# Automated Testing of Database Engines

*Project description for the AST course (Spring 2026)*

**To be undertaken in teams consisting of two students (the same groups as project 1).**

## 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 2 follow below.**

## Part 2: Automated Reduction of Bug-Triggering SQL Queries

Automated testing tools, such as the one you build in Part 1, typically produce complex bug-triggering inputs that are hard for developers to read, debug, and use for root-cause analysis. For example, a fuzzer may produce a 50-line SQL query that triggers a bug. Reporting such a long query upstream is impractical: it is hard to inspect, and there is no easy way to tell which parts are essential to the failure.

Automated test-input reduction addresses this problem by shrinking a bug-triggering input automatically. The goal is to produce a much smaller variant of the query (often just a few lines) that still reproduces the same bug. The reduced query need not be semantically equivalent to the original; it only needs to preserve the bug. Even so, a small reproducer is dramatically easier to diagnose, communicate, and fix.

In this part of the project, you will build an automated reducer for bug-inducing SQL queries. This is a research-oriented task: you are expected to design your own reduction methodology. Reasonable starting points include:

- **Delta debugging**, a general method for minimizing arbitrary bug-inducing inputs (covered in the lectures; see also references [1]–[3]);
- **Semantic-aware transformations**, for example removing predicates from a WHERE clause, dropping unused columns, or simplifying expressions while ensuring the query remains parseable.

You are free to combine these approaches or to design your own. What matters is that your reducer produces small reproducers reliably and quickly.

## Command-line interface

Your reducer must expose the following command-line interface exactly. The interface is **strict**: grading is automated, so any deviation will cause your submission to fail evaluation.

```
reducer --query <query-to-minimize> --test <oracle-script>
```

**Input parameters:**

- `--query <path>` — path to the SQL query that your reducer will attempt to minimize.
- `--test <path>` — path to a shell script (the oracle) that checks whether a candidate minimized query still triggers the bug.

**How the reducer must behave:**

1. Iteratively transform the query by applying reduction steps such as deleting tokens, removing expressions, or simplifying clauses.
2. After each candidate transformation, write the candidate query to disk (see the next section for the exact location and naming convention) and invoke the oracle script.
3. Interpret the oracle's exit code as follows:
    - **Exit code 0** — the bug still triggers; the reduction is accepted and becomes the new current query.
    - **Exit code 1** — the bug no longer triggers; the last modification is reverted.
4. Continue iterating until no further effective minimization is possible, then write out the final reduced query.

## Test-script (oracle) interface

In principle, the oracle script could follow any convention. However, to keep behavior predictable and to remain consistent with established test-case reducers such as C-Reduce [4] and Perses [5], your reducer must respect the following contract:

1. **Decoupling.** The oracle script is independent of the reducer. Anyone should be able to plug in their own oracle to check an arbitrary property (a specific crash signature, a result mismatch, a particular error message, and so on).
2. **Exit-code semantics.** The reducer assigns meaning to the script's exit code:
    - **Exit code 0** — the bug still occurs (reduction is valid).
    - **Exit code 1** — the bug no longer occurs (the last modification must be reverted).
3. **Inputs.** The script takes no command-line arguments. By default, it must read the candidate query from a file named `query.sql` in the current working directory. If the environment variable `TEST_CASE_LOCATION` is set, the script must read the candidate query from the path stored in that variable instead.

**Implication for the reducer.** Before invoking the oracle, your reducer must either (a) write the candidate query to `query.sql` in the working directory, or (b) write it to a chosen path and export `TEST_CASE_LOCATION` pointing to that path before invocation. This convention is consistent with how existing test-case reducers work.

## Evaluation

Your reducer will be evaluated on two axes:

- **Quality of reduction** — how much the reducer shrinks the input, measured as the percentage reduction in the number of SQL tokens between the original and the final query.
- **Speed of reduction** — the wall-clock time required to reduce each benchmark.

**Token counting.** To ensure all submissions are compared on a common basis, our evaluation script counts tokens with `sqlglot` (see the FAQ guide on Moodle for more details). We recommend you use the same library when computing your own evaluation numbers in the report.

## Benchmarks

Your reducer will be evaluated on 20 SQL queries that trigger bugs and inconsistencies in two historical versions of SQLite (3.26.0 and 3.39.4). Unlike Project 1, both versions are used unmodified, no manual patches have been applied. To simplify your setup, we provide a Docker image (based on Ubuntu 24.04) that contains both SQLite binaries already built:

- `/usr/bin/sqlite3-3.26.0`
- `/usr/bin/sqlite3-3.39.4`

To pull the image, run:

```
docker pull theosotr/sqlite3-reducer
```

As with project 1, you may create your own Docker image. However, the binaries of SQLite3 must be located at the designated locations in the Docker image that you will create.

