Automated Testing of Database Engines

Project description for the AST course (Spring 2025)

To be undertaken in teams consisting of two students.

Introduction and Motivation

In this project, you will gain hands-on experience with some of the concepts you will see in the lectures of the AST course. For this project, you will build your own automated testing tool for finding bugs in database engines, specifically SQLite.

Database engines back almost every modern application. Bugs in database engines undermine the reliability of the applications and infrastructures that rely on them. We can classify bugs in database engines into two categories:

- **Crashes:** the execution of the database engine terminates abnormally (e.g., seg faults) due to undefined behaviors that happen at runtime, such as buffer overflows.
- **Logic bugs:** the execution of the database engine terminates gracefully, but it does not produce the expected results.

Here is an example of a hypothetical logic bug in a database engine. The following SQL code first creates a table with a single column and inserts a record into it. Then, a `SELECT` query is executed, and since the `WHERE` condition is always true, the inserted record is expected to be returned. However, due to a logic bug, the database instead returns an empty result set.

```
CREATE TABLE t0 ( c0 INT );
INSERT INTO t0 ( c0 ) VALUES (1);

SELECT * FROM t0 WHERE 1 = 1; -- expected: row is fetched, actual:
row is not fetched
```

Detailed Project Description

The project consists of two parts, each of which addresses a different objective.

Part 1: Automated Bug Detection in Database Engines

The objective of the first part is to develop a tool that evaluates the reliability of database engines by detecting crashes and logic bugs. You are free to design your own testing methodology to achieve this. You will definitely need to build your own SQL generator to produce “interesting” queries that are more likely to trigger bugs in database engines. Additionally, you may choose to combine your SQL generator with techniques such as differential testing, metamorphic testing, or any other method covered in the course so that you trigger logic bugs that are difficult to detect.

Targets

There are dozens of relational database engines available, including SQLite, MySQL, PostgreSQL, DuckDB, and many more. In this project, the practical focus is on **SQLite.** The reason is that SQLite is lightweight and easy to set up, without having to set up a corresponding database daemon.

The most recent version of SQLite is probably very reliable given the prior testing campaigns over the past 6–8 years [1, 2, 3, 4]. Therefore, in this project you will focus on testing historical SQLite versions. Specifically, we provide you with a Docker image called **sqlite3-test** that contains the installation of two different versions of SQLite:

- **Version 3.26.0:** This version is seven years old and contains many bugs that were discovered in previous testing campaigns but have since been fixed in more recent versions. The executable of the database engine can be found in the Docker container in the following path: **/usr/bin/sqlite3-3.26.0**.
- **Patched version on top of version 3.39.4:** This is a patched version that relies on version 3.39.4, which is newer and supports more features, such as RIGHT JOIN. We manually injected ten reproducible bugs. It can be found in the Docker container in the following path: **/usr/bin/sqlite3-3.39.4**.

To load the Docker image from the file that accompanies this project description, run:

```
docker load < sqlite3-test.tar
```

Note that the Docker image is based on Ubuntu:22.04.

Evaluation

Once you have built your SQL query generator, you will need to evaluate its effectiveness based on the following four criteria:

1. Bug-finding capability
2. Characteristics of the generated SQL queries

3. Code coverage
4. Performance

Bug-finding capability: To assess how well your tool detects bugs, you need to count the number of **unique** bugs it discovers. You should classify these bugs based on their type:

1. Crashes: Queries that cause the database engine to crash.
2. Logic Bugs: Queries that return incorrect results.

Since bug discovery is *opportunistic*, you should track all bugs found throughout the development of your tool (see next section).

The remaining metrics (i.e., test-case characteristics, code coverage, and performance) should be evaluated through a controlled experiment. This means you can either: (1) run your tool for a fixed duration (e.g., 2 hours), or run it until it generates a fixed number of queries (e.g., 10,000 queries). Once the queries are generated, you can analyze them to assess your tool in terms of test-case characteristics, code coverage, and performance (to be explained below).

