

# Automated Testing of Database Engines

A Mutation-Based Fuzzer Utilizing Differential Testing for SQLite

Smail Alijagic and Quy Anh Nguyen

Automated Software Testing, Spring 2026 - Part I

ETH Zürich

May 2026

## Abstract

Database engines form the backbone of most modern software. Unnoticed bugs in queries can corrupt the application state or even user data. In this project, we designed, implemented, and evaluated a mutation-based fuzzer. After generating and differentially executing 10'000 queries, our fuzzer achieved 65.26% line coverage, 50.59% branch coverage, and 71.95% function coverage, in addition to discovering 12 unique bugs in the patched binary. In this paper, we discuss the design and implementation of our fuzzer, our evaluation results, and limitations of our approach.

## 1 Introduction

SQLite is a self-contained, serverless, and ACID-compliant relational database engine [10] with likely over one trillion databases in active use [14]. The Stack Overflow 2025 Developer Survey reported SQLite as the third most popular database technology, with 37.5% of respondents [16] reportedly having made extensive use of it in the preceding year. Naturally, the security and reliability of such a popular system is critical. Nevertheless, new bugs within the system are still continuously being found and reported, notwithstanding the fact that SQLite is already being tested with a test suite that achieves 100% branch coverage as part of its development [11]. Notably, a 15-year-old database corruption bug was only discovered and fixed 2 months prior to the writing of this paper [15]. As such, continuous research effort is needed to ensure the system remain secure and reliable.

In Part I of our project, we built a mutation-based fuzzer that performs differential testing for SQLite. Given an initial seed corpus, our fuzzer iteratively applies various mutations to the seeds to create new queries. These new queries are then fed to two different binaries, and the outputs are compared. One binary is an unmodified build of SQLite version 3.51.1, henceforth referred to as the reference binary, while the other binary is a patched build of version 3.39.4 that has been manually injected with 25 reproducible bugs, henceforth referred to as the patched binary. The patched binary was built with the provided source code and instrumented with gcov to keep track of statement execution. If the mutated queries are valid, i.e. returning exit code 0 when executed by the reference binary, the fuzzer adds it back to the corpus. We also added our own custom seeds to be used alongside the provided ones with our fuzzer. Finally, we ran our

fuzzer and evaluated its bug-finding capability, test case characteristics, achieved coverage metrics, and performance.

To summarize, our contributions are:

- A mutation-based fuzzer that produces new SQL queries by iteratively applying mutations to seeds from an initial corpus. The fuzzer also performs differential testing between two binaries using the mutated queries.
- Additional seed queries to be used by the fuzzer
- A quantitative evaluation of our fuzzer

This report is structured as follows. Section 2 covers some background information and related works. Section 3 covers the high-level design of our fuzzer. Section 4 covers the lower-level details of our fuzzer implementation. Section 5 presents the evaluation results with regards to the grading criteria. Section 6 discusses limitations of our approach and alternatives we have explored. Section 7 summarizes our entire report.

## 2 Background and Related Work

### 2.1 Database Engine Failure Taxonomy

As per the project description, we recognize three types of database engine failures: **(i) crashes**, **(ii) unexpected errors**, and **(iii) logic bugs**. Crashes occur when the database engine terminates abruptly, e.g. with a segmentation fault. Unexpected errors occur when the database engine terminates with a non-success exit status. Logic bugs occur when the database engine terminates correctly but returns a wrong result. In our differential testing setting, since

the goal is to find failures of the patched binary, we define the failures as follows:

- **Crashes:** Iterations when the patched binary exits with an exit code larger than 128 excluding 130 but the reference binary does not. This stems from the bash-scripting convention that reserves exit codes larger than 128 for fatal errors; 130 is excluded due to it indicating a Ctrl-C termination [6]. We do assume that our fuzzer runs only on Unix or Unix-like systems.
- **Unexpected Errors:** Iterations when the reference binary exits with an exit code of 0 (indicating successful completion) but the patched binary does not.
- **Logic Bugs:** Both binaries terminate successfully, but their outputs differ.

## 2.2 Existing Literature

The provided literature heavily features metamorphic testing and differential testing.

