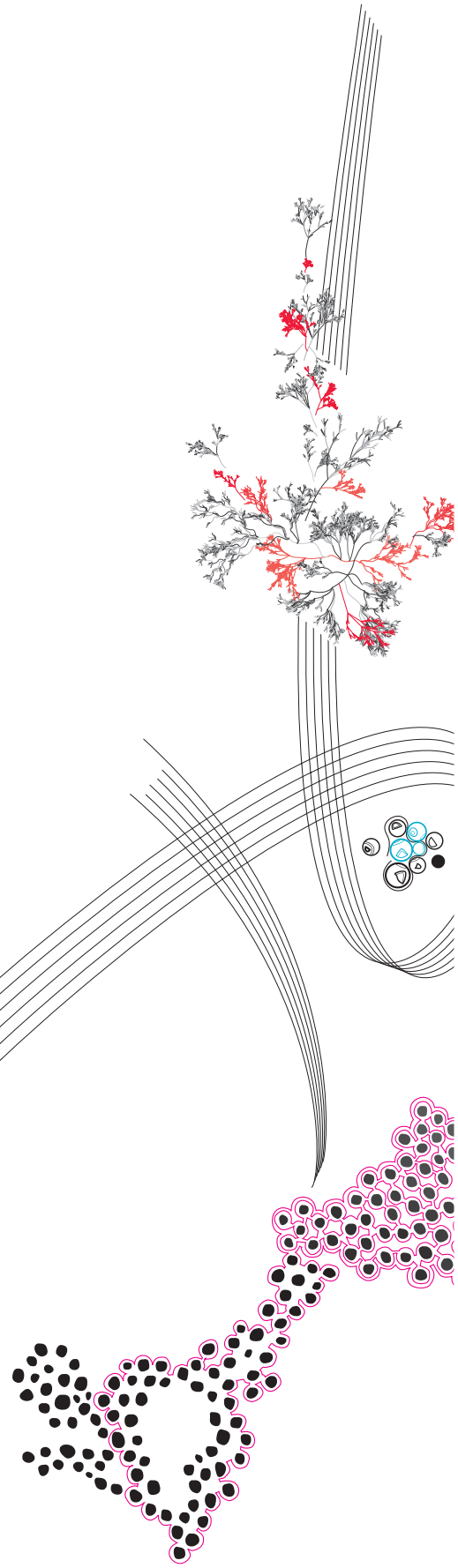

MSc Cybersecurity
Master Thesis

Effective Fuzzing with Constraint-Aware Oracle to Detect Logical Bugs in Database Management Systems

Niccolo Parlanti

Supervisor: Andrea Continella
Supervisor: Luca Mariot
Committee Member: Petra van den Bos

September, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Effective Fuzzing with Constraint-Aware Oracle to Detect Logical Bugs in Database Management Systems

Niccolò Parlanti
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
n.parlanti@student.utwente.nl

September 2024

Abstract

Database Management Systems (DBMSs) are fundamental to data storage, retrieval, and management across various applications. Ensuring their security and integrity is crucial, as vulnerabilities can lead to system crashes or exploitation through logical bugs, errors that cause the DBMS to return incorrect results without crashing. Prior research in DBMS testing has predominantly focused on identifying crash-related bugs, often overlooking logical bugs, like those related to the enforcement of constraints within the system. Constraints are essential rules that ensure data integrity and consistency, but their complexity makes them challenging to test with existing tools. This paper presents a novel approach that combines evolutionary algorithms and an oracle specialized in the detection of logical bugs related to DBMS constraints. We employ an evolutionary algorithm to generate new test inputs, which are then evaluated by our oracle to identify any violations of the constraints. This method aims to uncover logical bugs that traditional testing methodologies might miss. Our prototype has been evaluated on MySQL, and the results provide valuable insights into the effectiveness of this approach and its potential applications in improving DBMS reliability and security.

1 Introduction

Database Management Systems (DBMSs) play a crucial role in data-intensive applications, supporting billions of devices and managing trillions of databases. Given their extensive use, any bug in a DBMS does have significant impacts on a large number of users [17, 20, 26]. The correctness and reliability of DBMSs are paramount, especially for critical enterprise applications such as online banking, e-commerce, and electronic payment systems. Ensuring these systems function correctly is essential to maintaining trust and operational efficiency in these critical areas.

Both industry and academia have devoted considerable attention to DBMS testing, recognizing the importance of preventing failures. Techniques such as fuzzing have been widely employed, demonstrating their effectiveness in uncovering crash bugs and assertion failures that cause the system to terminate unexpectedly [27, 29, 6, 8]. However, while these methods are useful, they fall short when it comes to detecting logical bugs, errors that can lead to incorrect query results without causing the system to crash. Logical bugs present a more insidious challenge, as they do not manifest as obvious failures like system crashes. For instance, a logical bug might result in a DBMS leaking extra rows or returning incorrect data, which can go unnoticed without proper validation [11]. Detecting these logical bugs requires a more sophisticated approach, often involving the use of oracles to verify the correctness of each execution result.

Recent advancements in DBMS testing have significantly enhanced the detection of logical bugs, particularly through the use of oracles. In the context of software testing, an oracle is a mechanism that determines whether the output of a program under test is correct. Specifically, in DBMS testing, oracles are used to verify the correctness of query results by comparing them to expected outcomes. They work by predicting what the correct result of a query should be based on certain rules or transformations. For instance, Rigger et al. developed innovative oracles by transforming SQL queries into semantically equivalent forms [22, 24, 25]. This method leverages the fact that different DBMS code paths should produce identical results for these transformed queries. Discrepancies in the results highlight potential logical bugs. SQLancer [21], the tool that

embodies these oracles, has proven highly effective, uncovering numerous logical bugs across various DBMS.

However, SQLancer's approach is not well-suited for identifying logical bugs related to table constraints. SQLancer relies on generating and transforming SQL queries to exploit logical properties and detect inconsistencies in DBMS operations. This technique is effective for general query processing but does not specifically target the enforcement of table constraints such as `unique` constraints and `check` constraints. These constraints involve specific validation rules that must be upheld by the DBMS. Bugs in these areas often do not manifest through query transformations because they pertain to the integrity rules applied during data manipulation operations, rather than the logical consistency of query results.

In this paper, we present a novel fuzzing approach specifically designed to detect logical bugs in Database Management Systems, focusing on components such as `unique` constraints and `check` constraints. We designed this approach to utilize a mutation-based evolutionary algorithm [2], which generates various test inputs, including edge cases and invalid data entries. These inputs are then used to query the target database. Our custom-built oracle processes each query, understanding the specific constraints applied to the database schema and determining whether each input should pass or fail based on these constraints.

The oracle simulates the correct behavior of the DBMS by evaluating if the inputs adhere to the defined rules, such as uniqueness or specific conditions. Finally, we conduct a thorough evaluation by comparing the actual responses of the DBMS with the expected results generated by the oracle. Any inconsistencies between the DBMS and the oracle indicate the presence of a logical bug. We evaluate the framework on real-world MySQL systems, demonstrating its effectiveness in detecting constraint-related bugs and validating the robustness of the DBMS.

In summary in this paper we make the following contributions:

- We design an oracle capable of understanding and validating inputs against DBMS constraints, ensuring that the generated test cases accurately reflect the expected behavior and detect deviations.
- We introduce a novel fuzzing framework specifically designed to detect logical bugs related to DBMS constraints, such as `unique` constraints and `check` constraints, which are overlooked by traditional testing methodologies.
- We evaluate our program on real-world DBMS systems, specifically MySQL, identifying 4 potential bugs related to constraint handling. These bugs will be submitted to the MySQL team for further evaluation and verification.

2 Background and Motivation

To understand the research presented in this paper, it is essential to first introduce and explain the core principles of fuzzing methodologies [10]. This chapter offers a general overview of fuzzing, detailing its mechanisms and applications. The discussion will then shift to the specific challenges of applying fuzzing to Database Management Systems (DBMSs), particularly in the detection of logical bugs, accompanied by relevant examples. Furthermore, the critical role of oracles in identifying these bugs will be examined, with a focus on comparing traditional oracles from the custom-designed oracle developed in this study.

2.1 Fuzzing: An Overview

Fuzzing is a software testing technique that automatically generates and injects data into a program to uncover bugs [19]. Originally coined by Miller et al. in 1990 [15], the concept has evolved significantly with the years. Modern fuzzing tools are sophisticated, generating semi-random inputs that follow complex criteria, such as code coverage, to explore different execution paths within the software. This method allows fuzzing to effectively discover a wide range of vulnerabilities, from simple crashes to complex security weaknesses in a diverse set of applications.

The general workflow of fuzzing involves several key steps (Figure 1):

- System Preparation: Setting up the initial conditions, such as seed inputs and configurations, to ensure the system is ready for testing.
- Test Case Execution: Generating and running the test cases to observe how the software handles unexpected or malformed inputs, which is essential for detecting hidden bugs.
- Exploring Program States: Continuously navigating through different program states to uncover vulnerabilities that may not be evident during standard execution.
- Progress Reporting: Documenting the results, including any detected crashes or anomalies, to refine the fuzzing strategy and focus on areas of interest.