Characteristics of the generated SQL queries: It is important to evaluate the quality and complexity of the SQL queries generated by your tool. To do so, you need to collect various statistics about the queries your tool produces. Key characteristics include:

1. Frequency of SQL clauses per query (e.g., SELECT, JOIN, WHERE, GROUP BY)
2. Expression depth (how deeply nested expressions are)
3. Query validity: The ratio of semantically valid SQL queries (those that execute without errors) to invalid queries (those that contain syntax or logical errors).

Code coverage: To measure code coverage, you need to determine how much of SQLite's internal code is executed by the queries your tool generates. You can use a code coverage tool to track which parts of SQLite are exercised during query execution. Higher coverage indicates that your fuzzer is testing a broader range of database behaviors. To do so, you will need to instrument SQLite with a code coverage tool, such as gcov.

Performance: An important aspect of fuzzing is performance. You want to generate as many queries as possible so that you exercise different behaviors in the database engine under test in a small period of time. The performance evaluation involves measuring the throughput of your SQL query generator. Specifically, you need to calculate the number of queries generated per minute.

Reproducers for Detected Bugs

Once you find a bug, you need to submit a reproducer that consists of the following files:

1. **original_test.sql:** This file contains the original (unreduced) SQL query that triggered the bug. It must be the exact query generated by your tool, without any manual modifications.

2. **reduced_test.sql:** This file contains a manually minimized version of original_test.sql that still triggers the bug. If no reduction is possible, it may be identical to original_test.sql
3. **test.db:** This is the SQLite database that original_test.sql and reduced_test.sql operated on when the bug was triggered.
4. **README.md**: A README file describing the expected results of the database engine vs. its actual one.
5. **version.txt:** This file records the SQLite3 version in which the bug was found.

For each bug you report, you **must** submit a corresponding reproducer as described above. Submissions will be made in a private GitLab repository, to which you will be invited. Further details will be provided soon.

Part 2: Automated Reduction of Bug-Triggering SQL Queries

Automated testing tools, like the one you will build in Part 1, often generate complex bug-triggering inputs that are difficult to debug. For example, imagine your fuzzer produces an SQL query 50 **lines long** that triggers a bug. Reporting such a lengthy query to developers is impractical and counter-productive, as it is hard to analyze and debug.

To address this, automated test input reduction helps minimize bug-triggering inputs automatically. The goal is to find a smaller version of the SQL query (e.g., one with only a few lines of code) that still reproduces the same bug. While the reduced query may not be semantically equivalent to the original, it serves the same purpose: making the bug easier to diagnose.

In this part of your project, you will build an automated reducer for bug-inducing SQL queries. Since this is a research-oriented project, you need to design again your own methodology for SQL query reduction. Your approach could be based on: (1) delta debugging (to be covered in lectures), which is a general method for minimizing arbitrary bug-inducing inputs, or (2) semantic-aware transformations, such as removing expressions from a WHERE clause while ensuring the query still returns the same results.

Command-Line Interface

Your test-case reducer needs to have the following command-line interface:

```
./reducer –query <query-to-minimize –test <an arbitrary-script>
```

Input Parameters:

1. **--query:** The SQL query that your reducer will attempt to minimize.
2. **--test:** An arbitrary shell script that checks whether the minimized query still triggers the bug.

How the reducer works:

1. The reducer iteratively tries to minimize the given query by performing various updates (e.g., deletion of tokens, expressions, etc.).
2. After such an update, the reducer invokes the test script given as input at every iteration.
3. The script executes the minimized query and determines whether it still triggers the bug.
4. The script must return an exit code:
    a. Exit code 0: The bug still occurs, meaning the reduction is valid.
    b. Exit code 1: The bug no longer occurs, meaning the last modification should be reverted.

The reducer continues iterating, applying further modifications until no more effective minimizations are possible.

Remark: You only need to implement the test script once since it is general and can be used as input for every query.

Evaluation

You need to evaluate your reducer based on two criteria: (1) quality of reduction and (2) speed of reduction. The quality of reduction refers to the percentage of reduction measured in terms of the number of tokens in the SQL query. On the other hand, to evaluate the speed of your reducer, you simply need to track the overall time needed to reduce each given test case.

