CITS1402 Week 2 Lab

Gordon Royle

2023 Semester Two

This lab is an *individual assessment* and therefore anything you submit should be *your own work*. You may discuss the lab in general terms with other students, but you should not look at, share, or show anyone any actual code.

This lab should be completed by 5pm Sunday at the end of Teaching Week 2 (that is, 5pm Sunday 30 July). This is the *soft deadline* for this lab — you can submit files after this, but for a possible late penalty. Solutions will be released at 5pm on Tuesday (two days later) and this is the *hard deadline* after which no further solutions will be accepted from any student. (If you are a UAAP student, you should already have received information about how this hard deadline interacts with any extensions to which you may normally be entitled.)

Three of the questions are labelled with a yellow box such as Submission: `A1.sql`. This indicates that you should submit a file called `A1.sql` containing the answer to that particular question. In this week, the first labelled question is Question 4, but you should still put your answer into file `A1.sql`.

Just to reiterate, the three files that you should submit are called `A1.sql`, `A2.sql` and `A3.sql` regardless of which actual questions are labelled.

You should submit the files at

`https://secure.csse.uwa.edu.au/run/cssubmit`

These files should be *plain text files* with the *exact file name* as given, and should contain a single SQL query, that starts with `SELECT` and terminates with a semicolon. The file should contain *nothing else* at all. No names, no student numbers, no dot commands or anything else, but just the query on its own.

By default, Windows *silently hides* the filename extension (the letters after the dot), so that if you just use the usual File Explorer, it will only show you `A1`, `A2`, `A3` rather than `A1.sql`, `A2.sql`, `A3.sql`. Use the `ls` or `dir` commands in Powershell to get the full names.

You may test your command in the following manner on either Windows or Mac.

- Start up SQLite3 from the Powershell / Terminal prompt

  ```
  sqlite3
  ```

- At the `sqlite3` prompt, type in

  ```
  .open AFLResult.db
  .read A1.sql
  ```

Then check (by hand / eye) that the output meets the specifications.

Note that your submissions will mostly be checked by computer script and therefore the output has to be *exactly* as specified in the question.

Make sure that you test the *actual submission file* just before you submit it.

Learning Aims

This lab is concerned with

1. Creating and populating an SQLite table
2. Examining the table and its entries
3. Basic use of the `SELECT` statement

We will be using data about Australian Football League (AFL) matches from 2011–2022.

Required Setup

The database file for this lab is `AFLResult.db` which can be downloaded from the LMS.

Types of command

In the labs, I often use the phrase "command" meaning something that you type into the computer in order to get a response or accomplish something. To avoid frustration it is important to carefully distinguish the different types of command.

1. Terminal command

   A *terminal command* is something you type at the Terminal (Mac or Linux) or Powershell (Win) prompt, and that is interpreted by the operating system.

   Terminal commands are used to navigate the file system, create directories / folders, move files etc.

2. `sqlite3` dot command (or just dot command)

   A *dot command* is a command that you type after you have started `sqlite3` and it is displaying the prompt

   ```
   sqlite>
   ```

   and waiting for input. Dot commands start with a dot, and *do not* terminate with a semi-colon.

   Dot commands are specific commands that are mostly concerned with controlling the interface between the user and `sqlite3`. These commands do not work in other implementations of SQL (MySQL, PostgreSQL, etc.) or even in other interfaces to SQLite (SQLiteStudio). You will need to learn just enough dot commands to work effectively with `sqlite3` but they are not part of the assessable material in this unit.

3. SQL command

   A *SQL command* is an actual *database query* that you type at the `sqlite` prompt. SQLite commands *always terminate* with a semi-colon, and return a table as output, which (by default) is printed on the terminal window.

   Most of the core SQLite commands behave according to the SQL standard, although there are some variations. Most of these variations are minor, and we will carefully highlight the few areas where the differences are serious enough to lead to potential confusion.

Creating and Populating

Unlike other implementations of SQL, an SQLite database is stored in a single file on the file system. This file can be transferred from one machine to another, even across operating systems.