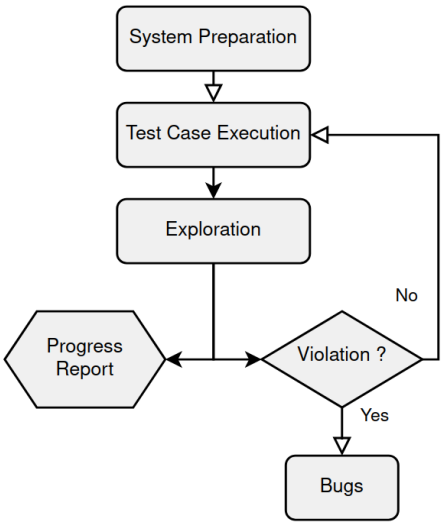

FIGURE 1: Basic working process of a fuzzing test.

Fuzzing can be classified into three main categories:

- Black-Box Fuzzing: Operates without knowledge of the program's internal structure, focusing purely on input-output behavior.
- White-Box Fuzzing: Involves deep analysis of the program's source code, using techniques like symbolic execution to generate and test inputs systematically.
- Grey-Box Fuzzing: Combines aspects of both black-box and white-box fuzzing, leveraging partial knowledge of the program to guide input generation and enhance test coverage.

Fuzzers can also be categorized based on the methods they use to generate inputs:

- Generation-Based Fuzzing: Involves crafting inputs that adhere to the application's expected formats, leveraging detailed knowledge of input specifications.
- Mutation-Based Fuzzing: Modifies existing valid inputs (known as seeds) to create new test cases by making small alterations.
- Evolutionary Fuzzing: Incorporates evolutionary algorithms to iteratively refine test cases based on feedback from previous tests, optimizing the fuzzing process for greater efficiency and effectiveness.

2.2 Fuzzing for DBMS

Fuzzing DBMSs is a specialized approach to discovering vulnerabilities that extends beyond conventional software fuzzing techniques. It involves systematically testing DBMSs by injecting malformed or unexpected inputs, monitoring for exceptions, crashes, or data integrity violations. This process requires specific strategies tailored to the unique characteristics of DBMSs. As with generic fuzzing, there are several strategies (Figure 2):

- Black-Box Fuzzing: Operates without any internal knowledge of the DBMS, focusing purely on external interfaces and behaviors, making it broadly applicable but less efficient at discovering deeper vulnerabilities.
- White-Box Fuzzing: Utilizes detailed internal knowledge of the DBMS structure and logic, allowing for targeted input generation and deep coverage of internal paths.
- Grey-Box Fuzzing: Balances internal knowledge with limited access, often employing instrumentation to guide test case generation.

While the general methodologies and types of bugs targeted, such as crashes, logical bugs, and performance bugs, are common to both DBMS fuzzing and generic software fuzzing, the distinct challenges of DBMS fuzzing arise from the complex interaction with SQL and the intricate nature of database systems. Unlike standard software fuzzing, where bugs might be identified through simple output comparisons or crash detection, DBMS fuzzing must account for the behavior of SQL queries, including logical bugs that might result in minor inaccuracies in data retrieval or manipulation. For instance, a logical bug might allow data to bypass a `CHECK` constraint without triggering an error, silently compromising data integrity. Such issues often require specifically crafted queries to expose the vulnerabilities. Moreover, DBMS fuzzing must consider the stateful nature of databases, where the sequence of operations during testing can affect the database's state, leading to different outcomes. This complexity is crucial for uncovering issues that might remain hidden in a stateless context.

2.2.1 Types of Bugs in DBMS Fuzzing

The three main types of bugs found in DBMSs are crashes, logical bugs and performance bugs, each of which present unique challenges and require distinct methodologies for detection and resolution.

Crashes: These bugs manifest as sudden interruptions in the DBMS due to unhandled exceptions or critical errors, compromising data integrity and availability. Tools like Squirrel [29] and Griffin [6] are designed to identify such vulnerabilities.

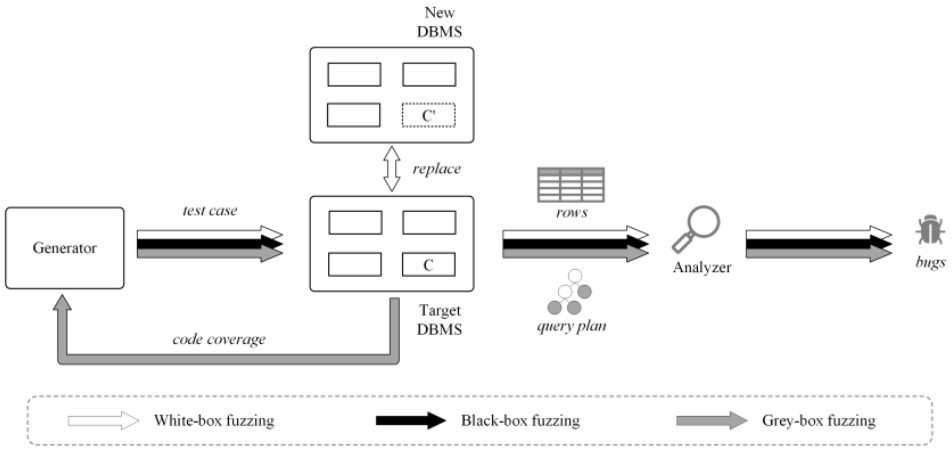

FIGURE 2: White, Black, Grey box DBMS fuzzing. Source [7]

Logical Bugs: These involve discrepancies between expected and actual DBMS behavior, leading to incorrect query results, data corruption, or security vulnerabilities. Detection requires sophisticated fuzzing techniques that can generate and evaluate complex queries. Tools like SQLRight [11], SQLancer [21], and its various methodologies (PQS [22], TLP [24], NoRec [25]) are effective in uncovering such issues.

Performance Bugs: These bugs affect the efficiency and scalability of the DBMS, leading to excessive resource consumption, slow query execution, or even denial of service. Fuzzing tools like APOLLO [9] and AMOEBA [12] focus on simulating workload scenarios to evaluate DBMS performance under various conditions.

2.3 Logical Bug Example

Listing 1 illustrates a logical bug in SQLite related to the handling of unique constraints with the NOCASE collation. This bug was detected by SQLancer using the PQS oracle [22], and has been subsequently fixed by the SQLite developers. The first statement creates a table `t0` with a single column `c0` of type INT and a UNIQUE constraint with NOCASE collation, that allows to perform case-insensitive text comparisons. The second statement inserts a value './' into the table. The third statement attempts to select rows from t0 where c0 matches './' using a LIKE clause, but it fetches no rows, which is incorrect behavior.

SQLancer, through its Pivoted Query Synthesis oracle, detects logical bugs by generating and transforming SQL queries to produce semantically equivalent forms. Discrepancies in the query results indicate potential logical bugs. In this case, the PQS oracle identified that the expected result should have fetched the row containing './', but SQLite returned no rows.

```
1 CREATE TABLE t0(c0 INT UNIQUE COLLATE
      NOCASE);
2 INSERT INTO t0(c0) VALUES ('./');
3 SELECT * FROM t0 WHERE t0.c0 LIKE './'; --
      fetches no rows
```

LISTING 1: Bug Example 1: SQL code illustrating the issue

The root of the problem lies in the way SQLite handles the LIKE operator with NOCASE collation under UNIQUE constraints. The UNIQUE constraint with NOCASE collation should treat the values case-insensitively, ensuring that the value './' is correctly matched. However, due to an oversight in the query processing logic, SQLite fails to retrieve the expected row.

To address this bug, SQLite developers had to refine the handling of NOCASE collation within unique constraints, ensuring that the LIKE operator correctly respects case-insensitive comparisons during query execution.

This logical bug can have significant security implications. For instance, it treats case-variant strings as distinct, potentially leading to incorrect query results. In scenarios where case-insensitivity is critical, such as authentication or data deduplication, this bug could result in data mismatches or unintended data exposure. Given SQLite's widespread usage, including in over 3.5 billion smartphones, the impact of such a bug could be extensive, affecting both functionality and security of numerous applications.

2.4 Oracle-based Logical Bug Detection

Detecting logical bugs in DBMS is inherently more challenging than identifying memory-related issues. Memory bugs often result in crashes or other easily detectable malfunctions [29, 6, 8]. In contrast,

logical bugs typically lead to incorrect query results without causing any immediate system failure, making them more elusive. Therefore, an oracle is essential to determine the expected outcomes and identify discrepancies.

An effective oracle serves as a reference, comparing the actual output of a DBMS to the expected correct result. For example, in the case described in Listing 1, the oracle should recognize that the expected result is a row containing `'./'`. Constructing a comprehensive, error-free oracle is a complicated task, often requiring manual analysis by several analysts.

