presented by Khalek et al. [21] also generates test databases. The goal of their tests is, however, to test database management systems (DBMS). The generated test databases contain data for a single test query.

- **Approach:** The approach generates tests by rewriting SQL queries (the schema queries and the test query) into *Alloy* specifications. Alloy is a language for relational models, and comes with a tool, the Alloy Analyzer, which is a SAT solver that generates data satisfying the specified relational model. For one query, it can generate many test cases. For each test generated by the Alloy Analyzer, the authors also generate a test oracle, i.e., the correct result of the test. They do this by executing the test on a trusted DBMS, that supports the features under test. They evaluate their approach by inserting bugs in existing databases, and running the generated tests on the buggy databases. In these case studies, their tests are able to find the bugs.

- **Limitations:** They support `FROM`, `WHERE`, `GROUP BY` and `HAVING` clauses. As for joins, they only support natural joins (an implicit join between two tables, using common column names) and cross joins (which have no predicates). The Alloy solver is unable to solve string constraints.
- **Comparison to EvoSQL:** As the goal of their tool is to test databases, rather than a query, a comparison to EvoSQL is not easily made. One difference is the SQL syntax that is not supported as described in the limitations. EvoSQL, on the other hand, supports all joins, as well as subqueries and string predicates.

QAGen

The QAGen tool presented by Binnig et al. [4] also generates test databases with the goal of testing a DBMS, based on a single query.

- **Approach:** As in the previous tools, the approach uses a constraint solver to generate test data. The model is built using symbolic query processing, their extension of symbolic execution. They also allow the user to set so-called *knobs*, for instance output size. In their evaluation, they generate databases of different sizes, 10MB, 100MB and 1GB, and analyze the efficiency of their approach.
- **Limitations:** They do not support subqueries, and solely support joins that just use foreign key constraints in the join predicate.
- **Comparison to EvoSQL:** The tool presented by the authors aims to test databases rather than queries. Therefore, a comparison to EvoSQL is not easily made. Their study also does not evaluate the effectiveness of their tests. As for limitations, EvoSQL does support subqueries and other joins, which QAGen does not.

Dynamic Test Input Generation for Database Applications

In this paper by Emmi et al. [10], they describe an algorithm to automatically generate test input for database applications, with the goal of maximizing branch coverage. The test input consists of input data for the program as well as a filled database to cover all paths of the program.

- **Approach:** The approach also uses a constraint solver, where the constraints are a combination of the path constraints in the program and the database constraints. They evaluate their approach with a case study of one method in which they argue they are successful.
- **Limitations:** Although they do handle string constraints (equality, inequality and LIKE), they support only `FROM` and `WHERE` clauses, excluding joins.
- **Comparison to EvoSQL:** This research has more similar goals to EvoSQL as the previous works, as they generate data with the aim of covering certain rules, albeit in this case the path constraints in a program. However, they do not evaluate their

effectiveness. In terms of limitations, they are very limited in the SQL syntax they support, contrary to EvoSQL.

6.5 Tuning the GA

Throughout Chapter 3 we presented the specifics of our GA. Many of the operators and probabilities we chose for our study were chosen through practice and by selecting default values for GAs. While they could be improved, by experimenting and tuning the parameters, it has been shown that default values will always give reasonable results [2]. Also, tuning could only improve the effectiveness of EvoSQL, even though it is already very high. In this section, we present our views on some of the operators we chose, and other options that are available. We also discuss how we picked some of the less trivial probabilities.

Crossover operator

In our approach, the crossover operator is straightforward: swapping a random table between two individuals. Although this has worked in our implementation, it may be interesting to investigate different, more fine-grained operators. Examples are: swapping one or more rows between two individuals, or going as deep as swapping column values between two rows from the individuals. We chose not to experiment with different crossover operators as the one we chose is also the simplest one to implement, and certain to affect the fitness value of the individuals. We also deemed this operator to be logically sound. When joining two tables, it seems more valuable to swap entire tables than just a few rows, as the new individual will have more rows that possibly are a better match for the join.

