Automated Testing of Database Engines

Project description for the AST course (Spring 2026)

To be undertaken in teams consisting of two students.

Introduction and Motivation

In this project, you are expected to 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 infrastructure 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. To motivate you and illustrate what can go wrong in database engines, we briefly introduce three bug types (this list is not exhaustive) that commonly appear in database engines:

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.

Example: Here is an example of a **hypothetical** crash 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, the underlying process seg faults:

```
$ cat test.sql
CREATE TABLE t0 ( c0 INT );
INSERT INTO t0 ( c0 ) VALUES (1);

SELECT * FROM t0 WHERE 1 = 1;

$ sqlite3 < test.sql
segmentation fault (core dumped): test.sql
```

Unexpected-error bugs: The database engine returns an error/diagnostic (message) when it shouldn't. Unlike crash bugs that involve the engine abnormally terminating (or being killed), unexpected-error

bugs represent cases where the database engine stays up but incorrectly reports an error/diagnostic for an operation that should have succeeded.

Example: Here is an example of a **hypothetical** unexpected error bug in a database engine. Although the given query is both syntactically and semantically valid, the database engine mistakenly marks it as invalid.

```
$ cat test.sql
CREATE TABLE t0 ( c0 INT );
INSERT INTO t0 ( c0 ) VALUES (1);

SELECT * FROM t0 WHERE 1 = 1;

$ sqlite3 < test.sql
Parse error near line 4: no such table: t0
```

Logic bugs: the execution of the database engine terminates gracefully, it produces a result (without raising any error) but this result is not the expected one.

Example: Here is an example of a hypothetical logic bug in a database engine. The inserted record is expected to be returned. However, due to a logic bug, the database instead returns an empty result set.

```
$ cat test.sql
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

$ sqlite3 < test.sql
<empty result set>
```

Goal: In this project, you will apply the concepts from the course to systematically test the SQLite engine, discover real bugs and learn how to turn a failing test into an actionable bug report. You will look for multiple bug classes, crash bugs (e.g., segfaults), unexpected-error bugs (incorrect diagnostics), and logic bugs (wrong results). The project is split into two complementary parts:

Part 1: Automated Bug Finding in Database Engines. You will design and run automated tests that

generate and execute SQL workloads to expose failures in SQLite.

Part 2: Automated Reduction of Bug-Triggering SQL Queries. You will build and apply reduction techniques that take a complex failing SQL query and shrink it to a minimal reproducer.

The two parts will be completed independently. **Detailed instructions for Part 1 follow below. The description of Part 2 will be announced later in the semester.**

Part 1: Automated Bug Detection in Database Engines

The objective of Part 1 is to build an **automated testing tool (a fuzzer)** that can uncover bugs in a database engine, such as crash bugs, unexpected-error bugs, and logic bugs. At its core, your tool will be a **random** SQL query generator that systematically produces interesting SQL statements and queries, then executes them against a target database engine to observe failures.

What should the SQL generator look like? You are **free** to choose your testing methodology, drawing inspiration from the course material. For example, you may:

- build a generation-based fuzzer that creates SQL queries from scratch (e.g., using a grammar), or
- build a mutation-based fuzzer that starts from a seed corpus and produces new queries by transforming existing ones (e.g., rewriting clauses, changing constants, altering schema/data, etc.).

You may also combine your generator with oracle techniques discussed in the course, such as differential testing, metamorphic testing (details will be provided later in the semester), or other methods to help detect subtle logic bugs that do not manifest as crashes.

Key thing to remember: There is no single “right” or “wrong” testing methodology. In the literature (see bibliography below), you will find dozens of approaches for uncovering bugs in database engines. Most are **complementary**: some are better at stressing specific components (e.g., the query optimizer), while others focus on different parts of the system (e.g., transactions, etc.). As emphasized earlier, the aim of this project is primarily to apply the concepts from the lectures of the course in a real-world setting: designing a practical testing approach, evaluating it, and potentially uncovering new bugs.

Targets

There are tens 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 7–10 years [1, 2, 3, 4]. Therefore, in this project you will focus on testing a historical SQLite version. Specifically, we provide you with a Docker image called **sqlite3-test** that contains the installation of:

- **Patched version on top of version 3.39.4:** This is the **target version on which you are expected to find bugs.** It is based on SQLite 3.39.4 and has been manually patched with 25 reproducible bugs.. We manually injected **25 reproducible bugs**. These bugs include different categories such as crashes, unexpected-error bugs, and logic bugs, and they vary in difficulty. Some can be triggered with a single SQL statement, while others may require up to ten SQL statements. Overall, our patched version contains **at least 25 known bugs**, and it may also contain additional bugs we are not aware of. It can be found in the Docker container in the following path: **/usr/bin/sqlite3-3.39.4**.
- **Source code of patched sqlite version:** The source code of the target database engine is available at **/home/test/sqlite3-src**. You may use it to instrument SQLite with gcov in order to measure code coverage (explained later).
- **Sqlite (version 3.51.1).** This is the latest unmodified ("vanilla") version of SQLite. You may use it if your approach involves differential testing. It is available at **/usr/bin/sqlite3.**
- **Seed corpus:** We provide a seed corpus of 79 SQL files that you may use if you choose to implement a mutation-based fuzzer. Using this corpus is **optional**: for example, if you build a generation-based fuzzer that produces SQL queries from scratch, you do not need it. The corpus is included for convenience and is available inside the provided Docker image at: **/home/test/seeds**

