W07 Practical Report

Overview

The practical requires that the program takes in 2-3 inputs and have the usage information:

```
Usage: java -cp sqlite-jdbc.jar:. W07Practical <db_file> <action> [input_file |
minimum_rating]
```

The program shall allow 6 possible actions:

1. `create` will create the SQLite database if it does not exist with the table restaurant. It will taken in the `input_file` which will be the database that shall be converted to the database.
2. `query1` will run a query and that prints out the "city, name, rating and cuisine_style" where the rating is equal to 5, the city the restaurant is in is either "Amsterdam" or "Edinburgh" and if the restaurant has the cuisine_style of "European" and then it will print out the output to the user.
3. `query2` takes in the `minimum_rating` argument and runs a query which prints out the number of restaurants with a rating greater than or equal to the minimum rating given by argument. Then returns output to user
4. `query3` takes in the `minimum_rating` argument and runs a query which prints out the city and number of restaurants with a rating greater than or equal to the minimum rating given by argument in that city. Then returns output to user.
5. `query4` runs a query which returns the average rating for restaurants for each city. Then outputs this information to the user.
6. `query5` for each city, prints out the "city, name, rating" for restaurants which have the lowest rating in that given city. It assumed that there can be multiple restaurants with the same minimum rating.

Problem Decomposition

- verify input is correct
- create SQLite database
- add data from `.csv` to to rows in that database.
- convert the different `query*` actions to SQL queries.
- Output the result of the action.

GROUP BY

Is used with aggregate functions and groups the result-set by one or more columns. This was implemented in the initial practical as it was the most effective way to implement solution giving the results already grouped meaning less code needed to be written in java (e.g. using java to do grouping of minimums) or 2 queries (or sub queries) in SQL to find solution.

VIEWS

Is a virtual table which is based on result set created from a SQL query. This could allow for the implementation as queries we want the database as databases and allow for simple

```
SELECT * FROM view
```

queries to get the desired result where `view` is the table which represents the desired result set. This simplifies the query command and allows others who are using the program to still get the results for certain queries without having to run them in the program. Also it means when a user enters for a specific query to be obtained that only a command which gets the query name from the database needs to be ran. e.g.

```
SELECT * FROM query1
```