Mutation probabilities

In Section 3.4.2 we introduced the four different actions that may be applied onto a table by the mutation operator: deleting, duplicating, adding or mutating a row. Because in practice we found that deleting, duplicating or adding a row is not often required, we set their probabilities relatively low with 1/6 for each of them. At the same time, we chose to always apply row mutation to each row. We made these choices because we saw that the GA spent most of its time needing column value mutations, creeping towards the correct value.

Let's say for instance that there is some join between two string columns $c_1 = c_2$ that needs to be satisfied. At some point, the GA will generate two values that bring this join close to being satisfied, $c_1 =$ '*abcde*' and $c_2 =$ '*abcdz*'. Without taking seeding into account, there is no point for the GA to add or delete any rows now. Duplicating a row may make sense so that there are multiple rows that are close to satisfying the join. However, the most important mutation to be done here is changing the column values.

Clearly, the other mutation operators are still necessary. It may be interesting to see if this thought process can be executed by the GA. It could use data from the query execution to decide whether row mutation is needed, or whether other actions should take priority.

Previous population cloning

During initialization, we re-use some individuals from the previous coverage rule's population, if any. Deciding on the probability of this action is not straightforward, as there is no known default value for it. A different seeding technique that has been researched is the cloning of previous solutions [31]. This is slightly different as in our implementation we may clone any individual from the population, not just solutions. We do this because these individuals are all closer to the previous solution than the initial random set of individuals, by definition of elitism. In the research done on cloning previous solutions, a probability of 0.9 was found to be the best. We do not want to hurt the global search aspect of our genetic algorithm by seeding any individual with such a high probability, which is why we chose 0.6 instead (also one of the better options in the aforementioned study [31])

6.6 Coverage Criteria

In our implementation, we used the SQLFpc coverage criterion to generate coverage rules. However, the approach we present is not limited to the SQLFpc criterion and should work on any criterion so long as the coverage rules can be represented as SQL queries. One of the benefits of using SQLFpc is that solving left and right joins is not ambiguous, the algorithm can aim to never satisfy these joins. If another criterion is used, our implementation of the fitness function may have to be adapted. If the left and right joins in this criterion are ambiguous, the implementation would have to adapt accordingly. A solution could be to use static query analysis to determine whether a joined table's data requires some matching data, or must be `NULL`.

6.7 Applicability of EvoSQL

We have successfully created a tool that generates test data for SQL queries, named EvoSQL. We argue this tool could be integrated into the testing pipeline seamlessly. Before this becomes reality, more work still has to be done. We see two directions to be worked on to make this happen.

1. Using the test data, test cases can be written in which methods are called that execute SQL queries. By using the test data from EvoSQL in these test cases, these test cases will check whether the query behavior is still correct. These test cases could be automatically generated.

2. Another use for the test data is for developers to help them understand their queries. As queries get more complex, mistakes are more easily made, as the amount of dimensions in the data grows with each join. By inspecting the generated data, as well as which rows of data appear in the output, the developer may spot mistakes and choose to modify the query. These mistakes could either be in the form of rows that appear in the output but shouldn't, or rows that should appear in the output but don't.

6.8 Future Work

We split the future work into two sections, improving EvoSQL, in which we discuss how we could improve the effectiveness, and extending EvoSQL, where we discuss how the next steps for EvoSQL could be implemented.

6.8.1 Improving EvoSQL

There are several ideas for improving EvoSQL, which we discuss below.

Using schema constraints Currently, the test data we generate does not adhere to all schema constraints of the given database. The unique constraint, specifying there can be only distinct values in a column, as well as primary and foreign keys are not used. While the unique constraint may be counterproductive during search, it should still be satisfied by the final individual. As for key columns, this can definitely be used by the GA, as it will improve the speed of solving joins. In implementation, a mutation on a column that is a primary key could automatically also mutate each linked foreign key, and vice versa.

String equality predicates The biggest problem that we see EvoSQL face is the string equality predicates, as we discussed in Section 6.1. While string mutation itself is hard, we believe that improving and tuning the seeding strategies may result in these predicates being solved more easily. A possible solution could be to tune seeding probabilities based on the type of a column, so that string types are more likely to use seeded values. If the supplied schema is well defined and joins are done using foreign key columns, using the schema constraints as suggested in the previous paragraph will also help solve these.

