Using Codeboard and Mantra to Run and Grade SQL Assignments

Liam Tarr
Supervisor: Dr. Ramon Lawrence

An honours thesis submitted in partial fulfillment of the requirements for the degree of
B. Sc. Data Science Honours

Irving K. Barber Faculty of Science
UBC Okanagan
April 2021

Using Codeboard and Mantra to Run and Grade SQL Assignments

Liam Tarr

Abstract

Students who want to work with data need to know fundamental database skills like SQL. Courses for these subjects are often very large. This makes manually grading assignments tedious. This paper explores two methods to solve this problem by automatically grading SQL assignments using Codeboard and Mantra. The first implementation harnesses the power of Python to do this. The second explores directly executing SQL on Codeboard and Mantra. Suggestions are then made on how to improve this second implementation.

Contents

Listings

Chapter 1

Introduction

Knowledge of databases and SQL are in high demand. Everybody who needs to work with data should know SQL. Courses like COSC 304: Introduction to Database Systems teach students these valuable skills.

These courses usually have a large number of participants. This makes manually grading assignments time consuming and tedious.

Some students may find it difficult to get the proper Database Management System working on their computer. Different operating systems and database management systems have different setups and not everything will work the same. This could discourage students and lead to worse outcomes.

It would be nice if we could solve both these problems at once. Is there a better way than manually grading SQL assignments? Can we automatically grade them? These are questions we will explore in this paper.

Chapter 2

Codeboard and Mantra

Codeboard is a "web-based IDE to teach programming in the classroom."[1] Many languages are supported out of the box, such as C, C++, Java, and Python. In general we refer to Codeboard as the front-facing part of the application, such as the IDE, and Mantra is the behind the scenes service that creates Docker containers, and evaluates the code.

2.1 Codeboard

The most essential front-facing feature of Codeboard is the Integrated Development Environment (IDE). There students can compile, run and submit their programming projects for marking. Instructors can create projects, code, and test files in order to mark projects. The instructor can then share a link to the project with their students, and then see and mark their submissions.

2.1.1 Instructors

We will first consider Codeboard from an instructors point of view. Instructors create projects, which they then can share with their students. These projects usually include a main file for running, a test file, and another test file for grading.

Figure 2.1 shows what a project looks like from the view of an instructor. It is worth noting that files marked with an "(h)" are hidden from the student's point of view. This means, for example, that students can not see the tests an instructor has written in order to grade the assignment.

The instructor can see the grades of their students, and check each students progress on the assignments. Figure 2.2 shows an example of what kind of information an instructor can get on a student's submission. They can click on the "Open in IDE" button to see the code that the student has written.

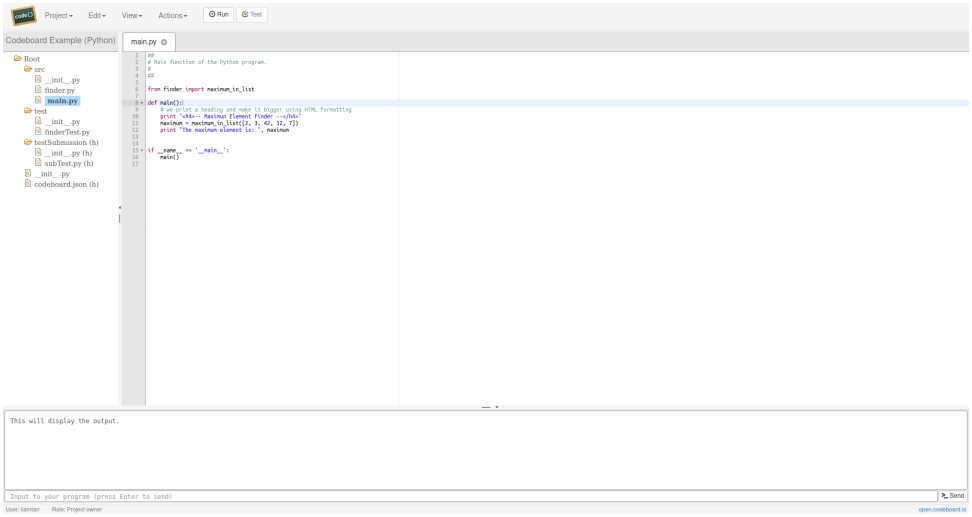

Figure 2.1: Instructors View on Codeboard

examplesqluser (# of Submissions: 1) (Student Number: 11995544)

Created	Result	Tests passed	Tests failed	LTI	
Apr 29, 2021 5:26:39 PM	0.5	2	2	no	Open in IDE

Figure 2.2: Instructor's View of a Submission

2.1.2 Students

The students view of the IDE is much like the instructors view of the IDE. Students can check their code by running tests, and when they feel comfortable with their answer, they can submit their code for marking. The student is given instant feedback on the mark that their submission got.