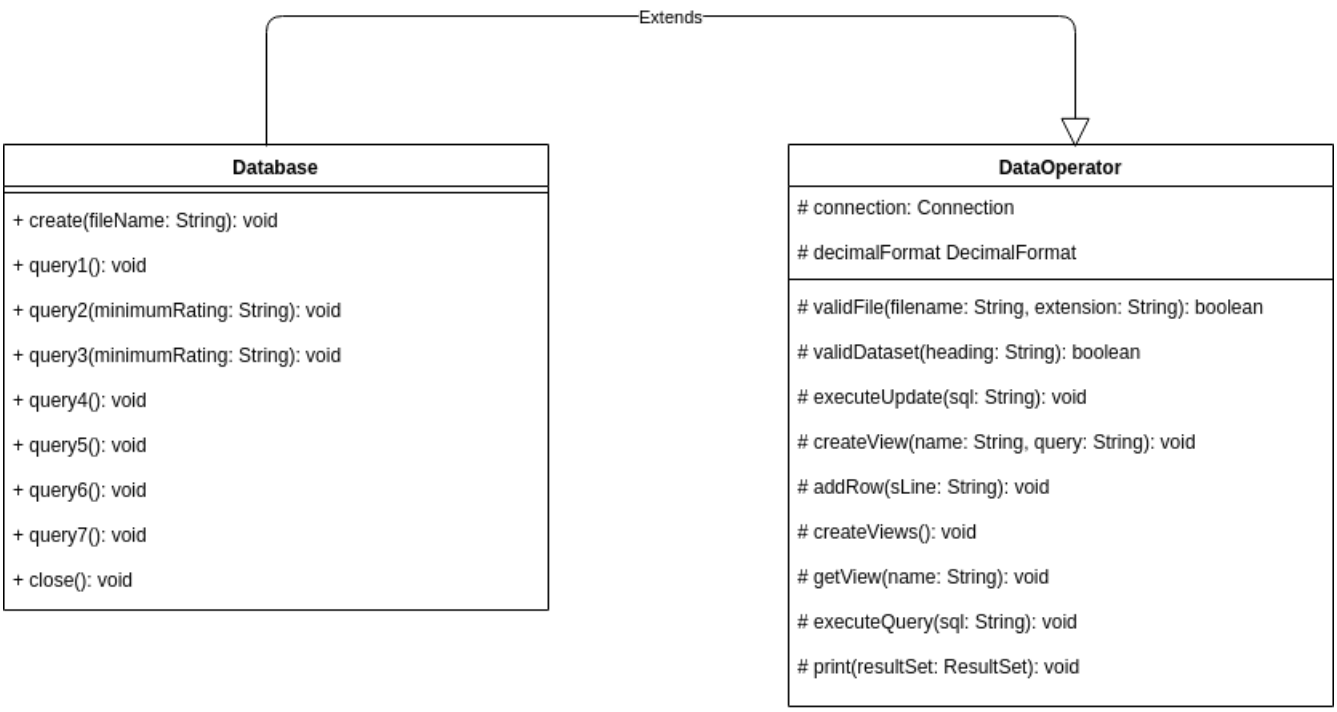
STDEV

This is a statistical aggregate function which find the deviation from mean in a column. Here additional functionally will be added to the program to find the standard deviation of rating for the total dataset find the standard deviation of rating for each city. This was added to the initial program as the `stacscheck` require verification of whether for queries which did not exist were attempted to be ran. So it was assumed additional queries implemented would be acceptable.

Extensions Implemented

- Use of GROUP BY
- Use of VIEWS
- Additional Statistical functionality (e.g. standard deviation of ratings in a city)

Design



The interactions of the program with the SQLite database has been abstracted into a class. Then the actions which the program is required to fulfilled were made into a methods of this class. It was found that each of the queries that were required to be ran by the program could be written almost entirely in SQL and then formatted in java, so this supported the implementation of them as methods as the amount of code would be minimal as they would be . This allows for the main methods to be clean and show the general functionality that is being carried out given the input of the program.

It was found when creating the program if only implemented in the one Database class that the lines of code in the file would exceed 450. So to make the file smaller and easier to read it was decided to add another layer of abstraction which would contain all the non-public methods of that would be required to be used by the Database query methods into another class as protected methods. Then for theses methods to have access to all these methods it was decided that they would then inherit from the new layer of abstraction and have access to all these files and this would mean these lines of code could then be distributed between these two files. Making the code and flow of logic easier to read.

It was decided to make the connection a protected variable of the `DataOperator` class. This design decision was chosen to allow for the use of the same connection to the database used across any of the `query*` methods without having to create a connection to the database in every single method. So instead the connection is opened in the constructor of the `Database` class and closed with the `close` method. As it was choose to keep public methods in Database class.

It was decided to implement: `query1`, `query4`, `query6`, `query7` as views in the SQL database. this allowed for the generalisation of behaviours in to functions e.g. `getViews()`. These queries were the optimal to implement as views since they did not require input from the user so it would be easy to make a view which represents the data they contain. A view was not made for `query5` as it was implemented as two as the query was too slow (discussed further in evaluation), instead it was implemented in two queries with additional java code.

Through the program when a string or concatenation gets too long involving the concatenation of string, concatenation is taken onto a new line to maintain the readability. e.g.

```
    String query1 = "SELECT city, name, rating, cuisine "
    + "FROM restaurant WHERE (city = 'Edinburgh' or city = 'Amsterdam') and cuisine "
    + "LIKE '%European%'and rating = 5";
```

It was chosen every method and the constructor of the database class would throw `SQLException`. This was decided to minimise the try catches throughout the program as every method in the database class then would have to have a try and catch for `SQLException`. So to prevent this the exception be caught in the main method.

Since all the queries have to print results to the terminal it was decided to generalise this behaviour into a method(`print`). This method prints all the results in a result set and separates the columns by `,`. This means that printing commands do not need to be separately created for each query.