Clause multi-satisfaction As we discussed in Section 6.2, clauses may sometimes have to return multiple rows. To implement this, the fitness function would need to be adapted so that it can 1) detect when multiple rows are needed and 2) target a clause to make it return more rows.

This is difficult to pull off, because it is difficult to analyze the SQL and detect how many rows should be returned, and how these rows should be different. However, as we have seen how the coverage rules with these constraints are constructed, a first step may be to generate 2 rows of output with different values in the columns in the join predicates.

Guided mutation We conjecture adding more guidance to the GA will improve the performance of EvoSQL. One possibility could be extending the database instrumentation to also tell the GA which columns are currently problematic, so that the mutation can be directed to these columns. Doing this does add risks; if the instrumentation wrongly informs the GA, it will focus mutation on the wrong columns, causing the GA to take longer than it otherwise would. Also, extracting this data will slow down the fitness function.

Another possibility is to retrieve more information through static analysis. Because the SQL grammar is well-defined, more information could be extracted through static analysis than is currently done. Symbolic execution techniques as presented in other work could be applied to limit the search space of certain column values. If a constraint is found specifying what value a certain column should be (e.g., `<column> = 'TU Delft'`), and this constraint has to evaluate to `TRUE`, then during mutation or even initialization, this value can be set. This could reduce the total search time while only adding a constant amount of time of static query analysis.

We argue that for both of these ideas the downsides do not outweigh the improvement, however future research needs to be conducted to assess this.

Multi-objective algorithm EvoSQL uses a single-objective genetic algorithm to generate test data. This means that one fitness function is used at a time, namely one for a single coverage rule. An interesting alternative is implementing a multi-objective approach (see Section 2.1) by using fitness functions for multiple coverage rules simultaneously. This approach may benefit from the idea that many solutions lie close to each other, and will detect when a solution is found for one of the coverage rules, while being able to continue on afterwards.

Improving data interpretability The data generated by EvoSQL is derived from random values. Therefore, many column values in the output test data are obscure, and not easy to interpret by a human. To improve on this, we could use dictionaries that are derived from either the table and column names, or user input. Doing this during test data generation may harm the speed and effectiveness of the GA, as less of the search space will be explored. A better alternative is to apply post-processing on test data, only altering column values if the coverage rule remains covered.

6.8.2 Extending EvoSQL

As we discussed earlier, there are multiple ways the data generated by EvoSQL can be used.

Generating test cases Using the test data, software tests can be written. The input to EvoSQL could be a method that executes a query, as well as parameters for this method. This input could be provided by a developer, or a test case generator, such as EvoSuite. Given the input, EvoSQL could extract the SQL query, generate test data, and generate a test case. This test case would then load up the test data, execute the method with the given parameter and test if the query output is what is expected. In this case, the expected output is the output that the original query would have. Issues that could arise here is that not all test data may fit in the same database, due to schema constraints. Even if it does fit in the same database, putting the data together may cause coverage rules with aggregations to no longer be covered. A possible solution to this problem is to execute the query on each coverage rule's data one by one. While this is a safe and easy solution, it does impact the runtime of the test.

Generating query data visualizations The other use for the test data is granting developers insights into their queries. The simplest approach to visualizing this is by presenting the test data in a database to look at, and the output of the developers' query. This is a hard task when many tables are involved, especially when all data is randomly generated. Therefore, the data interpretability should be improved first, as suggested in the previous section.

SQL Exceptions In Section 4.3 we discussed SQL exceptions and how the fitness function deals with them. Even if the individual is the cause of the exception, the query syntax is unsafe if it allows for these exceptions to be thrown. This information is also interesting for developers, and EvoSQL could also generate test data that throw exceptions. Ideally, the coverage criterion would generate coverage rules for which the data throws exceptions on the original query.

Chapter 7

Conclusion

With database-centric applications being central to modern-day life, and important business rules being applied through SQL queries, testing SQL queries is just as important as testing any other software code. As writing these tests manually is a difficult task, automated test data generation is necessary for developers to be able to test their queries. There has been other research aiming to do this, however they share common limitations like being unable to generate data for SQL queries with subqueries, or string constraints.