2.2 Mantra

Mantra could be described as the middleman between Codeboard and Docker. Mantra can create and destroy Docker containers, and command these containers to execute a students code. It is also responsible for getting the containers to run tests and grade assignments.

Chapter 3

SQL in Codeboard Using Python

We can utilize the ability of Mantra to run Python along with unit tests to indirectly run and grade SQL queries.

3.1 Behind the Scenes

Since we are going to use Python, nothing needs to change on the Mantra side. However, we need to change some things in Codeboard.

We need to choose a project structure for our Python SQL projects. This should have files for running. testing, and grading submissions. One way to configure such a project is as below.

```
Root
├── codeboard.json
├── __init__.py
├── Question1.txt
├── Question2.txt
├── Question3.txt
├── src
│   ├── main.py
│   ├── SQLService.py
│   ├── file_input.py
│   └── __init__.py
├── test
│   ├── test.py
│   └── __init__.py
└── testSubmission
    ├── subTest.py
    └── __init__.py
```

3.1.1 Configuration

First, we need to include a `codeboard.json` file.

```
{
  "_comment": "Configuration for this Python-SQL-UnitTest project.",
  "MainFileForRunning": "./Root/src/main.py",
  "DirectoryForSourceFiles": "./Root/src",
  "DirectoryForTestFiles": "./Root/test",
  "DirectoryForTestSubmissionFiles": "./Root/testSubmission"
}
```

Listing 3.1: codeboard.json

These properties point to important locations in our project structure. As configured, the file that will be run by pressing the "run" button is `main.py`. Also, we relegate test files to the `test` folder and grading tests to the `testSubmission` folder.

3.1.2 Helper Files

To make this implementation work, some methods were needed in order to read and process student's code. One such file is `file_input.py`

```
def getUserQuestionContent(questionNumber):
    result = ""
    for line in open("Root/Question" + str(questionNumber) + ".txt"):
        li=line.strip()
        if not li.startswith("#"):
            result = result + line.rstrip()

    return result
```

Listing 3.2: file_input.py

This file creates a method `getUserQuestionContent(questionNumber)` which takes as input the question number of the code that needs to be read. This method iterates through each line of a question text file, and returns only the code that is not marked as a comment with a "#" sign. This is important, as we only want to run the SQL part of a students answer.

Another helpful file is `SQLService.py`. This file contains three function that are essential to the functionality of testing and grading SQL assignments. We will look at each of these functions individually.

The first function to consider is `SQLTable(sql, tbname)`. This function is designed to be helpful in testing `CREATE TABLE` statements.

```
def SQLTable(sql, tbname):
    if(sql.split(' ')[2] != tbname): #Compares the table name argument with the table name from the CREATE TABLE statement
```

```
3         return "Did not find correct table name, please use CREATE
    TABLE (tablename);\nWe are looking for table name '" + tbname +
    "' but we found '" + sql.split(' ')[2] + "'"
4     conn = sqlite3.connect(":memory:")
5     cursor = conn.cursor()
6     result = ""
7     try:
8         cursor.execute(sql)
9         cursor.execute("pragma table_info('" + tbname + "')")
10        result = cursor.fetchall()
11    except:
12        pass
13    return result
```

Listing 3.3: "SQLTable()"

It takes as input `sql`, which is the `CREATE TABLE` statement to test. The argument `tbname` is used to drop the table when we're finished with it. The function executes the `CREATE TABLE` statement, and then returns table info on the column names and types. This information is used to compare answers on tests.

The second function we will consider is `SQLPopulate()`. This function is used to help test `INSERT`, `DELETE`, and `UPDATE` statements.

```
1 def SQLPopulate(setup, inputQuery, outputQuery, tbname):
2     if(setup[0].split(' ')[2] != tbname): #Compares the table name
    argument with the table name from the CREATE TABLE statement
3         return "Did not find correct table name, please use
    CREATE TABLE (tablename);\nWe are looking for table name '" +
    tbname + "' but we found '" + setup[0].split(' ')[2] + "'"
4     conn = sqlite3.connect(":memory:")
5     cursor = conn.cursor()
6     result = ""
7     try:
8         length = len(setup)
9         for i in range(length):
10            cursor.execute(setup[i])
11
12        cursor.execute(inputQuery)
13        cursor.execute(outputQuery)
14        result = cursor.fetchall()
15    except:
16        pass
17    return result
```

Listing 3.4: "SQLPopulate()"

The argument `setup` is an array of strings containing the queries to create the table to be tested on. The argument `inputQuery` is the SQL query we will be testing, which should be an INSERT, DELETE, or UPDATE statement. The `outputQuery` is the SQL query we will use to obtain output. Finally, `tbname` is the name of the table, which is used for comparison and to drop the table.

