REVERSE ENGINEERING

H2 DATABASE

RUNTIME TERROR

Test Cases

The test cases are present in the “src\test” folder. The test cases have been categorized into multiple packages.

TestAll.java

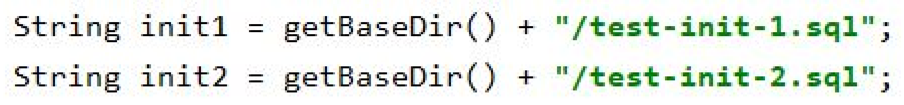
The main test application is TestAll.java. This class is the entry point for all test methods in the application. JUnit is not used because loops are easier to write in regular Java applications as most tests are run multiple times using different settings. Few examples for settings are:

- If remote database connections should be used or not
- If in-memory databases should be used or not
- If only Travis tests should be run or not and so on.

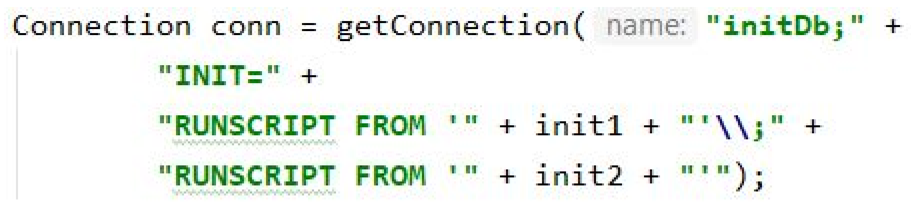
We can pass the options for various settings as command-line arguments to the main test application. Depending on the arguments passed, it will trigger the respective test cases. For example, if “travis” is passed as an argument, only Travis tests will be triggered. If arguments are not provided, it will run the tests with a number of different possible combinations of settings. We found this class interesting as this class is the entry point of H2’s custom testing framework and we could test the system by configuring various settings.

TestInit.java

This class tests INIT command within embedded/server mode that is responsible for initialising the database connection objects. The test() method in TestInit.java basically executes two SQL scripts and compares the output with the expected output. The first script creates a table and inserts some values in it. The second script only inserts values in the table that the first script creates. Then it creates a connection object that accepts the script file names as arguments. The scripts in the files will run automatically depending on the order of filenames passed as the arguments to the connection object. Here, the first script should run first and only after it is done, the second script should run. To confirm that the scripts were run as expected, we load the data that was just inserted and compare it with the expected output. We found this class interesting because it checks the case where database connection objects were created by passing the names of script files (files with ".sql" extension) as arguments that will automatically run the scripts after creating the connection object.



```java
String init1 = getBaseDir() + "/test-init-1.sql";
String init2 = getBaseDir() + "/test-init-2.sql";
```

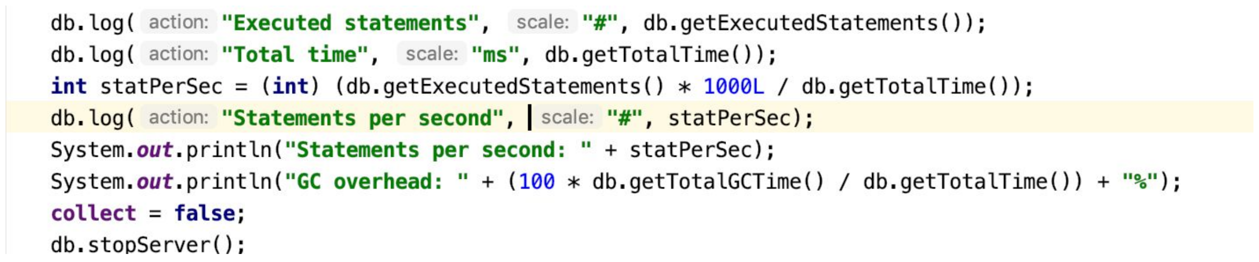


```java
Connection conn = getConnection( name: "initDb;" +
        "INIT=" +
        "RUNSCRIPT FROM '" + init1 + "'\\;" +
        "RUNSCRIPT FROM '" + init2 + "'");
```