Differential analysis provides an automated and scalable solution for logical bug detection [13, 9, 27]. Techniques such as sending the same query to different DBMSs and comparing their results can highlight inconsistencies. However, this method struggles with DBMS-specific bugs due to the diverse dialects and extensions supported by different systems. Consequently, cross-DBMS validation may miss issues that are unique to a particular DBMS.

Recent advancements have focused on constructing functionally equivalent queries to test DBMS behavior. For instance, the NoREC oracle modifies the conditions in WHERE clauses and shifts them to SELECT expressions [23]. This ensures that DBMS optimizations do not interfere with the core logic of the query, making it easier to detect inconsistencies in the results. Similarly, the TLP oracle breaks down a condition in a WHERE clause into three subqueries that evaluate the condition as TRUE, FALSE, or NULL [24]. By combining the results of these subqueries and comparing them to the original query, the TLP oracle can reveal logical discrepancies. The Pivoted Query Synthesis oracle, uses pivot rows from the database to generate test queries. PQS selects a row as a pivot and creates queries that should include this pivot row in their results [22]. If the DBMS fails to return the pivot row as expected, it signals a potential logical bug.

2.5 Advantages of The Proposed Oracle

While PQS and other advanced oracles have significantly improved logical bug detection, they are not well-suited for identifying bugs related to DBMS constraints, such as `unique` and `check` constraints. These constraints involve specific validation rules that are not directly addressed by the general query transformations used for example in PQS.

To better understand, we examine a situation where a state-of-the-art fuzzer, like PQS, fails to detect a logical bug related to constraints enforcement (Listing 2).

Consider a table with a check constraint that ensures all values in a column must be positive:

```
CREATE TABLE t1(c1 INT CHECK (c1 > 0));
INSERT INTO t1(c1) VALUES (1);
INSERT INTO t1(c1) VALUES (2);
INSERT INTO t1(c1) VALUES (-1);
```

Listing 2: Table Creation and Insertion

The PQS oracle might select a pivot row, say with the value -1, to generate test queries. The test query might look like the one showed in Listing 3:

```
SELECT * FROM t1 WHERE c1 = -1;
```

Listing 3: PQS Test Query

Given the check constraint `CHECK (c1 > 0)`, the row with `c1 = -1` should never be inserted into the table, and thus the query should return no rows. However, if there's a logical bug in the DBMS that incorrectly enforces the check constraint, the row with value `-1` might be inserted, but PQS would not necessarily detect this. PQS focuses on logical consistency based on pivot rows and the relationships between different queries, not on the specific enforcement of constraints during data manipulation. This means that PQS could detect discrepancies between expected and actual results when querying the database, but it might not generate a scenario where the constraint is directly violated because it assumes the data integrity rules (like check constraints) are correctly enforced during inserts and updates.

Our proposed oracle is designed to specifically target such constraint enforcement issues. By generating a variety of input values, including edge cases and invalid values, and attempting to insert them into the table, our oracle can directly test the enforcement of the check constraint as shown in Listing 4:

```
INSERT INTO t1(c1) VALUES (-1);
```

Listing 4: Testing Constraint Enforcement

The oracle would then evaluate whether the DBMS correctly rejects this input based on the check constraint. If the DBMS incorrectly allows the insertion of `-1`, our oracle will detect this as a logical bug because the input does not satisfy the constraint `CHECK (c1 > 0)`.

In summary, while other oracles are effective in identifying discrepancies in query results based on different parameters, they does not specifically test the enforcement of constraints during data manipulation. Our oracle, with its focus on generating and validating inputs against specific constraints, can uncover bugs related to the enforcement of constraints like `unique` and `check` constraints, providing a more comprehensive solution for ensuring DBMS reliability and correctness.

3 Approach

Our proposed approach aims to detect logical bugs specifically related to DBMS constraints, such as `unique` constraints and `check` constraints, by combining evolutionary algorithms, constraint solvers, and oracles. Logical bugs related to these constraints often go undetected by traditional testing methodologies, which focus more on general query processing and crash bugs.

An overview of the proposed approach is illustrated in Figure 3.

The process begins with input generation using an evolutionary algorithm. We developed two different grammars that are used for the generation of both tables and input queries [1]. The evolutionary algorithm evolves SQL queries through mutation and crossover techniques, creating a diverse range of possible inputs that are both syntactically and semantically valid. The diversity and validity of these queries are crucial for thoroughly testing the constraints of the DBMS.

Once the inputs are generated, they are processed by a constraint-aware oracle. This oracle is designed to understand and validate the constraints applied to the database schema, such as `unique` and `check` constraints. The oracle analyzes each generated input to verify whether it adheres to the defined constraints and predicts the expected outcomes. This validation step ensures that the generated inputs conform to the specified rules of the constraints, thereby focusing on testing the logical consistency and correctness of the DBMS.

The inputs are executed on the target DBMS, and the actual results are captured. The actual results are then comapared with the expected outcomes. Any discrepancies between the actual and expected results indicate potential logical bugs related to constraint enforcement.

The final phase of the approach involves feedback and refinement. The discrepancies identified are crucial for this process, as they indicate potential logical bugs. However, the feedback loop is not limited to just identifying discrepancies, it also incorporates other valuable insights gathered during the testing phase. Specifically, the feedback includes information on how close the generated inputs are to the values tested in the database. Additionally, the feedback considers whether specific inputs trigger errors that were not previously encountered, or if they trigger errors specifically related to the constraints. These insights are fed back into the evolutionary algorithm, guiding the selection and

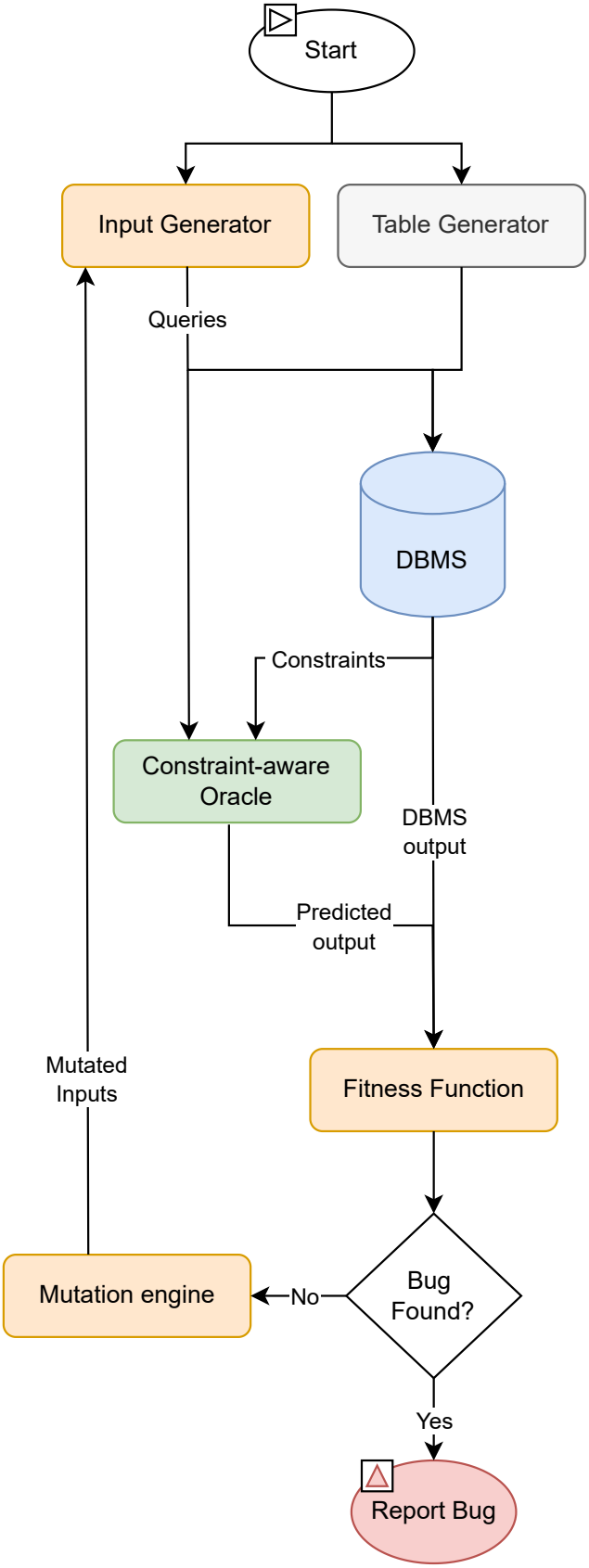

FIGURE 3: Framework Architecture Diagram

mutation of individuals that are more likely to uncover hidden logical bugs. Through this iterative process, the system continuously improves its ability to generate inputs that effectively test DBMS constraints, leading to a more comprehensive and robust solution for detecting logical bugs.

3.1 Input Generation Using Evolutionary Algorithm

Our approach to generating test inputs for detecting logical bugs in DBMS constraints leverages PonyGE2 [5], a grammatical evolutionary algorithm framework. Common fuzzing tools, such as SQLancer, typically rely on random input generation or mutation strategies to create test cases.