- Liang et al. proposed coverage-based guidance, a mutation-based fuzzing technique that iteratively mutates queries while preserving semantic validity to improve code coverage metrics [5].
- Jiang et al. proposed Equivalent Expression Transformation (EET), a differential technique that replaces expressions inside queries with equivalent expressions and checks if the new queries produce different results from the original [4].
- Ba and Rigger introduced Query Plan Guidance (QPG), a mutation-based technique that focuses on maximizing the number of explored unique query plans, rather than coverage metrics [2]. The same authors also proposed another approach using query plans called Differential Query Plan (DQP), whereby the same query is executed under different query plans and the results are compared [3].
- Rigger and Su proposed Non-Optimizing Reference Engine Construction (NoREC), a metamorphic technique that rewrites an optimized query into an unoptimized equivalent and compares the results to flag discrepancies [8]. The same authors also proposed Query Partitioning, another metamorphic technique that rewrites one query into partitioning queries, each of which computes a subset of the original query's result. The original query's result is then compared with the composition of the partitioning queries' results to detect logic bugs [9].
- Zhang and Rigger introduced Constant-Optimization-Driven Database Testing (COD-Test), a metamorphic technique that compares the results of a complex query and an equivalent simplified query, derived by substituting parts of the complex query with their pre-computed constant results [18].

Our approach is most similar to the coverage-based guidance technique by Liang et al. We measured the coverage metrics and tweaked the fuzzer after every 10'000 iterations to improve the metrics.

## 3 Fuzzer Architecture

### 3.1 Overview

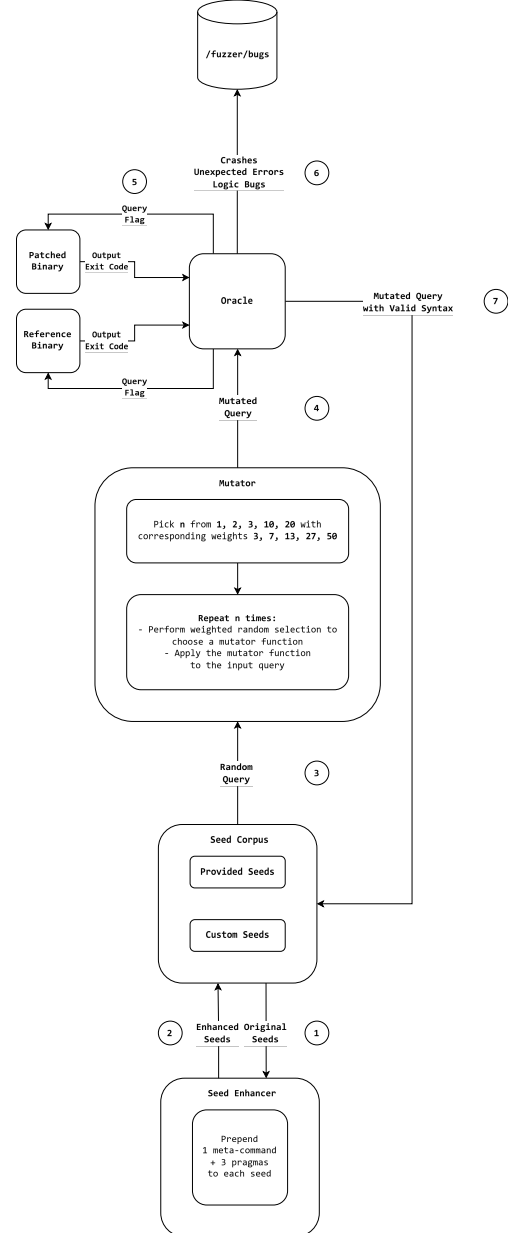


Figure 1: **High-Level Architecture**

Figure 1 shows the high-level architecture of our fuzzer. Steps 1 and 2 occur once every run, whereas steps 3-7 occur every iteration within each run.

### 3.2 Seed Corpus

The initial seed corpus contains the 79 provided seeds plus 38 of our own custom seeds.

### 3.3 Seed Enhancer

For every seed in the initial corpus, we create a duplicate query, prepend 3 random PRAGMA statements and 1 meta-command to the duplicate query, and add the result back to the corpus.

### 3.4 Mutator

For every iteration, a random query is selected from the seed corpus. We perform some random number of mutations to the selected query and then execute the mutated query on the two binaries. Both the number of mutations to apply and the mutator functions are chosen using weighted random selection. The weights were initially set based on intuition and then repeatedly tweaked by trial and error.