The function begins by creating the table for which we will be running the query on. The function then executes the `INSERT/DELETE/UPDATE` statement to modify the table. It will then return the output from the output query specified

as an argument. This information is then returned so that it be used to compare answers in tests.

The last helpful function in this file is `SQLSelect()`. This function is designed to aid in testing `SELECT` queries.

```
def SQLSelect(setup, inputQuery, tbname):
    if(setup[0].split(' ')[2] != tbname): #Compares the table name argument with the table name from the CREATE TABLE statement
            return "Did not find correct table name, please use CREATE TABLE (tablename);\nWe are looking for table name '" + tbname + "' but we found '" + setup[0].split(' ')[2] + "'"
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    result = ""
    try:
        length = len(setup)
        for i in range(length):
            cursor.execute(setup[i])

        cursor.execute(inputQuery)
        result = cursor.fetchall()
    except:
        pass
    return result
```

Listing 3.5: "SQLSelect()"

The argument `setup` is used similarly to the previous function in that it creates and modifies the table to be what we want to run queries on. The `inputQuery` is an argument that represents the `SELECT` statement we want to test. The argument `tbname` is used the same way as the previously mentioned functions.

The function starts by executing the setup code to create and modify the table. The `SELECT` statement is then executed and the results returned for use in comparison on tests.

3.2 Writing Tests

An instructor needs to write tests in order to mark SQL assignments. We will see an illustrative example of a test that an instructor could write.

Below is an example of a unit test that an instructor might write.

```
    def test_Create_Table(self):
        q1Table = """CREATE TABLE Recipe (
        id    int     NOT NULL,
      name    VARCHAR(40),
      authorId  int,
      directions  VARCHAR(255),
      PRIMARY KEY (id),
      FOREIGN KEY (authorId) REFERENCES Author(id)
    ON DELETE SET NULL ON UPDATE NO ACTION
```

```
10          )"""
11          self.assertEqual(SQLTable(q1Table, "Recipe"), SQLTable(
    getUserQuestionContent(1), "Recipe"))
```

Listing 3.6: An Example of Tests

The above unit test is to grade an assignment that asks for a student to create a specific table. We can see that the table is specified in the unit test. This table is then compared to the student's created table using the `assertEqual` method and the `SQLTable` function we examined earlier in the paper.

Similar tests can be written for `UPDATE`, `INSERT`, `DELETE`, and `SELECT` statements using the helper functions we defined earlier.

Chapter 4

Adding Native SQL Support to Mantra

4.1 Adding a Language to Codeboard and Mantra

The below will serve as a guide to adding support for a new language to Mantra and Codeboard. We will show how this is done, using `sqlite` as an example.

It would be nice if we could run SQL queries natively in Codeboard. This could have several advantages over running SQL through Python. To do this, we must add SQL support to Mantra.

There are various steps which must be taken in order to add a new language to the Mantra program.

4.1.1 Codeboard

The first step is to create a docker file, which will be used to configure the containers used by Mantra. For this container, we want to be able to run SQL on it. I have chosen to use SQLite since this is a simple implementation of a Database Management System.

```
1 FROM cobo/ubuntu
2
3 RUN \
4   # update the package manager
5   apt-get update && \
6
7   # install sqlite
8   apt-get install -y --force-yes sqlite3 libsqlite3-dev
```

Listing 4.1: sqlite.docker

The next step is to set up the default template for what an SQL project

looks like.

We want the file structure on the Docker container to be as below:

```
Root
├── codeboard.json
└── src
    └── main.sql
```