Benchmarks

Your reducer will be evaluated using specific benchmark SQL queries that trigger known bugs. **These benchmarks will be provided to you later during the course of the project.**

Expected Deliverables

For **Part 1 (fuzzer),** you are expected to submit the following:

1. **Dockerfile:** A docker file that builds an image with the installation of your tool. The executable file of your tool **must** be called **/usr/bin/test-db** inside the Docker image.
2. **README.md:** A README file that provides a user guide for your tool (i.e., a description of its command-line interface).
3. **report.pdf:** A technical report that has the following format:
    a. Technical description: A technical description of the methods implemented in your fuzzer, along with all the design decisions you made and and its limitations.

    b. Evaluation: Evaluation results based on the four criteria: bug-finding capability, test-case characteristics, code coverage, and performance (as detailed in Section evaluation).
4. **bug-reproducers.zip:** If you detected bugs with your tool, you also need to submit a zip file with all bug reproducers as explained above.

For **Part 2 (test-case reducer),** you are expected to submit the following:

1. **Dockerfile:** A docker file that builds an image with the installation of your reducer. The executable file of your reducer **must** be called **/usr/bin/reducer** inside the Docker image**.** Please note that the effectiveness of your reducer will be automatically graded.
2. **report.pdf:** A technical report that has the following format:
    a. Technical description: A technical description of the methods implemented in your test-case reducer, along with all the design decisions you made and its limitations.
    b. Evaluation: Evaluation results based on the quality and the speed of reduction.

Deadlines

- **Part 1 (fuzzer): May 15 at 16:00**
- **Part 2 (test-case reducer): June 10 at 16:00**

Grading Criteria

The first part of your project (i.e., the fuzzer) accounts for **70%** of your project grade, with the following breakdown:

- Writing quality of report.pdf **(10%)**
- evaluation – bug-finding results **(20%)**
- evaluation – test-case characteristics **(30%)**
- evaluation – code coverage **(30%)**
- evaluation – performance **(10%)**

The second part of your project (i.e., the reducer) accounts for **30%** of your project grade, with the following breakdown:

- Writing quality of report.pdf (**10%**)
- evaluation – quality of reduction (**60%**)
- evaluation – speed of reduction (**30%**)

Restrictions

You are **not** allowed to use any existing fuzzing tools for database engines. This includes, but not limited to:

- https://github.com/anse1/sqlsmith
- https://github.com/sqlancer/sqlancer
- https://github.com/JZuming/EET
- https://github.com/JZuming/TxCheck
- https://github.com/AFLplusplus/AFLplusplus
- https://github.com/google/AFL

Similarly, you are **not** allowed to use existing test-case reducers for the second part of the project. This includes, but not limited to:

- https://github.com/uw-pluverse/perses
- https://github.com/csmith-project/creduce
- https://github.com/renatahodovan/picire

However, you are **free** to implement your tools in the languages of your choice.

Final Remarks

This project is deliberately open-ended and research-oriented. It is up to you how far you wish to go in terms of implementing interesting testing strategies, generation of interesting SQL queries, etc. We hope that many of you will have fun implementing adventurous tools in the time that you have. Do bear in mind that there is no such thing as a *perfect* fuzzer. As with any open-ended coursework, at some point you hit diminishing returns with respect to time invested and marks achieved.

At some point you need to make a call as to when enough is enough, and submit!

Have fun!

References

[1] Rigger, Manuel, and Zhendong Su. "Testing database engines via pivoted query synthesis." *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020.
[2] Rigger, Manuel, and Zhendong Su. "Finding bugs in database systems via query partitioning." *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020): 1-30.
[3] Ba, Jinsheng, and Manuel Rigger. "Testing database engines via query plan guidance." *2023 IEEE/ACM 45th International.*

[4] Jiang, Zu-Ming, and Zhendong Su. "Detecting logic bugs in database engines via equivalent expression transformation." *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024.