To this end, we have presented our novel search-based approach to generating test data for SQL queries, that does not suffer these limitations. Our approach collects data from all constraints of a SQL query, and we instrumented a real database to realize this. In our study, we evaluated the effectiveness of our approach on $2,135$ queries from 4 real-world systems, and were able to fully cover $2,053$ of them.

We presented the following research questions, and answered them accordingly:

- **RQ$_1$:** *What is the difference in the query coverage achieved by EvoSQL and the baseline?* EvoSQL is more effective than the baseline. It is able to fully cover $2,053$ (i.e., 96.2%) out of $2,135$ queries, while the baseline only covers 140 queries. When not fully covering a query, EvoSQL still achieves high coverage.

- **RQ$_2$:** *What are the causes when EvoSQL does not achieve 100% coverage?* Coverage rules with few predicates are almost always successfully covered. The difficulties for EvoSQL lie in satisfying coverage rules with many string equality predicates, as well as inner joins.

- **RQ$_3$:** *What is the performance of EvoSQL?* EvoSQL has good performance for the majority of the queries, taking no longer than 11 seconds for 75% of the queries with up to 10 coverage rules. EvoSQL takes longer for queries with more coverage rules.

- **RQ$_4$:** *How do different time budgets impact the effectiveness of EvoSQL?* For failed queries with more than 10 coverage rules, the time budget has a substantial effect on query coverage. With a time budget of 2 hours, EvoSQL is able to increase coverage of these queries from 76.6% (in half an hour) to 90.2%.

We conclude that our approach is highly effective, yet needs time to achieve full coverage, especially for queries with many coverage rules. Using the results of our study, we discuss possibilities for improving EvoSQL in terms of effectiveness, performance and usability. We conjecture that, with these improvements, EvoSQL can be integrated into the testing pipeline seamlessly.

Bibliography

[1] ISO/IEC 9075-2:1999. *Information technology - Database Language - SQL - Part 2: Foundation (SQL/Foundation)*, 1999.

[2] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[3] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 506–515. IEEE, 2007.

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

[5] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases*, pages 1097–1107. VLDB Endowment, 2005.

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

[7] David Chays, Yuetang Deng, Phyllis G Frankl, Saikat Dan, Filippos I Vokolos, and Elaine J Weyuker. An agenda for testing relational database applications. *Software Testing, verification and reliability*, 14(1):17–44, 2004.

[8] John J. Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical report, Federal Aviation Administration, US Department of Transportation, Washington, D.C., 2001.

[9] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

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

[11] Gordon Fraser and Andrea Arcuri. 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*, pages 416–419. ACM, 2011.

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

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

[14] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. Quickly generating billion-record synthetic databases. In *ACM Sigmod Record*, volume 23, pages 243–252. ACM, 1994.

[15] Neelam Gupta, Aditya P Mathur, and Mary Lou Soffa. Generating test data for branch coverage. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 219–227. IEEE, 2000.

[16] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[17] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

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

[19] Thomas Jansen. *Analyzing evolutionary algorithms: The computer science perspective*. Springer Science & Business Media, 2013.

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

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

[22] Shadi Abdul Khalek and Sarfraz Khurshid. Systematic testing of database engines using a relational constraint solver. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 50–59. IEEE, 2011.

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

[24] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1098–1105. ACM, 2007.

[25] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2016.

[26] Phil McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

[27] Phil McMinn, Chris J Wright, Cody Kinneer, Colton J McCurdy, Michael Camara, and Gregory M Kapfhammer. Schemaanalyst: Search-based test data generation for relational database schemas. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 586–590. IEEE, 2016.

[28] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software testing, verification and reliability*, 13(1):25–53, 2003.

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

[30] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):12, 2014.

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

[32] Shetal Shah, S Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. Generating test data for killing sql mutants: A constraint-based approach. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1175–1186. IEEE, 2011.

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

[34] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*, pages 209–220. IEEE Press, 2017.

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

[36] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. *Tests and Proofs*, pages 134–153, 2008.

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

[38] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.