TestPerformance.java

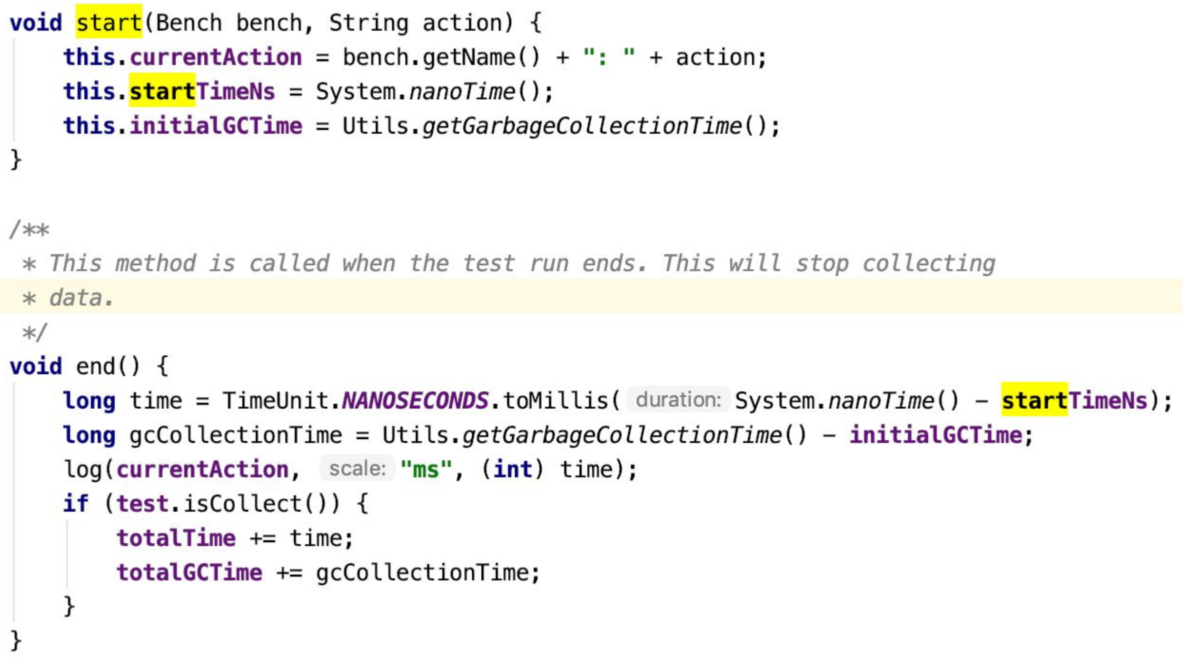
A database must be very fast while querying data for a user. The user should not wait for a time when they have made a query to get results. In every real-world software system(a large scale one in specific, like our example of H2 database), benchmarking is an important step, because it helps us measure(in a sense) how our system compares to the industry standards. Benchmarking can help identify potential areas of improvement.

- This particular test helps users benchmark the number of statements getting executed fully per second(see figure below)



```
db.log( action: "Executed statements",  scale: "#", db.getExecutedStatements());
db.log( action: "Total time",  scale: "ms", db.getTotalTime());
int statPerSec = (int) (db.getExecutedStatements() * 1000L / db.getTotalTime());
db.log( action: "Statements per second", scale: "#", statPerSec);
System.out.println("Statements per second: " + statPerSec);
System.out.println("GC overhead: " + (100 * db.getTotalGCTime() / db.getTotalTime()) + "%");
collect = false;
db.stopServer();
```

- These lines are present in TestPerformance.java(Lines 241 - 248)
- The actual nanoTime calculation happens in the method start() and end() in test class Database.java



```
void start(Bench bench, String action) {
    this.currentAction = bench.getName() + ": " + action;
    this.startTimeNs = System.nanoTime();
    this.initialGCTime = Utils.getGarbageCollectionTime();
}

/**
 * This method is called when the test run ends. This will stop collecting
 * data.
 */
void end() {
    long time = TimeUnit.NANOSECONDS.toMillis( duration: System.nanoTime() - startTimeNs);
    long gcCollectionTime = Utils.getGarbageCollectionTime() - initialGCTime;
    log(currentAction,  scale: "ms", (int) time);
    if (test.isCollect()) {
        totalTime += time;
        totalGCTime += gcCollectionTime;
    }
}
```

- The benchmark also tests how efficiently the system uses memory, We inferred this since the test calculates total garbage collection time as well. If our system reports an “out of memory error” it means that the garbage collector is spending more time cleaning up resources, but is only able to reclaim very little heap space. Now if this error is encountered, we could check where in the system are the objects using the heap, and then drill down further to understand.

Experience

We are quite familiar with test cases written using frameworks like JUnit, Mockito, etc. We didn't know that test cases could be written as regular Java applications. H2 uses a custom test framework where test cases were written as regular Java applications because loops were easier to write in Java as most tests are run multiple times using different settings. We also have learned that the application runs test cases in different settings by executing TestAll.java class.

We also learned that we can create database connection objects by passing the names of script files (files with ".sql" extension) in the argument so that it will automatically execute the scripts one by one depending on the order of the script file names after creating the connection object.

Although we knew about benchmarking, it is only after going through these tests that we realized the importance of runtime and space complexity analysis (checking for no of statements executed per second and checking for garbage collection limits).