### 3.5 Oracle

Our oracle executes the mutated query on the reference and patched binaries. The oracle then uses the outputs and exit codes from the two binaries to decide whether the iteration was successful or resulted in some failure based on criteria defined in Section 2. Any query that results in a crash, unexpected error, or logic bug is saved to a folder, along with the outputs from the two binaries and CLI flag used (if any). Additionally, if the mutated query has valid syntax, i.e. it was successfully executed (exit status code = 0) on the reference binary, we add it back to the seed corpus. This way, as the fuzzer runs, queries become longer and more complex, thereby exploring more execution paths and (hopefully) increasing our chance of detecting failures.

## 4 Implementation

### 4.1 Programming Language, Dependencies, and Runtime Environment

- The fuzzer is implemented in Python and runs inside a `python:3.14.4-slim-trixie` Docker container. The reason we did not build directly on top of the provided `theosotr/sqlite3-test` image is because that image was built on top of the `Ubuntu:22.04` image, and the APT repositories of Ubuntu are very slow to download from [1][17], thereby taking a long time to build. Instead, we copied the provided seeds, patched binary source code, and reference binary from the provided image into the `python:3.14.4-slim-trixie` image and ran our fuzzer there.
- We compiled the patched binary inside `python:3.14.4-slim-trixie` using `gcc` (Debian 14.2.0-19) 14.2.0 with the `-coverage -g -O0` flags so that program binary keeps track of statement execution.
- We relied on `gcover 8.6` to generate JSON coverage reports for the patched binary.

- We used the `multiprocessing` module [7] to implement process-based parallelism and improve performance. In particular, we spawn multiple worker processes to run queries on the binaries in parallel. By default, we spawn as many worker processes as there are available CPU cores on the machine.
- We included the following additional files in our image:
  - `rot13.c`: A loadable extension downloaded from the SQLite source code repository [12]. We compiled the extension using `gcc -g -fPIC -shared rot13.c -o rot13.so` inside `python:3.14.4-slim-trixie` and used the `rot13.so` file to test the `.load` meta-command as part of our queries.
  - `corrupt001.db`, `employee.db`, `manyblobs-512.db`, and `random-json.db`: These are sample database files, also downloaded directly from the SQLite source code repository [13]. We included these files to test various meta-commands e.g. `.backup`, `.import`, and `.open` as part of our queries. This is done solely to improve coverage metrics; our queries do not require any data from these `.db` files, presume any schema, or that `sqlite3` be opened with these files.
  - `additional.sql`: This file contains our own custom seeds.

### 4.2 Random-Flag Execution

In order to improve coverage metrics, during the first 300 and last 1000 iterations, we execute queries with random CLI flags of the `sqlite3` binary.

### 4.3 Command-Line Interface

To run the program, compile the image with `docker build -t fuzzer .` and run `docker run -it -v "$(pwd)/bugs:/fuzzer/bugs" fuzzer`. By default, this runs 10'000 iterations. For more options, get a shell into the container with `docker run -it -v "$(pwd)/bugs:/fuzzer/bugs" -entrypoint bash fuzzer` and run `test-db` with the corresponding flags. Table 5 shows the list of all available flags.

## 5 Evaluation

### 5.1 Experimental Setup

We ran the fuzzer 3 times, with 10'000 iterations each run, on the following machine:

```

1 Processors: 16 x AMD Ryzen AI 7 350 w/
   Radeon 860M
2 Memory: 32 GiB of RAM (30.6 GiB usable)
3 Graphics Processor: AMD Radeon 860M
   Graphics
4 Manufacturer: LENOVO
5 Product Name: 83HX
6 System Version: IdeaPad Slim 5 14AKP10

```

Coverage and query validity metrics are reported below for a sample run (specifically the second run), whereas performance metrics are averaged over three runs. The logs and flagged potential bugs of all three runs are available inside the `results` folder, whereas the bugs we believe to be unique are inside the `bug-reproducers` folder.