To load the Docker image, run:

```
docker pull theosotr/sqlite3-test
```

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

Evaluation

Once you have built your fuzzer, 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. Since bug discovery is *opportunistic*, you should track all bugs found throughout the development of your tool. 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 file that **original_test.sql** and **reduced_test.sql** were executed against when the bug was triggered. If both scripts fully set up the required database state themselves (e.g., they create all tables and insert any needed data), this file may be empty or unnecessary.
4. **README.md**: A README file describing the expected results of the database engine vs. its actual one. Specifically, README.md should have the following structure:

````
```
## Summary

<!--
Explain briefly what goes wrong and explain why you believe this is a
bug and not the intended behavior of SQLite (if it is not a crash).
-->

## Minimized query

``` sql
Code of the bug-triggering SQL query
```

## Actual output

```sql
// TODO add output here
```

## Expectation

```
````

Each of your bug reports should be human-reviewed. Please do not simply dump everything your fuzzer flags without review as a bug.

Remark: Don't panic if you don't manage to find a bug during your testing efforts. This does not mean you have failed the project. Your work will be evaluated on multiple aspects (see below). If you complete the required evaluation and document your methodology and results, you can still do very well, even if no bugs are found.

Characteristics of SQL Queries

The remaining metrics (i.e., test-case characteristics, code coverage, and performance) should be evaluated through a controlled experiment. This means that you need to run your tool 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 assess the quality and complexity of the SQL workloads produced by your tool. To do so, collect and report statistics for 10,000 generated queries, including the following:

1. **SQL keyword coverage and frequency.** Using the list of SQLite keywords provided in this link, report:
    a. Coverage: in how many of the 10,000 queries each keyword appears at least once (hint: you may display the top-30 most frequently used keywords), and
    b. Average frequency: the mean number of occurrences of each keyword per query.

    The goal of these measurements is to approximate which SQL features your fuzzer exercises (e.g., JOIN, WHERE, VIEW, etc.).

2. **Query validity.** Report the proportion of queries that are successfully executed versus those that are invalid. Specifically, count how many of the 10,000 queries execute without any SQLite error, and how many fail due to syntax errors or semantic/runtime errors (e.g., missing tables/columns, constraint violations).

Remark: We do not expect your fuzzer to support every SQLite keyword. The purpose of this analysis is to provide a better understanding of which SQL features your tool exercises, and how broadly it explores the SQLite language surface.

Code Coverage

Code coverage is a useful metric because it approximates how thoroughly your generated queries exercise SQLite's implementation. A fuzzer that reaches more lines, branches, and functions is more

likely to explore diverse behaviors and, therefore, more likely to expose bugs (though high coverage alone does not guarantee bug finding).

To measure coverage, instrument SQLite with a coverage tool (in particular, gcov) and execute your 10,000 generated queries. Then report the overall SQLite coverage results:

1. Line coverage
2. Branch coverage
3. Function coverage

Performance

An important aspect of any fuzzing tool is performance. Ideally, 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. Based on the generated 10,000 queries, you need to calculate and report the following:

1. The number of queries generated per minute.
2. The number of queries that are generated and executed per minute.

Technical report

You are expected to prepare a technical report that documents the design of your fuzzer and your testing methodology and present the results in the aforementioned evaluation criteria. Your technical report should contain at least the following:

- **Design and Implementation:** Document the design of your testing tool and its implementation. This includes the high-level design of your query generator and testing methodology; the test oracles you use; any notable design choices you have made.
- **Limitations and Discussion:** Current limitations of your fuzzer; notable design ideas you experimented with that did not make it into the final tool, including details of optimizations you have implemented.
- **Evaluation:** Present the results from your experiments:
    - Bug-finding capabilities
    - Query generation characteristics
    - Code Coverage
    - Performance

Note that there is **no** specific length requirement for your report. However, please try to be as concise and comprehensive as possible.

Expected Deliverables

For **Part 1 (fuzzer),** you are expected to submit the following inside **a single zip file named with your team members' names** (e.g. harry-potter-hermione-granger.zip):

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. You **must include** all source files for your project in the zip file. You are **not allowed** to download the source files for your tool from any external resource (e.g., GitHub).
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/:** If you detected bugs with your tool, you also need to include a directory with all bug reproducers as explained above.

Deadlines

- **Part 1 (fuzzer): May 15 at 16:00**

Grading Criteria

The first part of your project (i.e., the fuzzer) accounts for **60%** of your project grade, with the following breakdown:

- Writing quality of report.pdf **(10%)**
- evaluation – bug-finding results **(30%) –** this includes the clarity of your bug reports
- evaluation – test-case characteristics **(15%)**
- evaluation – code coverage **(30%)**
- evaluation – performance **(15%)**

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
- https://github.com/PSU-Security-Universe/sqlright

However, you are **free** to use any libraries that help with your implementation (e.g., SQL parsing libraries and related tooling, etc.). You may also implement your tool in any programming language 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

Here's a non-exhaustive list of some useful 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] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. *In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).*
[3] 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.
[4] Ba, Jinsheng, and Manuel Rigger. "Testing database engines via query plan guidance." *2023 IEEE/ACM 45th International.*
[5] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of DBMS with coverage-based guidance". *in 31st USENIX Security Symposium (USENIX Security 22).*

[6] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. *In Proceedings of the 45th International Conference on Software Engineering (ICSE '23).*
[7] 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.
[8] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data 2, 3, Article 188 (June 2024).*
[9] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proc. ACM Manag. Data 3, 1, Article 24 (February 2025).*