In contrast, our method uses PonyGE2 [5] to ensure the generation of syntactically and semantically valid SQL queries tailored for testing constraints. This involves defining a detailed grammar for SQL that guides the creation of valid and diverse queries. Our approach employs an evolutionary algorithm [14], which evolves these queries over successive generations based on their fitness scores 4.

To illustrate, we start by generating a table with constraints, using the grammar. For example:

```
CREATE TABLE t1 (
    id INT AUTO_INCREMENT PRIMARY KEY,
    c1 INT UNIQUE COLLATE utf8mb4_bin,
    CONSTRAINT v1 CHECK ( c1 > SQRT (9))
);
```

This table includes a primary key, a unique constraint on `c1`, and a check constraint `v1` to ensure all values are greater than the square root of 9.

Next, PonyGE2, using the BNF grammar, generates SQL queries to interact with this table. Initially, the generated queries are simple Insert or Update functions:

```
INSERT INTO t1(c1) VALUES (30);
INSERT INTO t1(c1) VALUES (1.5);  --
    Flagged by the check constraint
```

As the algorithm evolves, it produces increasingly complex queries that are more likely to thoroughly test the DBMS constraints. This progression is driven by the evolutionary process, where the fitness of the individuals in each generation is evaluated, and those with higher fitness scores are selected for further refinement. PonyGE2, in its default setup, uses a fixed number of generations as the stopping criterion. This means the algorithm continues to evolve solutions for a pre-defined number of generations, after which it stops.

3.2 Constraint Validation

The next step involves processing the generated inputs through a constraint-aware oracle. This oracle is capable of parsing SQL queries and applying the constraint logic defined in the database schema. It evaluates whether each input respects constraints such as unique constraints and check constraints. The oracle then predicts the expected outcome for each input, simulating the correct behavior of the DBMS.

For example, given the table creation:

```
CREATE TABLE t1 (
    id INT AUTO_INCREMENT PRIMARY KEY,
    c1 INT UNIQUE COLLATE utf8mb4_bin,
    CONSTRAINT v1 CHECK ( c1 > 10)
);
```

Listing 5: Table creation with constraints

The oracle is able to first understand that the constraint in use is that the input has to be:

```
"input" > 10
```

And then is able to parse the input query to exctract the value and test it against the constraint, for example for a queries like this:

```
INSERT INTO t1(c1) VALUES (50);
INSERT INTO t1(c1) VALUES (-2);
```

The oracle can evaluate:

```
50 > 10 -- return TRUE
-2 > 10 -- return FALSE
```

With this evaluation, the oracle recognizes if, in this case, the input violates or respect the `CHECK (c1 > 10)` constraint and should predict that the DBMS will reject or accept this insertion.

3.3 Query Execution and Evaluation

The inputs are then executed directly on the target DBMS. The results of these executions could be:

- **Success**: The query executes without any errors.
- **Random Error**: An error occurs during execution. This type of error can be generated by very particular inputs produced by the grammar that break the semantic correctness of the SQL query.
- **Specific Error Code "Error 3819: Check Constraint v1 is Violated"**: This is a specific type of error that indicates a violation of the check constraint 'v1'.This information is interesting for the fuzzing process, as it helps guide the evolution of test cases through edge cases that may or may not violate these

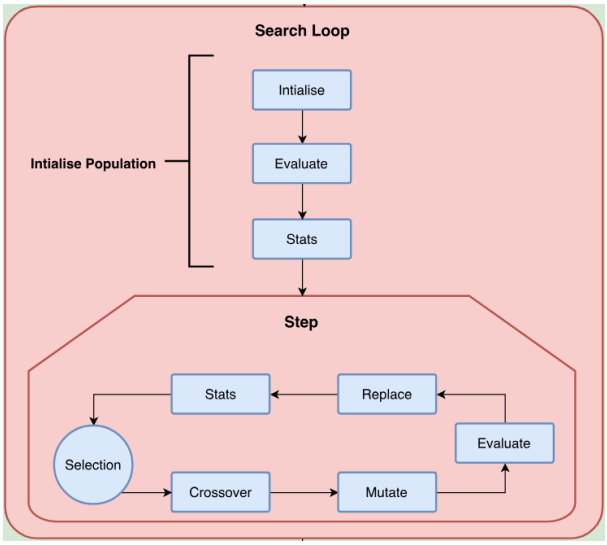

FIGURE 4: PonyGE2 control flow. Source [5]

constraints, thereby refining the input generation strategy to explore the boundaries of constraint enforcement.

These execution results are captured and compared against the expected outcomes predicted by the oracle. The oracle can return one of two values:

- **TRUE**: Indicating that the input should pass the constraints.
- **FALSE**: Indicating that the input should fail the constraints.

The comparison process to identify potential logical bugs is as follows:

- If the DBMS returns a constraint error (e.g., Error 3819) while the oracle returns **TRUE**, it indicates a discrepancy because the input should have passed according to the oracle.
- Conversely, if the DBMS executes the query successfully but the oracle returns **FALSE**, it also indicates a discrepancy because the input should have failed according to the oracle.

Any such discrepancies between the actual DBMS results and the oracle's expected outcomes are flagged as potential logical bugs (Figure 5).

3.4 Fitness Function and Feedback Loop

The final step involves feeding the discrepancies identified by the oracle back into the evolutionary algorithm. The fitness function plays a crucial role in this process by evaluating the effectiveness of the generated queries in uncovering logical bugs. The fitness function is a critical component that reacts to feedback from both the DBMS and an oracle:

- **DBMS Feedback:** This includes checking for errors, constraint violations, and the correctness of the result set produced by the DBMS.
- **Oracle Feedback:** The oracle independently validates the query results against the defined constraints, ensuring they match the expected behavior and flagging any deviations as potential logical bugs.

By guiding the fuzzing process toward inputs that are more likely to reveal logical bugs, like those that produce interesting or unexpected results, our approach maintains the syntactic and semantic validity of the queries.

For example, a fitness function might score higher for queries that are closer or farrer to the target constraint; having the table 5 (with the constraint `CHECK` ($c1 > 10$)) and queries like:

```
INSERT INTO t1(c1) VALUES (10.02355289350);
    -- High score because is neart to the
   target 10.
INSERT INTO t1(c1) VALUES (123184738294328)
   ; -- High score because is far from the
    target 10.
INSERT INTO t1(c1) VALUES (356); -- Lower
   score
```

The feedback loop allows the algorithm to refine its input generation process. By selecting and mutating the best individuals based on the feedback, the algorithm focuses on creating inputs that are more likely to reveal logical bugs in constraints. This iterative process enhances the thoroughness and effectiveness of the DBMS testing, continuously improving the detection of logical bugs.

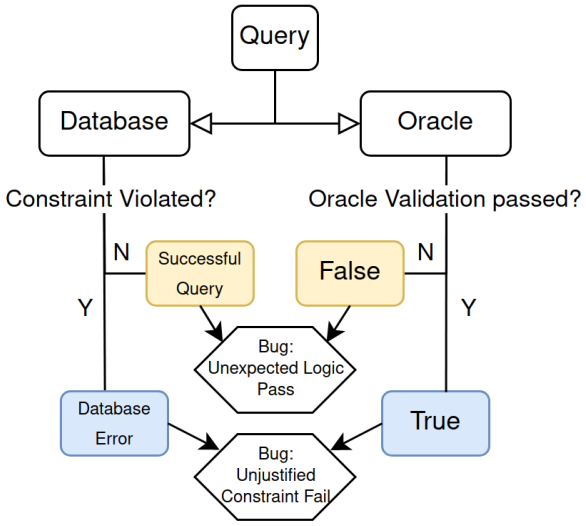

FIGURE 5: Detection of logical bug workflow: A potential bug is identified when a query successfully executes in the database yet fails the oracle validation, indicating a constraint violation missed by the DBMS. Conversely, a bug is also flagged if a query raises an error in the database but passes the oracle validation, suggesting an overly restrictive DBMS constraint.

4 Implementation Details

This chapter provides a specific look at the implementation details of our proposed system for detecting logical bugs in DBMS constraints. We will explore the key components that constitute the system architecture, detailing how each part contributes to the overall functionality. The chapter is structured to first introduce the system's architecture, followed by a discussion of the implementation specifics of each component, and finally, the interaction between these components within the system.

4.1 System Architecture

Our system architecture for detecting logical bugs in DBMS constraints is tailored specifically for MySQL-like languages [16, 17]. Although the system is highly specialized, it offers a degree of extensibility, allowing for the inclusion of new data types [18] and controls through focused effort. The architecture consists of three main components and the DBMS:

- **Table Generator**: Responsible for creating the tables in the DBMS with the appropriate constraints.
- **Constraint-Aware Oracle**: Designed to understand and validate the constraints applied to the database schema, ensuring that the generated test cases are assessed against the expected behavior.
- **PonyGE2**: Incorporates the input generator, the fitness function, and the mutation engine, driving the evolutionary algorithm that generates and refines the SQL queries used for testing.