## 5.2 Bug-Finding Capability

We managed to discover 12 unique bugs with our tools. For instance, with the below query:

```
1 .once log.txt
2 PRAGMA parser_trace = OFF;
3 PRAGMA wal_checkpoint('RESTART');
4 PRAGMA trusted_schema;
5 .imposter off
6 CREATE TABLE T1 (
7   A VARCHAR(20) PRIMARY KEY,
8   X VARCHAR(10) UNIQUE
9 );
10 CREATE TABLE T2 (
11   A VARCHAR(20) PRIMARY KEY,
12   Y VARCHAR(10) UNIQUE
13 );
14 INSERT INTO T1 VALUES ('a', 'm');
15 INSERT INTO T1 VALUES ('b', 'n');
16 INSERT INTO T1 VALUES ('c', 'o');
17 INSERT INTO T2 VALUES ('b', 'k');
18 INSERT INTO T2 VALUES ('c', 'l');
19 SELECT A FROM T1 INTERSECT SELECT A FROM T2
20 ;
21 ALTER TABLE T2 ADD COLUMN extra_8207 CHAR
   (10);
```

The reference binary produced the following output:

```
1 0|-1|-1
2 0
3 b
4 c
```

Whereas the patched binary produced a different output:

```
1 0|-1|-1
2 1
3 b
4 c
```

## 5.3 Characteristics of Generated Queries

We analyzed the 10'000 generated queries along two axes: keyword usage and query validity.

**Keyword Coverage and Frequency:** The below table reports, for the top-10 most frequently used SQLite keywords, (a) the percentage of queries in which each keyword appeared at least once and (b) the keyword's mean number of occurrences per query. The full top-30 table is provided in the appendix.

Keyword	Coverage (%)	Avg./query
IN	99.8	48.156
TABLE	99.5	23.427
CREATE	98.7	9.153
SELECT	97.8	55.958
TO	97.7	31.935
FROM	97.4	55.319
INSERT	96.9	22.209
VALUES	96.8	29.903
INTO	96.7	32.128
AS	93.0	36.046

Table 1: Coverage and Average Frequency of the Top-10 Most Frequently Used Keywords

**Query Validity:** The table below reports the ratio of valid:invalid queries amongst the 10'000 generated queries. An invalid query is any query that causes the binary to return a non-zero exit code when executed.

Outcome	#	%
Valid Queries	2481	24.8
Invalid Queries	7519	75.2
Total	10000	100.0

Table 2: Validity of the 10'000 generated queries

## 5.4 Code Coverage

Metric	Mean
Line Coverage	65.26%
Branch Coverage	50.59%
Function Coverage	71.95%

Table 3: Coverage metrics achieved after 10'000 iterations

## 5.5 Performance

Metric	Mean	Std. dev.
Queries generated / min	4912.9	361.1
Queries generated + executed / min	3110.7	421.8
Wall-clock for 10 000 queries (s)	195.10	24.56

Table 4: Throughput of the fuzzer measured after 10'000 iterations, averaged over 3 runs

# 6 Discussion, Limitations, and Threats to Validity

## 6.1 Limitations

There are several aspects upon which our fuzzer could be improved. First, our distribution of keyword frequency is rather skewed, as keywords like SELECT and

FROM appeared over 80 times in each query on average while other keywords such as REGEXP and CURRENT\_DATE virtually did not appear at all. Second, our fuzzer produces a lot of false positives, especially for logic bugs. For instance, the `sqlite_version()` naturally produces different results for the two binaries, since they are of different versions. However, our fuzzer would incorrectly flag these different versions as a logic bug. Thus, proper handling for functions like `sqlite_version()` and `random()` could be implemented to lower the false positive count. Our tool also does not report the different types of invalid queries, e.g. crashes and syntax errors, but rather grouped them altogether. This is not because of our tool's inability to distinguish between these different types of errors, but rather because we did not manage to implement the functionality to keep track of these different types of errors under our own personal time constraints. Finally, our fuzzer relies solely on differential testing and assumes that the reference binary always behaves correctly. This means that if the fuzzer manages to produce a query that causes both binaries to behave incorrectly, the fuzzer would not recognize this failure.

## 6.2 Explored Alternatives

Initially, we attempted to build a generation-based fuzzer. We managed to build a very basic grammar-based generator, but this generator produced too many invalid queries and achieved very low code coverage, even worse than simply running all the provided seeds without any mutation. We tried to use this generator as a mutator function that simply adds randomly generated statements to an existing query, but this did not work out either as the generator would most likely cause the mutated query to have invalid syntax. Thus, we abandoned the generator all together and went with a mutation-based approach.

## 7 Conclusion

We built a mutation-based fuzzer that performs differential testing for SQLite. Our tool discovered 12 unique bugs in the patched binary, in addition to achieving 65% line coverage, 50% branch coverage, 71% function coverage. Our tool also leveraged multiprocessing for maximum performance, achieving a throughput of 4900 queries generated per minute on a machine with 16 CPU cores.