To do this we have to edit the `dbTemplateProjects.js` file.

```
var addTemplateForSQLite = function() {
  return new Promise(function (resolve, reject) {
    db.TemplateProject
      .find({
        where: {language: 'SQLite'}
      }).then(function(result) {
        if (result === null) {
          var _templateProjectId = -1;

          db.TemplateProject.create({
            language: 'SQLite'
          }).then(function(templatePrj) {
            _templateProjectId = templatePrj.id;

            // the Root folder
            return db.TemplateFile.create({
              filename: 'Root',
              path: '',
              uniqueId: 0,
              parentUId: -1,
              isFolder: true,
              content: '',
              isHidden: false,
              TemplateProjectId: _templateProjectId
            });
          }).then(function(templateFile) {
            // the Src folder
            return db.TemplateFile.create({
              filename: 'src',
              path: 'Root',
              uniqueId: 1,
              parentUId: 0,
              isFolder: true,
              content: '',
              isHidden: true,
              TemplateProjectId: _templateProjectId
            });
          }).then(function(templateFile) {
            var _content = fs.readFileSync('db_templates/SQLite/
    Root/codeboard.json', 'utf8');

            return db.TemplateFile.create({
              filename: 'codeboard.json',
              path: 'Root',
              uniqueId: 2,
              parentUId: 0,
              isFolder: false,
              content: _content,
```

```
              isHidden: true,
              TemplateProjectId: _templateProjectId
            });
          }).then(function(templateFile) {
            var _content = fs.readFileSync('db_templates/SQLite/
    Root/src/main.sql', 'utf8');

            return db.TemplateFile.create({
              filename: 'main.sql',
              path: 'Root/src',
              uniqueId: 3,
              parentUId: 1,
              isFolder: false,
              content: _content,
              isHidden: false,
              TemplateProjectId: _templateProjectId
            });
          }).then(function(templateFile) {
            resolve();
          });
        }
        else {
          // the template project already exists
          resolve();
        }
      });
  });
};
```

Listing 4.2: dbTemplateProjects.js (pt 1)

We must also add another section of code as seen in the listing below:

```
  }).then(function() {
    return addTemplateForSQLite();
  }).then(function () {
```

Listing 4.3: dbTemplateProjects.js (pt 2)

There are certain configuration options that Codeboard and Mantra expect for each supported language. For `sqlite`, we recognize that it is not compiled, and that (as of this moment) testing is not supported. We must let Codeboard and Mantra know this by editing the `codeboard/lib/config/env/all.js` file.

We must first add information about our `sqlite` setup to the `containers` object in the file. We do this by adding a property to it.

```
  sqlite:{
    name: 'SQLite',
    isEnabled: true,
    isDynamic: true,
    isTestable: false
  }
```

Listing 4.4: Adding language configuration (pt 1)

In the same file, we must also add some information to the `templates`

object. This information again reiterates that the language will be enabled, and points to the language information object we previously set up.

```
  'SQLite':{
    language: containers.sqlite,
    isEnabled: true
  }
```

Listing 4.5: Adding language configuration (pt 2)

The last thing we need to do in Codeboard, is allow a user to select the language as a project type. This is done in the file `app/views/partials/projectNew.html`. We simply need to add an `option` tag in a form.

```
<option value="SQLite">SQLite</option>
```

Listing 4.6: Adding SQLite option to New Project Form

4.1.2 Mantra

A change also has to be made to some configuration files on the Mantra side of things. The `mantra/server/config/const.js` file defines some constants that are used throughout Mantra. We must add the name of the newly supported language. In the case of `sqlite`, we add the name as a property to the `LANGUAGE_NAME` object.

```
  LANGUAGE_NAME: {
    C: 'C',
    CPP: 'C++',
    HASKELL: 'Haskell',
    HASKELL_HSPEC: 'Haskell-HSpec',
    JAVA: 'Java',
    JAVA_JUNIT: 'Java-JUnit',
    PYTHON: 'Python',
    PYTHON_UNIT_TEST: 'Python-UnitTest',
    SQLITE: 'SQLite'
  }
```

Listing 4.7: const.js

Next up is to add a "language file". This file is important, as it tells Mantra many of the properties the specific language you want to support has. I will give a brief overview of the important parts of this file.

```
  // name of the language
  name: CONST.LANGUAGE_NAME.SQLITE,

  // file extension of the source files
  filenameExtension: '.sql',

  // is this a static or dynamic language? e.g. Python is dynamic
  isDynamicLanguage: true,

  // does the language come with supports for (unit) tests?
  isLanguageWithTestSupport: false,

```

```
13  // name of the docker image that is used to execute "compile" or
     "run" for this language
14  dockerImage: 'cobo/sqlite',
```

Listing 4.8: sqlite.js (pt 1)

The `name` property should be set to the constant we previously set in `const.js`.

Next up, we need to provide the file extension of the files that will be run. In this example it will be `.sql` files.

In lines 8 and 11 we see lines that should look familiar from previous configurations that we've set up. The property `isDynamicLanguage` should be set to `true` if the language is not compiled. The property `isLanguageWithTestSupport` will give information to Mantra about whether the language should support tests.