Figure 3 illustrates the interaction between the components.

4.2 System Details

Our system's implementation is composed of several interrelated components, each designed to fulfill a specific role in the process of detecting logical bugs in DBMS constraints. These components work together to ensure that the generated SQL queries are both syntactically and semantically valid, and that they effectively test the constraints defined in the database schema.

4.2.1 Table Generator

The Table Generator is the component responsible for creating the database tables that will be used in the testing process. This generator is implemented using IslaSolver [28], a grammar-aware string constraint solver that allows for the precise definition of table schemas, including various constraints such as unique and check constraints. The generator is highly configurable, enabling the creation of tables with different data types, indexes, and constraints tailored to the specific needs of the test cases. This flexibility ensures that the system can thoroughly evaluate how the DBMS enforces constraints across a wide range of scenarios.

Grammar Design Choices

In designing the grammar for the Table Generator, several key choices were made to ensure that the generated tables and queries would be effective in testing the DBMS's ability to enforce constraints. These choices were guided by the need to reflect common scenarios in real-world applications while also challenging the DBMS with diverse and complex inputs.

One of the primary decisions was the selection of data types. The grammar includes `INT`, `VARCHAR`, and `FLOAT`, which represent a broad spectrum of data categories: numeric, textual, and floating-point values. These types were chosen for their prevalence in typical database schemas. By focusing on these key types, the grammar is capable of generating tables that are representative

of real-world applications while also exploring a wide range of potential logical bugs that may arise from the interaction of these data types with various constraints.

Another design choice was the inclusion of various arithmetic and logical operations within the grammar. Operations such as `>`, `<`, `=`, `!=` and `LIKE`, as well as functions like `ABS`, `COS`, and `LOG`, were incorporated to test how the DBMS handles complex expressions in `CHECK` constraints. By using a diverse set of operators and functions, the system is equipped to challenge the DBMS's ability to evaluate and enforce constraints under different conditions. This diversity ensures that the generated queries can uncover subtle bugs that might occur when the DBMS processes these operations in conjunction with the specified constraints.

Additionally, the grammar was designed to generate both simple and complex conditions within the constraints, such as direct comparisons (`c1 > 0`) and more involved expressions that combine functions. This choice was made to ensure that the system can test the DBMS's handling of both straightforward and more intricate constraint logic, providing a more comprehensive evaluation of its constraint enforcement capabilities.

These strategic design choices allow the Table Generator to produce a wide variety of SQL queries and table schemas, ensuring thorough testing of the DBMS's ability to enforce constraints across different data types and operations. By covering a broad spectrum of potential scenarios, the system is better equipped to detect logical bugs that may compromise the integrity of the database.

For example, the grammar might define a simple table creation command as follows:

```
"<start>": "CREATE TABLE t1 ( id INT
    PRIMARY KEY, c1 <data-type> UNIQUE
    CHECK <check-constraint>");"

"<data-type>": [
                 "INT",
                 "VARCHAR",
            ],

"<check-constraint>":[
                 "(c1 > 0)",
                 "(c1 < 10)",
                 "(c1 = 'abc'),
                 ],
```

Listing 6: BNF Grammar for SQL Table Creation

This example illustrates how the generator creates a table with a primary key and a column that enforces both a unique constraint and a check constraint.

4.2.2 Constraint-Aware Oracle

The Constraint-Aware Oracle is the core component responsible for validating the inputs generated by the system against the constraints defined in the database schema. This oracle is implemented with a custom parser and validation engine written in Python. Its primary function is to predict the expected outcomes of the database operations based on the constraints, simulating the correct behavior of the DBMS. The oracle must accurately replicate the DBMS's processing logic, particularly in handling unique constraints and check constraints, to detect any discrepancies that indicate logical bugs. The design of the oracle is DBMS-specific for MySQL, requiring specific tailoring to ensure it behaves consistently with the target.

To illustrate how the Constraint-Aware Oracle functions, consider the following table definition:

```
CREATE TABLE t1 (
    id INT AUTO_INCREMENT PRIMARY KEY,
    c1 INT UNIQUE COLLATE utf8mb4_bin,
    CONSTRAINT v1 CHECK ( c1 > 10 )
);
```

And the query to be tested:

```
INSERT INTO t1(c1) VALUES (15);
```

The oracle processes the following steps:

1. Parsing the Table Schema: The oracle identifies the `CHECK` constraint `c1 > 10` on column `c1`, extracting the operator `>` , the value `10`, and the data type `INT`.

2. Parsing the Input Query: The oracle parses the input query, noting the value `15` and its type `INT`.

3. Evaluating the Constraint: Using the extracted information, the oracle evaluates the constraint:

```
if 15 > 10:
    return True
else:
    return False
```

Listing 7: Constraint Evaluation

Since the condition `15 > 10` evaluates to `TRUE`, the DBMS should accept the insertion. This process demonstrates how the oracle simulates the DBMS's behavior to ensure that constraints are correctly enforced.

Implementation Details

The implementation of the Constraint-Aware Oracle involved an in-depth analysis of MySQL's documentation to understand the interactions between different data types, particularly in how they are compared within the context of `CHECK` constraints. The primary focus was on ensuring that the oracle could accurately predict and validate the outcomes of queries involving these constraints.

Given that `CHECK` constraints are heavily dependent on how data types interact during comparisons, I heavily studied these interactions. This involved a significant amount of manual testing and trial-and-error to identify specific behaviors and edge cases that were not explicitly covered in the documentation. Through this process, I gained insights into how MySQL handles various data types, which informed the design and logic of the oracle.

However, it is important to note that the implementation assumes that MySQL correctly handles these data types under typical conditions. This assumption could introduce a bias into the system, as the oracle's effectiveness is partly dependent on the correctness of MySQL's type handling. If MySQL itself has undocumented behaviors or bugs related to type handling, the oracle might not detect logical bugs as effectively.

4.2.3 PonyGE2 Framework

The input generation and mutation processes are managed by PonyGE2 [5] , a grammatical evolutionary algorithm programming framework. PonyGE2 plays a central role in generating SQL queries that are not only diverse but also maintain syntactic and semantic validity. The framework uses a detailed grammar for SQL, which guides the creation of queries that can effectively test the DBMS constraints.

For example, the grammar might define SQL insert operation as follows:

```
<operation> ::= <insert>

<insert> ::= "INSERT INTO t1 (c1) VALUES
    (("<values>"));"

<values>::= 0|1|2|3|4|5|6|7|8|9
```

Listing 8: BNF Grammar for SQL Operations

The evolutionary algorithm within PonyGE2 iteratively refines the generated queries based on feedback from the oracle and the DBMS. This feedback loop improve the detection of logical bugs, as it allows the system to focus on generating inputs that are more likely to expose hidden vulnerabilities. The mutation engine within PonyGE2 introduces variability into the queries by altering specific parts, such as the values used in comparisons or the structure of the SQL statements. This targeted mutation process allows the system to explore a diverse set of possible scenarios, covering a wider range of potential edge cases that could reveal hidden bugs. By combining different mutations across generations, the algorithm maintains a broad exploration of the search space, increasing the likelihood of uncovering logical bugs.

4.2.4 Integration and Feedback Loop

The integration of these components within the system, as shown in figure 3, is designed to be seamless, with each part contributing to a continuous feedback loop that enhances the effectiveness of the testing process. After the initial queries are generated and refined by PonyGE2, they are executed on the DBMS. The results are then validated by the Constraint-Aware Oracle. Any discrepancies identified between the expected and actual outcomes are flagged as possible bugs, then the interesting informations taken during the execution are fed into the fitness function, which uses this information to further refine the input generation process.

4.2.5 Fitness Function for the Evolutionary Algorithm

In evolutionary algorithms, the fitness function is the element that drives the optimization process. It provides a measure of how well an individual performs with respect to the problem's objectives. This paragraph details the construction of the fitness function used in our evolutionary algorithm, designed to evaluate test cases based on multiple criteria.

Given the complexity and varied nature of the problem, our fitness function is a weighted composite of several components, each representing a different aspect of input quality. The use of weights allows us to control the importance of each component, tailoring the fitness function to the specific goals of our DBMS testing framework.

Weighted Fitness Function

A weighted fitness function is a mathematical construct that aggregates multiple objectives into a single score, with each objective being assigned a specific weight. These weights allow the algorithm to prioritize certain goals over others.

In our context, the weighted fitness function evaluates test inputs based on:

- **Proximity to Expected Outcomes**: The similarity between the generated input and

a predefined target value present in the constraint.

- **Error Discovery**: The ability of the input to trigger new, previously undetected errors.
- **Constraint Adherence**: Whether the input respects the system's constraints.
- **Execution Time**: Unlike typical performance metrics, we prioritize slower inputs, which might indicate more complex queries or edge cases that challenge the DBMS.
- **Syntactical Correctness**: Inputs that are syntactically malformed are heavily penalized to ensure the focus remains on generating valid SQL queries.