The benchmarks themselves are provided in `queries.zip` on Moodle. Each benchmark (`query1/`, `query2/`, and so on) is a directory containing the following files:

- `original_test.sql` — the SQL query subject to minimization.
- `test.sql` — the oracle script for this benchmark, conforming to the interface described above.
- `oracle.txt` — a brief description of the property that makes the query interesting (i.e., the condition that signals a bug). This file is provided for your information only. Your reducer is not expected to read or parse `oracle.txt` because the oracle script alone determines whether a candidate query still triggers the bug. The contents of oracle.txt is one of the following:
    - `CRASH(<dbversion>)` — the query causes a crash on the specified SQLite version (e.g., `CRASH(3.26.0)`).
    - `DIFF` — the query produces different results when executed on SQLite 3.26.0 and 3.39.4.

**Important.** Grading is performed using the provided oracle scripts exactly as shipped. Therefore, your reducer must work correctly against them. You are, of course, free to write your own oracle scripts during development and debugging (e.g., to log intermediate state or to test against different bugs), but any such scripts will not be used during evaluation.

**Beyond the benchmarks.** Grading is based exclusively on these 20 benchmarks; we will **not** evaluate your reducer on any other inputs. That said, this is a good opportunity to play with your reducer by applying it on the bug-triggering queries you discovered in Project 1, and checking whether the output produced by your reducer is close to the minimized query created by hand. Just for fun. 🙂

## Expected Deliverables

For Part 2 (the test-case reducer),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):

- **Dockerfile**. A Dockerfile that builds an image with your reducer installed. The reducer executable **must** be located at `/usr/bin/reducer` inside the image. Grading is automated, so this path is non-negotiable. The source code of your reducer must be included directly in your submission: your Dockerfile is not allowed to clone or download source from any remote location (e.g., GitHub or GitLab). You may, however, install third-party dependencies from remote sources.
- **report.pdf**. A short technical report with the following structure:
    - **Technical description** — the methods implemented in your reducer, the design decisions you made, and the limitations of your approach.
    - **Evaluation** — your evaluation results in terms of both quality and speed of reduction across the 20 benchmarks.

## Deadlines

- **Part 2 (test-case reducer):** June 11, 17:00.

## Grading Criteria

- Writing quality of `report.pdf` — 10%
- Quality of reduction (evaluation) — 60%
- Speed of reduction (evaluation) — 30%

## Restrictions

You are not allowed to reuse an existing test-case reducer as your submission. This restriction includes, but is not limited to:

- Perses — https://github.com/uw-pluverse/perses
- C-Reduce — https://github.com/csmith-project/creduce
- Picire — https://github.com/renatahodovan/picire

You may, however, consult their published descriptions for inspiration. You are also free to implement your reducer in any programming language.

## References


[1] Andreas Zeller and Ralf Hildebrandt. *Simplifying and Isolating Failure-Inducing Input.* IEEE Transactions on Software Engineering 28(2), 2002, 183–200. https://doi.org/10.1109/32.988498

[2] Ghassan Misherghi and Zhendong Su. *HDD: Hierarchical Delta Debugging.* In Proceedings of the 28th International Conference on Software Engineering (ICSE '06), 142–151. https://doi.org/10.1145/1134285.1134307

[3] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. *Probabilistic Delta Debugging.* In Proceedings of ESEC/FSE 2021, 881–892. https://doi.org/10.1145/3468264.3468625

[4] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. *Test-Case Reduction for C Compiler Bugs.* ACM SIGPLAN Notices 47(6), 2012, 335–346. https://doi.org/10.1145/2345156.2254104

[5] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. *Perses: Syntax-Guided Program Reduction.* In Proceedings of the 40th International Conference on Software Engineering (ICSE '18), 361–371. https://doi.org/10.1145/3180155.3180236

[6] Zhenyang Xu, Yiran Wang, Yongqiang Tian, Mengxiao Zhang, and Chengnian Sun. 2025. Latra: A Template-Based Language-Agnostic Transformation Framework for Effective Program Reduction. In 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Press, 2274–2285. https://doi.org/10.1109/ASE63991.2025.00188