Testing

`stacscheck`

```
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical $ stacscheck /cs/studres/CS1003/Practicals/W07/Tests
Testing CS1003 Week 7 Practical
- Looking for submission in a directory called 'src': found in current directory
* BUILD TEST - basic/build : pass
* COMPARISON TEST - basic/Test01_no_arguments/progRun-expected.out : pass
* COMPARISON TEST - basic/Test02_small_create/progRun-expected.out : pass
* COMPARISON TEST - basic/Test03_small_query1/progRun-expected.out : pass
* COMPARISON TEST - basic/Test04_small_query2/rating_30/progRun-expected.out : pass
* COMPARISON TEST - basic/Test04_small_query2/rating_35/progRun-expected.out : pass
* COMPARISON TEST - basic/Test04_small_query2/rating_40/progRun-expected.out : pass
* COMPARISON TEST - basic/Test04_small_query2/rating_50/progRun-expected.out : pass
* COMPARISON TEST - basic/Test05_small_query3/rating_30/progRun-expected.out : pass
* COMPARISON TEST - basic/Test05_small_query3/rating_35/progRun-expected.out : pass
* COMPARISON TEST - basic/Test05_small_query3/rating_40/progRun-expected.out : pass
* COMPARISON TEST - basic/Test05_small_query3/rating_50/progRun-expected.out : pass
* COMPARISON TEST - basic/Test06_small_query4/progRun-expected.out : pass
* COMPARISON TEST - basic/Test07_small_query5/progRun-expected.out : pass
* COMPARISON TEST - basic/Test08_large_create/progRun-expected.out : pass
* COMPARISON TEST - basic/Test09_large_query1/progRun-expected.out : pass
* COMPARISON TEST - basic/Test10_large_query2/rating_30/progRun-expected.out : pass
* COMPARISON TEST - basic/Test10_large_query2/rating_35/progRun-expected.out : pass
* COMPARISON TEST - basic/Test10_large_query2/rating_40/progRun-expected.out : pass
* COMPARISON TEST - basic/Test10_large_query2/rating_50/progRun-expected.out : pass
* COMPARISON TEST - basic/Test11_large_query3/rating_30/progRun-expected.out : pass
* COMPARISON TEST - basic/Test11_large_query3/rating_35/progRun-expected.out : pass
* COMPARISON TEST - basic/Test11_large_query3/rating_40/progRun-expected.out : pass
* COMPARISON TEST - basic/Test11_large_query3/rating_50/progRun-expected.out : pass
* COMPARISON TEST - basic/Test12_large_query4/progRun-expected.out : pass
* COMPARISON TEST - basic/Test13_large_query5/progRun-expected.out : pass
* INFO - basic/TestQ_CheckStyle/infoCheckStyle : pass
--- submission output ---
Starting audit...
Audit done.
---

27 out of 27 tests passed
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical $
```

Example test for the functionality of the program. The `stackscheck` shows that each method is able to run correctly and execute and give the correct output but `stacschecks` does not test for incorrect input by the user. So following tests shall test the program is capable to deal with incorrect input.

Test Case 1: Input file is not a database file `.db`

Here the program is able to deal with when the user does not request a database file.

Input

```
java -cp sqlite-jdbc.jar:. W07Practical test create clean_small.csv
```

Expected Output

Program warns the user the input file is not a `.db` file but creates the database anyway in expected file name.

Actual Output

```
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $ javac *.java
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $ java -cp sqlite-jdbc.jar:. W07Practical test create clean_small.csv
Warning Extension is not .db
OK
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $
```

Test Case 2: CSV Errors

Here a bunch of tests which can occur due to Errors in CSV are tested.

1. CSV file with incorrect extension

```
java -cp sqlite-jdbc.jar:. W07Practical test.db clean_small
```

2. When the first line of the CSV doesnt match the CSV column heading of the expected file.

```
touch empty.csv
java -cp sqlite-jdbc.jar:. W07Practical test.db empty.csv
```

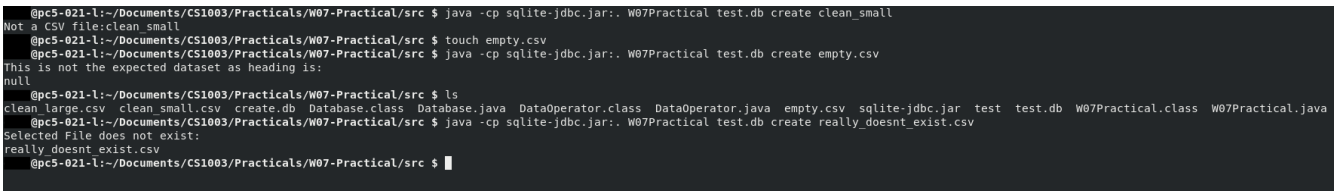
3. Program ran on CSV file which doesn't exist

```
ls
java -cp sqlite-jdbc.jar:. W07Practical test.db really_doesnt_exist.csv
```

Expected Output

1. OK is not printed as nothing is added to database as CSV is not being inputted as data file and an error informing User that input file is not a CSV is prompted.
2. User is told this is not expected dataset and no data is added to table
3. User is informed CSV doesn't exist.

Actual output



Test Case 3

Here it is tested when the minimum rating argument is given in the wrong format

1. When the minimum rating is given as a text instead of number

```
java -cp sqlite-jdbc.jar:. W07Practical test.db query2 IMPORT_TENSORFLOW
```

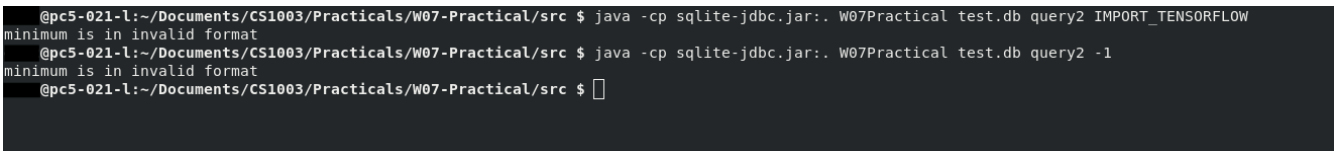
2. When minimum rating is less than zero

```
java -cp sqlite-jdbc.jar:. W07Practical test.db query2 -1
```

Expected Output

It is expected that in both cases the error prompt that the minimum rating is in the wrong format is given. As program only accepts number between 0 and 5 inclusively.