The Fitness Function

The fitness function F used in our evolutionary algorithm is defined as shown in Figure 6:

Where:

- w_1, w_2, w_3, w_4 are the weights that determine the contribution of each component to the overall fitness score.
- $d(x, x^*)$ is the distance between the generated input x and the expected value x^*, defined as:

$$d(x, x^*) = \begin{cases} |x - x^*| & \text{if } x \text{ and } x^* \text{ are numbers} \\ L(x, x^*) & \text{if } x \text{ and } x^* \text{ are strings} \end{cases}$$

  Here, $L(x, x^*)$ is the Levenshtein distance, which measures the minimum number of single-character edits (insertions, deletions, or substitutions) needed to transform one string into the other.
- $\text{Errors}(x)$ is a binary function that rewards inputs which trigger new errors:

$$\text{Errors}(x) = \begin{cases} 1 & \text{if } x \text{ triggers a new error} \\ 0 & \text{otherwise} \end{cases}$$

- $\text{Constraint}(x)$ penalizes inputs that violate system constraints, defined as:

$$\text{Constraint}(x) = \begin{cases} 1 & \text{if } x \text{ violates a constraint} \\ 0 & \text{otherwise} \end{cases}$$

- $T(x)$ is the execution time of the input x. Unlike in typical optimization scenarios, here we prefer slower execution times as they may indicate more complex and thorough testing:

$$T(x) = \text{Execution time of } x.$$

- $\text{Syntax}(x)$ introduces a penalty for malformed SQL inputs:

$$\text{Syntax}(x) = \begin{cases} \infty & \text{if } x \text{ results in syntax error} \\ 0 & \text{if } x \text{ is syntactically correct} \end{cases}$$

Choosing the Weights

Selecting the appropriate weights w_1, w_2, w_3, w_4 is a custom problem that requires consideration of the specific goals and context of the DBMS testing framework. The relative importance of each component may vary depending on factors such as the types of bugs being targeted, and the constraints of the testing environment. The process of weight selection involves empirical testing and iteration to achieve the desired balance and to ensure that the evolutionary algorithm effectively prioritizes the most relevant aspects of input quality.

5 Experimental Validation

We evaluate our tool on real-world most popular DBMS system, MySQl, to answer the following questions:

> **Q1:** How effective is our tool in finding bugs in real world scenarios?

This question aims to evaluate the effectiveness of our tool by determining its ability to identify logical bugs in MySQL, specifically those related to `CHECK` and `UNIQUE` constraints. The effectiveness will be assessed based on the number and types of bugs detected in a real-world database setup, providing an indication of the tool's practical utility in real-world applications.

> **Q2:** What is the impact of different evolutionary algorithm parameters on the effectiveness of the fuzzing process?

This question explores how varying parameters within the evolutionary algorithm, such as the number of individuals per generation or the number of generations, affect the tool's performance in detecting logical bugs. The impact will be measured by conducting an ablation study, where the performance of the tool is evaluated under different configurations of the evolutionary algorithm. This will help determine which parameters are most critical for optimizing bug detection and constraint coverage.

$$F = w_1 \cdot d(x, x^*) + w_2 \cdot \text{Errors}(x) + w_3 \cdot \text{Constraint}(x) + w_4 \cdot (\frac{1}{1+T(x)}) + \text{Syntax}(x)$$

FIGURE 6: Fitness function used in the evolutionary algorithm.

Q3: How does our tool's bug detection capability compare to state-of-the-art tools like SQLancer?

This question focuses on comparing the performance of our tool with SQLancer in terms of the number of bugs detected, the types of bugs found, and the overall coverage of constraints tested. The comparison will involve running both tools for the same amount of time and analyzing their outputs.

Ethical and Practical Considerations in Database Testing

In the field of Database Management Systems (DBMS) testing, it is both an ethical and practical standard to use synthetically generated databases rather than real-world databases. This approach is especially critical in the context of fuzzing for logical bugs, where the testing process involves extensive manipulation of data and constraints that could lead to unintended consequences if applied to real databases containing sensitive or proprietary information.

For ethical reasons, the testing conducted in this study exclusively utilized synthetic databases. Using real databases could expose sensitive data or disrupt operational environments, leading to significant privacy concerns and potential security breaches. By employing synthetically generated schemas and data, the testing process adheres to ethical standards, ensuring that no actual data is compromised during the evaluation of the system. Furthermore, the testing methodology received approval from the ethical committee of the University of Twente, confirming that all practices followed comply with ethical research guidelines.

Beyond ethical concerns, the use of synthetic databases is also driven by practical considerations. In DBMS testing, particularly when detecting logical bugs, there are no established benchmark databases that can be universally applied across different systems. This limitation arises from the specialized nature of logical bug detection, which requires test cases tailored to the specific behaviors and constraint-handling mechanisms of each DBMS.

Logical bugs often result from complex interactions between various data types, constraints, and SQL operations, which can differ significantly across different DBMS implementations. As a result, a one-size-fits-all benchmark database does not exist, and generic benchmarks are insufficient for thoroughly evaluating the effectiveness of fuzzing tools in this context.

To overcome this challenge, the testing framework in this study relies on a Table Generator to create a diverse array of database schemas and queries. These are specifically tailored to test the DBMS's ability to enforce constraints, allowing for a comprehensive evaluation of its robustness. This approach not only adheres to ethical testing practices but also addresses the practical necessity of creating customized test environments that reflect the unique characteristics of the DBMS under evaluation.

5.1 Experimental Setup

The experiments will be conducted on a Dell XPS 15 9520 with the following specifications:

- **Operating System**: Ubuntu 22.04.4 LTS (Jammy Jellyfish)
- **Kernel**: 6.5.0-44-generic x86_64
- **CPU**: 14-core (6-mt/8-st) 12th Gen Intel Core i7-12700H
- **RAM**: 31.01 GiB

The evaluation will focus on the DBMS platform of MySQL . We will test for logical bugs in CHECK and UNIQUE constraints using our developed fuzzing tool. The environment will be controlled, and each tool will run for a fixed period to ensure consistent and comparable results.

5.2 Metrics for Evaluation

To comprehensively evaluate our tool's effectiveness, we employ a set of metrics tailored to assess different aspects of the tool's performance. Each metric is designed to answer specific research questions, ensuring that our evaluation is both targeted and thorough.

Query Generation Rate

This metric measures the number of SQL queries generated per unit of time during the fuzzing process. It provides insight into the tool's efficiency in producing diverse queries.

Relevance:

- Q1 (Real-World Scenarios): Understanding the rate at which queries are generated is crucial for assessing the tool's ability to cover a wide range of inputs in real-world scenarios.
- Q2 (Impact of Evolutionary Algorithm): Query generation rate can be influenced by different evolutionary parameters, making it a key metric for understanding how these parameters affect overall performance.
- Q3 (Comparison with SQLancer): This metric helps compare the efficiency of our tool with SQLancer by evaluating which tool can generate and test more queries in a given time frame.

Logical Bug Detection

This metric tracks the number of logical bugs detected by the tool during the testing phase. It is particularly focused on bugs related to `CHECK` and `UNIQUE` constraints.

Relevance:

- Q1 (Real-World Scenarios): This metric helps in assessing the tool's effectiveness in real-world settings by identifying the number of logical bugs.
- Q3 (Comparison with SQLancer): This metric directly addresses the effectiveness of the tool in detecting logical bugs compared to SQLancer, providing a measure of comparative performance.

False Positive Rate

This metric measures the proportion of reported bugs that are not actual bugs. A high false positive rate indicates inefficiency in the oracle's ability to accurately detect logical bugs.

Relevance:

- Q3 (Comparison with SQLancer): This metric is crucial for comparing the accuracy and reliability of our tool's bug detection capabilities. A low false positive rate indicates a more precise tool, and benchmarking this against SQLancer provides a valuable perspective on our tool's performance.

Query Validity

This metric assesses the proportion of generated queries that are syntactically correct, ensuring that the tool produces valid SQL queries that can be executed by the DBMS.

Relevance:

- Q1 (Real-World Scenarios): Query validity is critical for ensuring that the tool functions effectively in real-world scenarios.
- Q2 (Impact of Evolutionary Algorithm): This metric is relevant when adjusting evolutionary parameters to ensure that query generation remains robust and valid.
- Q3 (Comparison with SQLancer): Comparing the validity of queries generated by our tool versus SQLancer can highlight differences in the robustness of the query generation process.

5.3 Experiments

5.3.1 Detection Capabilities

To evaluate the detection capabilities of our tool, we conducted a 24-hours run using the base settings. These settings were chosen based on literature [4, 3]. The base settings used were a population size of 50, 50 generations, a crossover rate of 0.75, and 10 mutation events per generation.

This run was designed to simulate a real-world testing scenario, applying the tool to a MySQL database to assess its effectiveness in identifying logical bugs related to `CHECK` and `UNIQUE` constraints.

Results and Analysis

Over the course of the 24-hours run, the tool generated and tested a total of 69,293,700 queries. The performance of the tool was measured across several key metrics, which are summarized below.

