Automated Testing of Database Engines

FAQ (Part 1)

> Q1: We are lost. How should we get started with the project?

Relax 🙂 We recommend adopting an iterative approach.

1. **Start with planning.** Think about the overall design of your fuzzer. Will it be generation-based or mutation-based? Will it be black-box or grey-box? What components will it need (query generator, executor, oracle)?
2. **Build a minimal end-to-end prototype.** Implement a very simple version that works from start to finish. For example, a generation-based fuzzer might initially support the creation of simple database schemas and basic SELECT statements consisting of WHERE clauses, and nothing else.
3. **Run it against the target database.** Even a basic prototype helps you validate your setup, measure performance, and identify early issues.
4. **Iteratively extend and refine.** Gradually add features (e.g., JOIN, VIEW, aggregates, indexes, GROUP BY, etc.) and improve bottlenecks (e.g., generation speed, logging).
5. **Enhance with testing oracles.** Once the core pipeline is stable, experiment with oracle-based techniques such as metamorphic testing, differential testing, something else(?) to uncover more subtle bugs, especially logic bugs.

> Q2: Are there any recommendations for a programming language to implement the project in? Is there a specific advantage in using a specific language or type of language?

It often takes less time to develop prototypes using languages, such as Python or JavaScript. However, if you feel more prolific in a statically-typed language, then go for it.

It's very important that you enjoy the project. So picking the language and the programming environment that you like will make the experience more enjoyable. 🙂

> Q3: Are we expected to build a fuzzer that exercises every aspect of SQLite (e.g., every keyword)?

Absolutely not! You have roughly 2.5 months for this project. It is neither realistic nor expected that you cover the entire SQL language or every SQLite feature.

Instead, focus on depth over breadth. Prioritize features that are more likely to expose interesting behavior and subtle bugs (e.g., JOINs, VIEWs, aggregates, subqueries, indexes). A fuzzer that thoroughly exercises a meaningful subset of features is far more valuable than one that superficially touches everything. Your goal is not completeness.

> Q4: How are we supposed to interact with the system under test (SQLite)?

The simplest way to interact with SQLite is through the provided binary in the project's Docker image.

A straightforward workflow is the following: whenever your fuzzer generates a query (or a sequence of queries), write it to a file (e.g., test.sql) and then invoke the patched SQLite binary as follows:

```
/usr/bin/sqlite3-3.39.4 < test.sql
```

That said, if for any reason, you prefer to interact with SQLite through its API (e.g., by including its development header files), you are free to do so.

Q5: For those who employ differential testing, which SQLite version should we use as the reference implementation (oracle) when testing sqlite3-3.39.4?

Please use the unmodified (vanilla) SQLite version 3.51.1, which is provided in the Docker image. This version serves as the reference implementation for comparison.

Q6: Can we extract and build the patched SQLite source code on our own machine or Docker image instead of using the provided Docker image?

Yes, you are free to do so.

The purpose of the image provided by us was to ease the setup for you. But since our Docker image is already lightweight, re-building the binary of sqlite3 in your Docker image should be fairly straightforward. When doing so, please use the following commands to build the sqlite3 binary

```
cd sqlite3-src
mkdir build
cd build
../configure –enable-all
make
```

You can also use the aforementioned commands to build the reference implementation of SQLite (version 3.51.1), in case you decide to employ differential testing.

Q7: We read about the XXX technique from the YYY paper in the database testing literature and found it interesting. Are we allowed to implement this technique in our fuzzer?

Yes, absolutely. That is one of the main goals of the project: to apply theoretical concepts from the course and the research literature in practice.

You are encouraged to draw inspiration from existing techniques and even re-implement them. If you do so, make sure to clearly reference the original paper and explain how you adapted or evaluated the technique in your project.

Q8: Are we permitted to use coverage information to guide our tool (i.e. after generating some queries, see how much coverage of SQLite they achieve, and use those to rank inputs for subsequent iterations – as in mutation based fuzzing, which we learned about in the course)?

Yes, you are allowed to design either a black-box or a grey-box fuzzing tool. The choice is yours.

Q9: We decided to build a random query generator that produces SQL queries from an existing grammar. Are we required to implement a SQL grammar ourselves, or can we reuse an existing one (like sqlglot or other open-source tools)?

Yes, you can reuse an existing grammar. You don't have to build your own grammar.

Q10: The project description states that we are not allowed to use existing database fuzzers or general-purpose fuzzing tools. Are we allowed to reuse other software libraries that are not related to fuzzing?

Yes. You may reuse software libraries that are not directly related to fuzzing, such as parsing libraries, logging frameworks, utility libraries, etc.

The restriction applies only to existing fuzzing tools or frameworks. You are expected to design and implement the fuzzing/testing logic yourself.

Q11: We tried a lot, but we did not manage to find any bugs in the target system. Does it mean that we failed the project?

No! This does not mean you have failed the project. Your work will be evaluated on multiple aspects (code coverage, performance). If you complete the required evaluation and document your methodology and results, you can still do very well, even if no bugs are found.

That said, successfully uncovering bugs will positively contribute to the evaluation under the “bug-finding capability” criterion.

Q12: In the project description, we were asked to submit a `test.db` file for which the query fails. What is the purpose of this file?

If your test input query includes statements like INSERT, CREATE TABLE, etc., and it does not depend on a specific existing database schema or state, then you are not required to submit a test.db file.

However, if your SQL query assumes the presence of a specific database schema (e.g., certain tables exist) without creating it as part of the query, then you must provide a corresponding test.db file.

Q13: When submitting a reduced test case (reduced_test.sql) for our bug reports, does it need to produce exactly the same output as the original?

A minimized input is *not* always semantically equivalent to the original input. The desired property that you need to preserve is that it triggers the same bug as the original one.

Q14: Are you going to evaluate and grade the clarity of our bug reports?

Yes. Writing a clear bug report is an essential part of the project.

Imagine you discover a bug in an important real-world system and submit a report to the development team. The report must be precise, well-structured, and easy to understand so that developers can quickly reproduce and debug the issue. Similarly, you are expected to submit clear, self-contained bug reports that follow the structure specified in the project description. This includes:

1. A properly minimized, reproducible test case
2. A clear description of the observed behavior
3. An explanation of why you believe the behavior constitutes a bug

Clarity in reporting will be part of the evaluation of “bug-finding capability”.

Q15: In the project description, there's the measure of frequency of SQL keywords per query. At what granularity should we provide this measure?

Using the 10,000 queries generated by your tool, you should measure in how many distinct queries each keyword appears at least once.

For example, if the keyword SELECT appears 10 times within a single generated query, this still counts as one occurrence for that query in the keyword coverage metric.

Q16: We read through the AST project description again and for the Dockerfile part it says: 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.

We are somewhat confused on what exactly test-db is, could you elaborate on that?

As described in the project documentation, **/usr/bin/test-db** is the location of your tool's executable inside the Docker image. This allows your fuzzer to be invoked using the command "test-db".

For example, if your fuzzer is written in Python, this could be a Python script with the shebang "#!/usr/bin/env python" at the top. If it's written in C++ or Rust, it would be the compiled executable binary. For Java, it might be a shell script that launches the JVM with the appropriate arguments, and so on.

Q17: Our technical report comes to around XXX pages (with narrow margins and 12pt font). Does this seem appropriate? Is there a specific length requirement we should be aware of?

No, there is no page limit for your technical report. The only requirement is that your report follows the structure described in the project description. That said, try to keep your report concise and comprehensive.

Q18: Do both project members have to upload the submission on Moodle? Or is it enough if one of us uploads it?

One is sufficient.