Testing Plan: SQamL

Group Members:

- Simon Ilincev
- Andrew Noviello
- Alex Noviello
- Eashan Vagish

Note about Number of Tests

While the OUnit test suite has 36 testing functions, each function has several unique tests, and thus, we have a lot more than 50 passing tests.

Overview of System

SQamL is a database management system that mimics SQL. It has a few main components:

1. **Parser/Lexer:**
   This takes human-inputted queries and tokenizes them based on keywords and other syntax. For example, the SQL query
   SELECT * FROM users WHERE id = 1;
   would get parsed into identifiers such as SELECT, FROM, and WHERE, table names and fields such as users and id, operators such as =, and other tokens such as numbers.

2. **Interpreter:**
   This component takes the parsed identifiers and decides which operations need to be conducted. For example, in the query above, we know we need to select all entries from the table users where a certain condition is met. The interpreter does operations such as constructing a predicate function and calling our data structure's relevant functions.

3. **Data Structure/Backend:**
   This component stores the actual data and has functions for operations such as inserting, selecting, deleting, updating, creating, and displaying itself. The data structure also handles writing to files and persisting data between users' SQamL sessions.

4. **Frontend:**
   Our TUI involves the user typing in certain SQL commands, and our system performing the relevant operations/giving the user a response.

Testing Plan

Our testing suite involves unit tests for the components mentioned above, integration tests that tested interactions between different components, and finally end-to-end tests for the entire system. Most of our integration and unit tests were **glass-box** while our end-to-end tests were **black-box.**

The frontend components were tested **manually**, while the other components and logic were tested **automatically** with OUnit.

We stratify our tests by unit, integration, and end-to-end, and then further do so based on the components involved i.e. parser/lexer, interpreter, etc.

<u>**Unit Tests**</u>

1. **Parser/Lexer:**
    a. We tested if various strings were mapped to expected token lists. For instance, the query SELECT * FROM users WHERE id = 1 would get tokenized as follows: [Identifier "SELECT";.Identifier "*";Identifier "FROM"; .Identifier "users"; Identifier "WHERE"; .Identifier "id"; Identifier "="; .Identifier "1;"]
2. **Interpreter:**
    a. Creation of predicates - functions that determined a truth value - were tested as unit tests.
3. **Data Structure:**
    a. We tested all the main operations of our data structure including creating tables, inserting data, updating based on a predicate, deleting, sorting certain entries, enforcing certain primary keys, constructing the correct types for different fields based on the parsed input.
4. **Frontend:**
    a. Testing of the frontend interface was done completely manually as it proved very difficult to programmatically mock up entering a TUI session.

Integration Tests:

1. **Interpreter + Data Structure:**
    a. Tests were written to ensure that a certain list of tokens resulted in a certain operation being successfully taken on the table data structure.
    b. We also checked that our system was finding certain errors in the tokenized list such as a table being absent in our database, a certain primary key in the tokenized list already existing, deleting non-existent rows, etc.

<u>**End-to-End Tests**</u>

The end-to-end tests involved programmatically checking for intended behavior of SQL queries. The parsing and executing of queries was tested all at once and the desired state of the data structure was used to test correctness. These were solely black-box tests, queries were performed on standard SQL shells, the output was translated by hand to our output format, and we checked for equality with the output of our system. Moreover, extensive manual testing of all the different commands was done to ensure the validity of the frontend interface.

<u>**Testing Scope & Notes**</u>

To be clear, all tests (unit tests, integration tests, and E2E tests) were performed **automatically**, with the exception of the frontend interface, which was tested **manually**. Please see the file entitled INSTALL.md for details on how the manual testing was done. (In short, it demonstrates expected inputs and outputs as you interact with SQamL.)

For our automatic tests, we aimed for and achieved ~90% code coverage. (Some edge cases and functions proved difficult to fully test or redundant on account of being variable-type dependent, duplicate error messages, etc.)
In terms of modules / breadth covered — every file in the library folder, lib, was tested. Public-facing .mli files were used as the basis for which functions to test. The only **exception to this** was **storage.ml**, which was tested manually. It deals with reading and writing to the file system, and so OS permissions plus concurrency meant that we simply confirmed that this worked by ensuring manually that state was persisted across sessions (opening and closing the database).

The 41-line user-facing to-binary compilation main.ml was not automatically tested on account of being very simple and easily tested through the generated frontend SQL shell.

We know that our testing approach demonstrated the correctness of our system because of the extensive nature of tests (over 750 lines), deep automatic code coverage (>90.3%), and demonstrated manually-checked correctness of all supported input-output pairs as per INSTALL.md. Furthermore, if one compares the results of our program to that of a standard SQL program (e.g. MySQL or PostgreSQL), one will notice that functionality is identical (bar some minor framework-dependent rendering differences, additional features, etc.)