- **Query Generation Rate:** The tool maintained an average rate of approximately 802 queries per second, demonstrating a high level of efficiency in generating and executing test cases within the given time frame.
- **Invalid Query Rate:** Out of the total queries generated, 82% were valid. This moderate rate of invalid queries suggests that while the tool is aggressive in exploring the search space, there is significant room for optimizing the query generation process to reduce the number of unexecutable queries.

- **Resource Usage:** During the run, the tool's resource usage was closely monitored. The tool consumed approximately 48% of CPU resources. Memory usage was also significant, with a maximum resident set size of 495,312 kB. Despite the intensive workload, the system's performance remained stable throughout the run.
- **Logical Bugs Detected:** The tool successfully identified 3 possible logical bugs related to `CHECK` constraints. These bugs require further investigation to confirm whether they represent real issues to the integrity of the database.

DBMS	Total Queries	Bugs Detected
MySQL	69,293,700	3

TABLE 1: Summary of Detection Results in MySQL

The results of this initial run demonstrate the tool's capacity to efficiently generate and test a substantial number of queries in a relatively short period. The resource usage metrics indicate that the tool operates effectively within acceptable system performance limits. However, the rate of invalid queries highlights an area for improvement. Reducing the invalid query rate could enhance the tool's effectiveness by increasing the proportion of valid, executable queries that directly contribute to detecting logical bugs.

Despite the invalid query rate, the detection of 3 possible logical bugs underscores the tool's potential in uncovering issues related to DBMS constraints. These findings indicate that the tool is capable of identifying scenarios where constraints might fail, but further investigation is needed to confirm the validity of the detected bugs.

5.3.2 Ablation Study

To assess the impact of various parameters within the evolutionary algorithm on the tool's performance, we conducted an ablation study. The parameters were systematically varied to observe their effects on the efficiency and effectiveness of the tool in generating and validating queries. The parameters under consideration include:

- **Population Size:** Tested values of 25, 50, and 100 were chosen to explore how the number of individuals per generation influences the diversity and quality of generated queries.
- **Number of Generations:** We tested 25, 50, and 100 generations to determine how long the evolutionary process should continue to maximize the discovery of logical bugs.
- **Mutation Events:** This parameter was varied with values of 1, 10, and 20 to assess how the frequency of mutation affects the overall tool performance.
- **Crossover Rate:** Values of 0.6, 0.75, and 0.9 were tested to understand the impact of crossover on maintaining a balance in the evolutionary process.

Results and Analysis

The results of the ablation study are summarized in Tables 2 and 3. These tables provide insights into how each parameter influences the tool's performance across two key metrics, Query Generation Rate and Query Validity. The results are summarized below.

- **Population Size:** Increasing the population size from 25 to 100 resulted in a higher number of total queries generated, with a trade-off in query validity. A larger population leads to more diverse query generation but may also introduce more invalid queries due to the increased exploration of the search space.
- **Number of Generations:** The results indicate that increasing the number of generations slightly improves the query validity, as the evolutionary process has more time to refine the solutions. However, the impact on the query generation rate is minimal.
- **Mutation Events:** A lower number of mutation events (1) resulted in the highest query generation rate. Conversely, higher mutation events (20) slowed down the generation rate and reduced validity, indicating that too much mutation may disrupts the evolutionary process.
- **Crossover Rate:** The crossover rate of 0.6 seems to good balance the query generation rate and validity. A higher crossover rate (0.9) led to a decrease in query validity, likely due to excessive mixing of genetic material, which can lead to less stable solutions.

5.3.3 Tool Comparison with SQLancer

In this section, we evaluate the performance of our tool in comparison with the state-of-the-art tool SQLancer across multiple oracles. The goal of this comparison is to assess how well our tool, which focuses on DBMS constraints such as `CHECK` and `UNIQUE` constraints, performs relative to SQLancer in terms of query generation, validity, bug detection, and false positives.

Total Queries	Query Gen/sec	Query Validity
481,200	801	84.12%

Table 2: Results for Base Parameter Configuration (Population 50, Generations 50, Crossover Rate 0.75, Mutation Events 10)

Parameter	Value	Total Queries	Query Gen/sec	Query Validity
Population Size	25	432,600	721	83.27%
Population Size	100	510,600	851	80.49%
Generations	25	424,800	708	81.68%
Generations	100	454,800	758	83.12%
Mutation Events	1	660,000	1,100	79.95%
Mutation Events	20	357,600	596	75.23%
Crossover Rate	0.6	489,000	815	84.58%
Crossover Rate	0.9	439,200	732	78.61%

Table 3: Impact of Varying Key Parameters on Tool Performance

SQLancer employs three different oracles, PQS, TLP, and NoREC, each of which detects logical bugs through query transformation and result comparison techniques. In contrast, our tool uses a custom oracle specifically designed for constraint enforcement in DBMS, focusing more narrowly on violations related to constraints.

The comparison between our tool and SQLancer is structured around the following key metrics:

- **Total Queries Generated:** Measures the raw efficiency of each tool in producing queries over a set period.
- **Query Validity:** Assesses the proportion of generated queries that are valid and executable by the DBMS. A higher validity rate indicates more efficient and focused query generation.
- **Bugs Detected:** Tracks the number of logical bugs identified by each tool, with a particular emphasis on bugs related to constraints.
- **False Positives:** Measures the proportion of incorrectly flagged bugs, helping to assess the accuracy and reliability of the oracle used.

Result and Analysis

Both tools were run on the same DBMS (MySQL)for an hour under identical conditions to ensure a fair comparison.

- **Query Generation:** SQLancer significantly outperformed our tool in terms of total queries generated. SQLancer's PQS oracle generated nearly 30 million queries, with TLP and NoREC generating 20 million and 15 million queries, respectively. In contrast, our tool generated 2.8 million queries.
- **Query Validity:** SQLancer also achieved higher query validity across its oracles. PQS had a 95% validity rate, NoREC achieved 96%, and TLP reached 84%. Our tool, by comparison, had a query validity of 83%, which is comparable to TLP but still falls short of the other SQLancer oracles. This highlights that while our tool is effective in exploring the search space, further optimization is needed to improve the ratio of valid queries.
- **Bug Detection:** While SQLancer generated far more queries with higher validity, our tool identified 1 bug related to DBMS constraints, whereas SQLancer did not detect any bugs through its oracles. NoREC detected 1 bug, matching our tool in this regard. This difference is likely because SQLancer has been widely used and has already exposed and helped correct many bugs in DBMS.
- **False Positives:** Our tool reported 34 false positives, while SQLancer had none with PQS and TLP, and 1 false positive with NoREC. This suggests that while our tool is effective at detecting bugs, its custom oracle needs further refinement to reduce incorrect bug reports.

Discussion and Conclusion

This comparison highlights the differences between our tool and SQLancer. SQLancer excels at generating large volumes of valid queries and minimizing false positives, making it a highly efficient general-purpose DBMS testing tool. However, despite generating fewer queries, our tool's specialized focus on constraint testing allowed it to detect bugs that SQLancer may miss.

Tool	Oracle	Total Queries	Query Validity (%)	Bugs	False Positives
Our Tool	Custom Oracle	2,889,600	83%	1	34
SQLancer	PQS	29,974,663	95%	0	N/A
SQLancer	TLP	20,191,952	84%	0	N/A
SQLancer	NoREC	15,789,862	96%	0	1

Table 4: Tool Comparison with SQLancer

Future improvements to our tool should focus on increasing query validity, enhancing its query generation rate, and reducing the false positive rate. While SQLancer is more efficient in broader DBMS testing, our tool proves valuable in detecting constraint-specific issues, making it a possible complement to SQLancer in DBMS testing strategies.

6 Current Limitations

The current implementation of our tool presents an approach for detecting specific logical bugs related to DBMS constraints, but several limitations need to be addressed to enhance its effectiveness and applicability.

Our tool is specialized for MySQL-like languages. Extending support to other popular DBMS platforms, such as PostgreSQL and Oracle, would increase its utility and ensure broader applicability. Each DBMS has unique features, such as different implementations of constraints and data types. These differences can affect how constraints are enforced and how logical bugs might manifest, highlighting the need for broader support to make the tool versatile across different environments. Additionally, while the constraint-aware oracle has been effective in detecting logical bugs, further refinement is needed to enhance its accuracy, especially in handling complex constraint scenarios, like those with constraints that involve intricate conditions (e.g., combine multiple columns with AND/OR logic).

Finally a significant limitation lies in the challenge of accurately emulating the behavior of a database. Databases are complex systems with intricate internal mechanisms, and emulating every possible interaction is inherently difficult. This complexity is compounded by the fact that each DBMS has its own unique implementation details, even if the SQL grammar is similar. For this reason, even if the syntax and basic operations may be similar, the way each system handles query optimization, data storage, indexing, and constraint enforcement can differ. These differences make it challenging to create a universal oracle that can effectively validate constraints across multiple platforms. The oracle must be meticulously tailored to replicate the behavior of each target DBMS accurately. Achieving this level of detail is labor-intensive and requires deep knowledge of each system's internals.

