Motivation

Extensive testing is required for successful implementation of a DBMS. However, manual testing methods are usually not sufficient to cover the input space and state space of a database system. However, given that other DBMS exists and had been tested to adhere to SQL standards, it is possible to automate parts of the database testing process by comparing the output of the tested database with a pre-existing and tested “truth” database.

Functional Requirements

- Black box, differential testing framework that runs sql queries and compares output on the peloton DBMS and a chosen truth DBMS.
- Covers a reasonable portion of the sql language (sql 92 standard).
- Produces a human-readable test report that integrates well with the development tools of the peloton team.
- An exhaustive run of the framework completes within 8 hours for nightly builds.

Quality Attributes

- Usability. For the testing framework to be useful, it is essential that the tests be easily runnable and produce human-readable results and debugging information in the case of an error. *Metric: an average peloton team member should be able to figure out how to run the system, and reproduce bugs reported by the system from test output within 1 - 2 hours of reading documentation*
- Customizability. Different test configurations and plans need to be easily switchable, so that clients can limit the generation space of possible sql queries and focus on the part of the system that is important for testing. *Metric: no code change should be required to change the target and truth database, type of existing query definitions to generate, and the depth limit for generated syntax trees.*
- Maintainability / Modifiability. Considering that peloton is still in development, it is also important for the testing framework to be easily extensible and clearly documented for future additions. *Metric: the addition of a new query type should not require changing existing code, except maybe the register the newly defined top level query element in a centralized place.*
- Performance. Faster test iterations means that the test can be run more often. If the system burden is low enough, we might even be able to fit a randomized version onto CI builds. *Metric: test system latency should be negligible to database operation latency, memory and other resource consumption should be reasonable.*

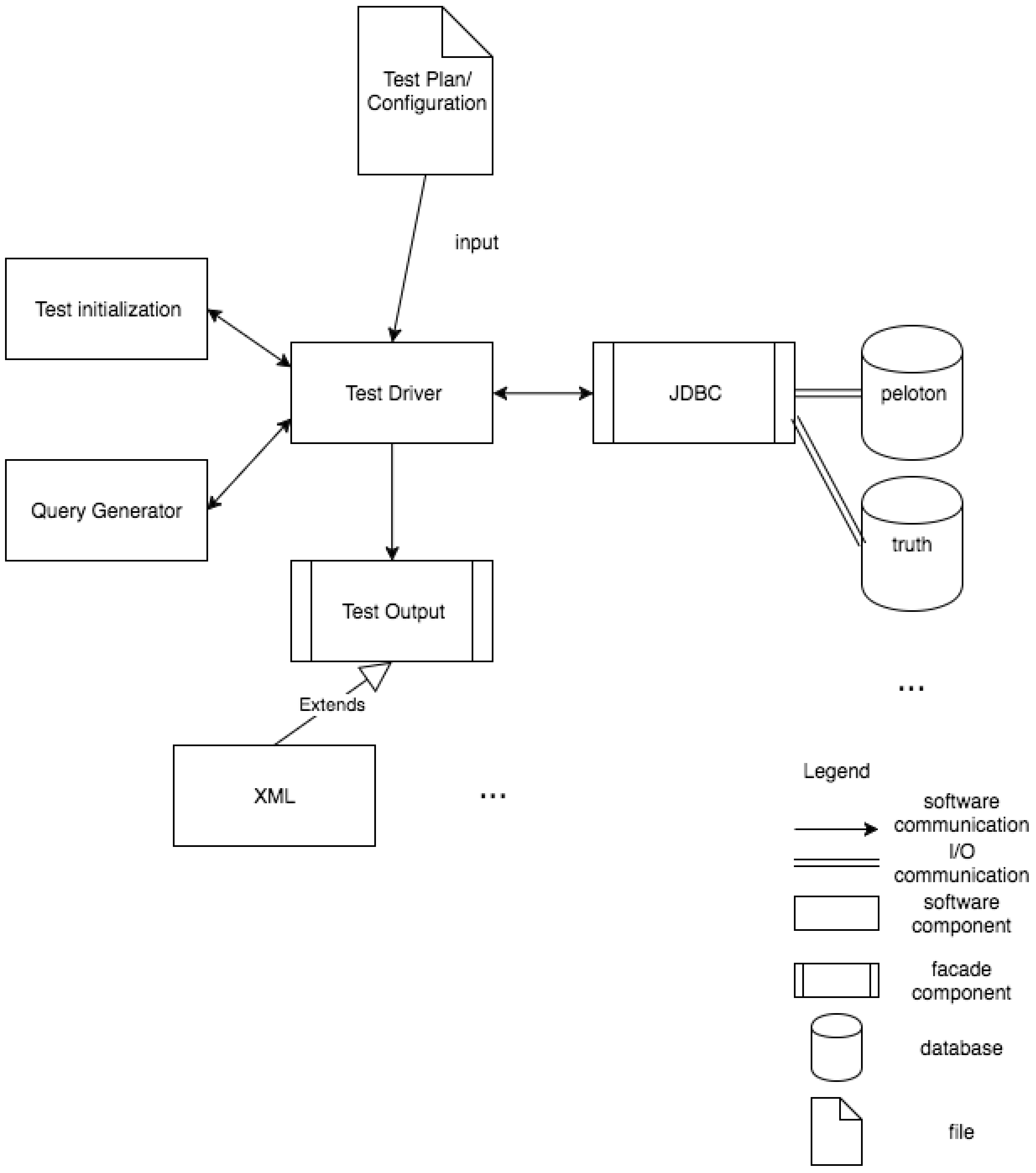
Basic project information

Language: Java 8

Project github: https://github.com/cmu-db/peloton-test

Contact: tli2@andrew.cmu.edu

Top Level Architecture Diagram



Notes on the diagram

Individual components should have clearly defined responsibilities and avoid communications with other components not shown in the architecture. Responsibility assignment for the components are documented as follows.

Test Driver

This component should be the entry point of the system. It is responsible for reading and processing client input in the format of some configuration file or plan, and initializing other components of the system with correct parameters. It establishes connections to the databases through JDBC, calls the test initialization component to populate the database, and for each generated query from the query generator, executes the query on the provided databases and log the output by calling on the test output component.

Test initialization

This component is responsible for preparing the database for query generation. At the very least, the component should understand database schemas and translate them into a format that query generator understands and populate the table with random values with the said schema. It is highly desirable that test initialization also be able to generate database schema to some extent.

Query Generator

This component is the primary component of the testing framework and should be general purpose enough for potential reuse in other circumstances. On a high level, it should understand sql grammar and be able to generate exhaustively to a specified complexity all queries that are legal. In practice, the component will be the main extension point of the system, where users are expected to implement new query grammar and load them into the testing framework. The addition of a new grammar rule should be declarative, with no additional query generation algorithm or logic, and the extension should be plug-in style without modifying other parts of the component

Test Output

The test output component is responsible for taking the test results from test driver and writing them to a client readable format. This is defined as an interface as it is independent from the rest of the system, and it makes little sense to bake in a certain type of output and prevent potential future changes.