The most important part of this file indicates how the files should be run. Below is the configuration for our `sqlite` instances.

```
1  getCommandForRunAction: function (aCodeboardConfig) {
2    var cmd = 'sqlite3 ./Root/main.db < ' + aCodeboardConfig.
     MainFileForRunning;
3    return cmd;
4  },
```

Listing 4.9: sqlite.js (pt 2)

This simple method pipes the input file (`main.sql`) to `sqlite` for it to run.

After this file has been set up, you need to update `languages/index.js` to refer to it. In this case, we need to add the lines below.

```
1  {
2    name: CONST.LANGUAGE_NAME.SQLITE,
3    propertyFile: 'sqlite.js'
4  }
```

Listing 4.10: languages/index.js

4.2 Trying Out SQL Implementation

To start a new `SQLite` project, the project owner must first create a new project with the language set as `SQLite`, as seen in figure 4.1.

The project owner is then brought to the IDE. From there they can setup the `main.sql` file to contain some boilerplate code to instantiate a new table. One way they might do this is seen in figure 4.2

Running the SQL code from figure 4.2, we get the expected output as seen in figure 4.3.

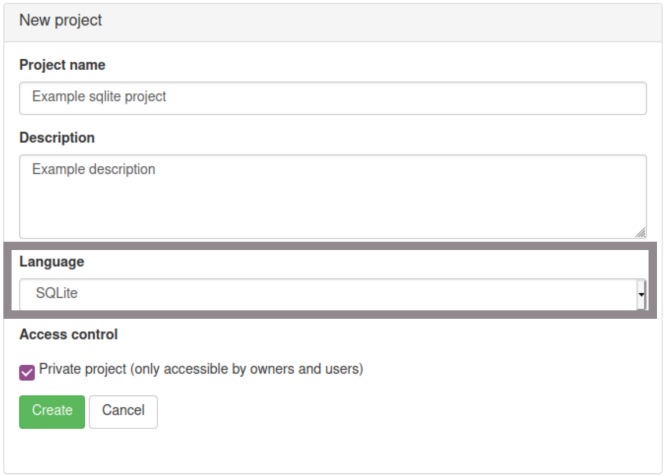

Figure 4.1: New Project View

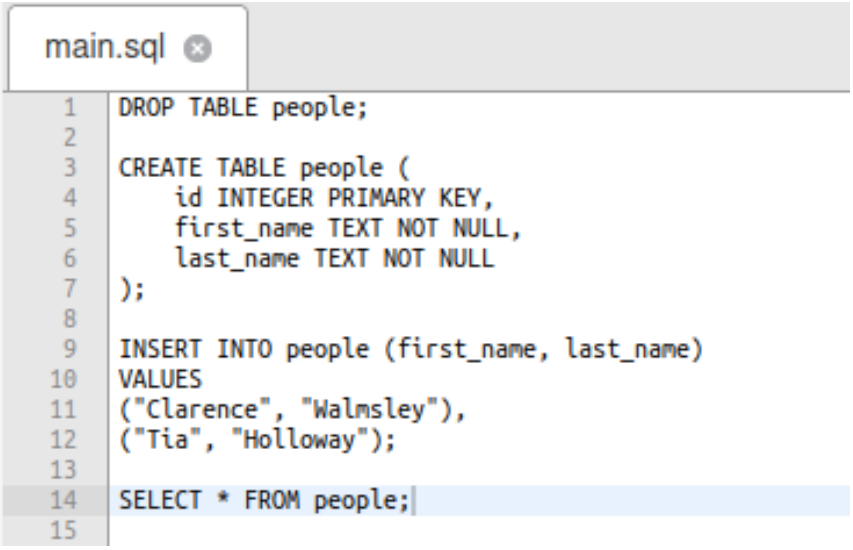

Figure 4.2: Example SQL

4.3 Areas to Improve

There are many ways to improve on this implementation of running SQL directly on Codeboard and Mantra.

This implementation does not support testing and grading assignments. An optimal solution would be able to do both of these things. To be able to do this testing, SQLite might have to be abandoned in favour of other technologies that support unit testing such as T-SQL with the tSQLt framework.

Something we need to be cautious with in our implementation is students getting confused and altering the database on their own project. There's not much of a security issue with this, because each project in an isolated Docker container, but it would still lead to confusion on assignments. One way we could protect students from this is using permissions to limit the amount of damage a student can do to

```
1|Clarence|Walmsley
2|Tia|Holloway
```

Figure 4.3: SQL Output

their project.

Bibliography

[1] *Codeboard - the IDE for the classroom*. 2021. URL: https://codeboard.io/.