7 Future Works

There are several promising directions for future research. Continuing to refine and enhance the oracle is a critical area of future work. Achieving a level of completeness in emulating DBMS behavior would allow the oracle to be reused for testing with more complex queries and different mutation approaches. A fully developed oracle could significantly improve the accuracy and reliability of the testing process. Moreover a fully developed oracle could be used as a base for the creation of other oracles that could be used for different DBMSs.

Exploring other types of constraints is another interesting direction. While our current focus is on `unique` and `check` constraints, incorporating tests for foreign keys and other advanced constraints would provide a more comprehensive assessment of a DBMS's reliability and correctness.

Another possible direction could be to integrate this method with other fuzzers for logical bugs. By enabling these fuzzers to also focus on table constraints, they can incorporate our method to provide more refined and comprehensive testing. This integration would allow fuzzers to not only use their logical bug detection methodologies but also to validate constraints, enhancing the overall effectiveness of DBMS testing.

8 Conclusion

In this thesis, we set out to address the need for detecting logical bugs in Database Management Systems (DBMSs), particularly those related to constraints such as `unique` and `check` constraints. These constraints are essential for maintaining data integrity and consistency, yet their complex nature makes them difficult to thoroughly check with generic testing tools.

To tackle this challenge, we proposed a novel approach combining evolutionary algorithms and an oracle to detect logical bugs specifically in DBMS constraints. Our approach uses an evolutionary algorithm with mutation to guide the generation of new inputs, which are then validated by a constraint-aware oracle. This oracle is designed to understand and validate inputs against the defined constraints, ensuring that any discrepancies are identified as potential logical bugs.

8.1 Summary of Contributions

- **Oracle Design**: We developed an oracle capable of understanding and validating inputs against DBMS constraints, ensuring that generated test cases reflect the expected behavior.
- **Fuzzing Framework**: We introduced a fuzzing framework specifically designed to detect logical bugs related to DBMS constraints, which are often overlooked by traditional testing methodologies.
- **Evaluation**: We evaluated our framework by testing it on real-world DBMS systems, specifically MySQL, to assess its ability to detect logical bugs related to constraints.

8.2 Summary of Results

To address the research questions, we evaluated the tool in three ways: with a detection capabilities evaluation, through an ablation study, and in comparison with SQLancer.

- **Standalone Evaluation:** The tool generated 69,293,700 million queries with a validity rate of 83%, identifying 3 constraint-specific bug related to `CHECK` and `UNIQUE` constraints.
- **Ablation Study:** Varying parameters in the evolutionary algorithm impacted query generation rate, validity, and bug detection. Larger populations and lower mutation rates led to higher query validity, while more mutations reduced precision but increased exploration.
- **Comparison with SQLancer:** SQLancer generated significantly more queries (up to 30 million) with higher validity rates (up to 96%). However, our tool was more effective at identifying constraint-related bugs, which SQLancer's oracles missed.

In total, we identified 4 possible bugs, which will be sent to the MySQL team for verification.

8.3 Summary of Future Work

We acknowledged several limitations in our current implementation and identified promising directions for future research. Refining and enhancing the oracle to achieve complete emulation of DBMS behavior will significantly improve the accuracy and reliability of the testing process. Expanding support to other DBMS platforms and exploring advanced constraints such as foreign keys will provide a more comprehensive assessment of DBMS reliability and correctness.

8.4 Vision for the Future

Looking forward, there are several promising directions for the continued development of our tool. By building on the insights gained from this research, we aim to contribute to the broader field of DBMS testing, ultimately supporting the development of more robust and secure database management systems.

In conclusion, while our tool provides a solid foundation for detecting logical bugs, we see significant potential for future advancements. We are committed to exploring these opportunities as they arise, with the goal of advancing DBMS reliability and security.

References

[1] *Backus-Naur Form*. URL: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form.

[2] Martin Eberlein et al. "Evolutionary Grammar-Based Fuzzing". In: *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings*. Ed. by Aldeida Aleti and Annibale Panichella. Vol. 12420. Lecture Notes in Computer Science. Springer, 2020, pp. 105–120. DOI: 10.1007/978-3-030-59762-7_8. URL: https://doi.org/10.1007/978-3-030-59762-7%5C_8.

[3] A. E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. "Parameter control in evolutionary algorithms". In: *IEEE Trans. Evol. Comput.* 3.2 (1999), pp. 124–141. DOI: 10.1109/4235.771166. URL: https://doi.org/10.1109/4235.771166.

[4] A. E. Eiben and Selmar K. Smit. "Evolutionary Algorithm Parameters and Methods to Tune Them". In: *Autonomous Search*. Ed. by Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. Springer, 2012, pp. 15–36. DOI: 10.1007/978-3-642-21434-9_2. URL: https://doi.org/10.1007/978-3-642-21434-9%5C_2.

[5] Michael Fenton et al. "PonyGE2: grammatical evolution in Python". In: *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*. Ed. by Peter A. N. Bosman. ACM, 2017, pp. 1194–1201. DOI: 10.1145/3067695.3082469. URL: https://doi.org/10.1145/3067695.3082469.

[6] Jingzhou Fu et al. "Griffin : Grammar-Free DBMS Fuzzing". In: *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, 49:1–49:12. DOI: 10.1145/3551349.3560431. URL: https://doi.org/10.1145/3551349.3560431.

[7] Xiyue Gao et al. "A Comprehensive Survey on Database Management System Fuzzing: Techniques, Taxonomy and Experimental Comparison". In: *CoRR* abs/2311.06728 (2023). DOI: 10.48550/ARXIV.2311.06728. arXiv: 2311.06728. URL: https://doi.org/10.48550/arXiv.2311.06728.

[8] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. "DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 4949–4965. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming.

[9] Jinho Jung et al. "APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems". In: *Proc. VLDB Endow.* 13.1 (2019), pp. 57–70. DOI: 10.14778/3357377.3357382. URL: http://www.vldb.org/pvldb/vol13/p57-jung.pdf.

[10] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecur.* 1.1 (2018), p. 6. DOI: 10.1186/S42400-018-0002-Y. URL: https://doi.org/10.1186/s42400-018-0002-y.

[11] Yu Liang, Song Liu, and Hong Hu. "Detecting Logical Bugs of DBMS with Coverage-based Guidance". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 4309–4326. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/liang.

[12] Xinyu Liu et al. "Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries". In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 225–236. DOI: 10.1145/3510003.3510093. URL: https://doi.org/10.1145/3510003.3510093.

[13] Eric Lo et al. "A framework for testing DBMS features". In: *VLDB J.* 19.2 (2010), pp. 203–230. DOI: 10.1007/S00778-009-0157-Y. URL: https://doi.org/10.1007/s00778-009-0157-y.

[14] Sean Luke. *Essentials of Metaheuristics*. second. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/. Lulu, 2013.

[15] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (1990), pp. 32–44. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[16] *MySQL*. URL: https://www.mysql.com/.

[17] *MySQL Customers*. URL: https://www.mysql.com/customers/.

[18] *MySQL Data Types*. URL: https://dev.mysql.com/doc/refman/8.4/en/data-types.html.

[19] OWASP. *Fuzzing*. URL: https://owasp.org/www-community/Fuzzing.

[20] *PostgreSQL Clients*. URL: https://wiki.postgresql.org/wiki/%20PostgreSQL_Clients.

[21] Manuel Rigger. *SQLancer*. GitHub repository. URL: https://github.com/sqlancer/sqlancer.

[22] Manuel Rigger. "Testing Database Engines via Pivoted Query Synthesis". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Banff, Alberta: USENIX Association, Nov. 2020. URL: https://www.usenix.org/conference/osdi20/presentation/rigger.

[23] Manuel Rigger and Zhendong Su. "Detecting optimization bugs in database engines via non-optimizing reference engine construction". In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 1140–1152. DOI: 10.1145/3368089.3409710. URL: https://doi.org/10.1145/3368089.3409710.

[24] Manuel Rigger and Zhendong Su. "Finding Bugs in Database Systems via Query Partitioning". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). DOI: 10.1145/3428279.

[25] Jiansen Song et al. "Testing Database Systems via Differential Query Plans". In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2072–2084. DOI: 10.1109/ICSE48619.2023.00175. URL: https://doi.org/10.1109/ICSE48619.2023.00175.

[26] *SQLite Users*. URL: https://www.sqlite.org/famous.%20html..

[27] *SQLsmith*. URL: https://github.com/anse1/sqlsmith..

[28] Dominic Steinhöfel and Andreas Zeller. "Input invariants". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 583–594. DOI: 10.1145/3540250.3549139. URL: https://doi.org/10.1145/3540250.3549139.

[29] Rui Zhong et al. "SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti et al. ACM, 2020, pp. 955–970. DOI: 10.1145/3372297.3417260. URL: https://doi.org/10.1145/3372297.3417260.