## AI Disclaimer

In this project, we used LLMs to aid in the writing of regular expressions for our mutator functions and in the debugging of our scripts.

## References

- [1] Andrew and contributors. Speeding up 'apt-get update' to speed up docker image builds, 2019.
- [2] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2060–2071. IEEE, 2023.
- [3] Jinsheng Ba and Manuel Rigger. Keep it simple: Testing databases via differential query plans. *Proceedings of the ACM on Management of Data*, 2(3):1–26, 2024.
- [4] Zu-Ming Jiang and Zhendong Su. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on operating systems design and implementation (OSDI 24)*, pages 821–835, 2024.
- [5] Yu Liang, Song Liu, and Hong Hu. Detecting logical bugs of {DBMS} with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4309–4326, 2022.
- [6] Scott Mendel et al. Exit codes with special meanings, 2024. Accessed: 2026-05-14.
- [7] Python Software Foundation. *multiprocessing — Process-based parallelism*. Python Software Foundation, 2024.
- [8] Manuel Rigger and Zhendong Su. 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*, pages 1140–1152, 2020.
- [9] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.
- [10] SQLite Consortium. About SQLite. <https://sqlite.org/about.html>. Accessed: 2026-05-07.
- [11] SQLite Consortium. How SQLite is tested. <https://sqlite.org/testing.html>. Accessed: 2026-05-07.
- [12] SQLite Contributors. `rot13.c` — loadable extension implementing the `rot13()` substitution function. <https://sqlite.org/src/file/ext/misc/rot13.c>, 2024. Part of the SQLite source tree, `ext/misc` directory; Accessed: 2026-05-15.
- [13] SQLite Contributors. Sqlite sample database files, 2024.
- [14] SQLite Development Team. Most deployed software in the world, 2024. Accessed: 2026-05-14.

- [15] SQLite Development Team. *Write-Ahead Logging (WAL)*. SQLite, 2024. Accessed: 2026-05-14.
- [16] Stack Overflow. Technology | 2025 stack overflow developer survey, 2025. Accessed: 2026-05-14.
- [17] u/LinuxIsFree. Is the ubuntu apt repo slow right now or just us?, 2025.
- [18] Chi Zhang and Manuel Rigger. Constant optimization driven database system testing. *Proceedings of the ACM on Management of Data*, 3(1):1–24, 2025.

Flag	Default	Description
-seeds	/home/test/seeds	Directory containing seed .sql files for mutation.
-buggy	/home/test/sqlite3-src/build/sqlite3	Path to the buggy SQLite binary under test.
-reference	/usr/bin/sqlite3	Path to the reference SQLite binary for comparison.
-count	10000	Number of queries to generate and execute.
-mutate-timeout	0.5	Per-mutation timeout in seconds (prevents hanging mutations).
-max-query-length	100000	Hard character limit for generated queries (longer queries reset to seed).
-workers	CPU count	Number of parallel <code>check()</code> worker threads.
-validate-seeds	False	Run an upfront validation pass on all seeds before fuzzing.
-run-baseline	False	Run seeds without mutation (baseline coverage measurement).

Table 5: Available command-line flags for the fuzzer.

Keyword	Cov. (%)	Avg./q	Keyword	Cov. (%)	Avg./q	Keyword	Cov. (%)	Avg./q
IN	99.8	48.156	ON	92.1	20.721	IF	73.8	8.659
TABLE	99.5	23.427	WHERE	90.0	33.661	ALTER	70.0	1.777
CREATE	98.7	9.153	NO	86.2	3.031	ORDER	68.5	7.566
SELECT	97.8	55.958	OR	85.4	11.790	INDEX	65.5	5.946
TO	97.7	31.935	IS	84.6	15.968	COLUMN	63.9	6.422
FROM	97.4	55.319	NULL	83.2	20.864	AND	63.2	6.645
INSERT	96.9	22.209	PRAGMA	82.6	8.529	UPDATE	62.1	4.585
VALUES	96.8	29.903	NOT	82.6	16.137	BEGIN	60.7	3.998
INTO	96.7	32.128	BY	79.7	11.981	WITH	60.3	1.117
AS	93.0	36.046	EXISTS	75.6	9.296	DROP	59.8	4.466

Table 6: Coverage and Average Frequency of the Top-30 Most Frequently Used Keywords