Actual Output



Test Case 4

Here a full example of program execution is shown as well as the showing the `STDEV` queries correctly execute.

Expected Output

- `*` represents a result

```
OK
Standard Deviation of Rating
*
City, Standard Devoation of Rating
*, *
...
```

Actual Output

```
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $ javac *.java
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $ java -cp sqlite-jdbc.jar:. W07Practical test.db create clean_large.csv
OK
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $ java -cp sqlite-jdbc.jar:. W07Practical test.db query6
Standard Deviation of Rating
0.7
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $ java -cp sqlite-jdbc.jar:. W07Practical test.db query7
City, Standard Deviation of Rating
Amsterdam, 0.6
Athens, 0.6
Barcelona, 0.7
Berlin, 0.6
Bratislava, 0.8
Brussels, 0.7
Budapest, 0.7
Copenhagen, 0.7
Dublin, 0.6
Edinburgh, 0.7
Geneva, 0.6
Hamburg, 0.7
Helsinki, 0.7
Krakow, 0.6
Lisbon, 0.6
Ljubljana, 0.6
London, 0.7
Luxembourg, 0.6
Lyon, 0.7
Madrid, 0.7
Milan, 0.7
Munich, 0.6
Oporto, 0.6
Oslo, 0.6
Paris, 0.7
Prague, 0.7
Rome, 0.4
Stockholm, 0.7
Vienna, 0.6
Warsaw, 0.7
Zurich, 0.6
    @pc5-021-l:~/Documents/CS1003/Practicals/W07-Practical/src $
```

Extra

A small bash file was written to show execution of all the actions of the program to run it

```
bash test.sh
```

The code contained in the bash file

```
javac *.java
java -cp sqlite-jdbc.jar:. W07Practical test.db create clean_large.csv
java -cp sqlite-jdbc.jar:. W07Practical test.db query1
java -cp sqlite-jdbc.jar:. W07Practical test.db query2 4
java -cp sqlite-jdbc.jar:. W07Practical test.db query3 4
java -cp sqlite-jdbc.jar:. W07Practical test.db query4
java -cp sqlite-jdbc.jar:. W07Practical test.db query5
java -cp sqlite-jdbc.jar:. W07Practical test.db query6
java -cp sqlite-jdbc.jar:. W07Practical test.db query7
```

Evaluation

The specification required that a program which was able to parse a CSV dataset obtain from kaggle (restaurant dataset) and make it into a database extracting only (`number`, `name`, `city`, `rating`, `cuisine_style`) as the features which would be used as the columns of the table in the database. The specification also required the program be able to execute 5 queries which obtained data from the `CSV` file in the database. The `stackscheck` and the `test.sh` show the the programs queries all execute and provide the correct output. Thus showing the program matches the program as the required specification as expected output has been produced.

Through the further testing done it was also shown the program was able to correctly deal with incorrect input of `.csv` file and `.db` file. It was also shown that the program could deal with when the user enters incorrectly formatted minimum rating.

Conclusion

In this practical a program which was capable of interacting with a SQL database was produced. It was capable of creating tables and inserting values into rows of the table and running queries and operation upon data in the table. It was capable of dealing with possible errors in input which could occur.

Difficulties

Original the SQL query for `query5`

```
SELECT city as currentCity, name, rating FROM restaurant WHERE rating = (SELECT MIN(rating)
FROM restaurant WHERE city=currentCity)
```

was used but it was found when running this query on the large dataset the query was very slow. This query would also timeout on the `stacscheck`. So increase the speed at which the program executes the `query5` action it was separated into two queries and ran in java.

```
SELECT city, MIN(rating) as minimum FROM restaurant GROUP BY city
```

Then this result set was looped over for all the cities in the result set

```
SELECT city, name, rating FROM restaurant WHERE rating = currentMinimum and city =
currentCity
```

Given more time

Given more time additional methods could have been created to further verify if the input data was in the correct format. e.g. inconsistencies in elements on each row in `CSV`. Also a `stacscheck` could have been written for this program to check the output of the `STDEV` extensions and also the output of when incorrect data is inputted into the program.