Here are two different ways to start working with a pre-existing SQLite database — the end result is the same, so you can do either one.

1. Start SQLite with an existing database file

   Copy `AFLResult.db` to your SQLite directory and at the terminal prompt, issue the command `sqlite3 AFLResult.db` to simultaneously start SQLite and connect the database.

2. Start SQLite and connect to an existing database file

   Use the terminal command `sqlite3` to start SQLite, and then (at the SQLite prompt), use SQLite command `.open AFLResult.db` to connect the database.

After these steps you should have a database with a single table `AFLResult`, which you can verify with the `.tables` dot command.

Examining the Table

Warning: SQLite has recently changed the way it handles string literals. Previously, you could use either *single quotes* or *double quotes* to delimit a string. So the system would treat `'West Coast'` and `"West Coast"` as being the same thing. Depending on exactly what version of SQLite you are using, the second way (using double quotes) may no longer work. So to avoid unexpected failures, *always use single quotes* to delimit a string.

1. The word *schema* refers to the *structure* of the table (not its contents). In SQLite this is viewed using a dot command

   ```
   .schema AFLResult
   ```

   This output from this command is code that was originally used to *create the table*. It shows the *names* of the columns and the *data type* to be used to store the values int that column. For example, the column `homeScore` is a variable that will be used to store an `INTEGER` value, while `homeTeam` is a variable that will be used to store `TEXT` data.

   This section just contains worked examples of queries — you should run them yourself, and figure out how they work.

   Alternatively, you can just go straight to the next section which contains the *questions* for this week.

   > *I am writing the queries using this multi-line convention that displays the structure of the query. SQLite does not care, and will simply read everything from `SELECT` to the semicolon, and then treat that as the entire query. So you can write the whole query out on a single line if you prefer.*

2. Look at the entire contents of the table

```
SELECT *
FROM AFLResult;
```

   The * is a special character meaning "everything" or "all the columns".

3. Look at a few rows from the table

```
SELECT *
FROM AFLResult
LIMIT 10;
```

   The LIMIT statement simply restricts the number of rows of output. It is mostly useful for interactive sqlite3 sessions.

4. Look at *particular rows* from the table using *boolean expressions*

   This returns all the matches where West Coast is the home team.

```
SELECT *
FROM AFLResult
WHERE homeTeam=='West Coast';
```

   This returns all the high-scoring matches

```
SELECT *
FROM AFLResult
WHERE homeScore + awayScore > 150;
```

5. Conditions can be combined using the logical operators AND, OR and NOT.

   This returns all the matches where either Fremantle or West Coast was the home team.

```
SELECT *
FROM AFLResult
WHERE homeTeam == 'West Coast'
    OR homeTeam == 'Fremantle';
```

   The OR statement must combine two *boolean* values; this query is accepted by SQLite, but does not do what you want!

```
SELECT *
FROM AFLResult
WHERE homeTeam == 'West Coast' OR 'Fremantle';
```

6. A query can create *new columns* by combining old ones

   To see this, first use the dot command `.headers on` so that SQLite lists the column names.

```
SELECT homeTeam, awayTeam, homeScore + awayScore
FROM AFLResult
LIMIT 10;
```

This table has three columns, two of which (homeTeam and awayTeam) are just copied over, while the third is a *calculated column*, whose value is given by some calculation or computation.

Notice that the name of the new column is homeScore + awayScore which is not a very useful name, but we can change this using the AS keyword.

```
SELECT homeTeam, awayTeam, homeScore + awayScore AS totalScore
FROM AFLResult
LIMIT 10;
```

Now the new column is called totalScore — when we are just using SQLite interactively, it seems that giving names to the new columns is just for decoration, but later we will see ways in which calculated tables can be used as input in more complex queries.

7. The statement ORDER BY <expr> can be used to *sort the output* according to the value of the expression <expr>. For example

```
SELECT *
FROM AFLResult
ORDER BY homeScore;
```

will list all the games in *increasing order* of the home score. If you want to order them by *decreasing* order, then just add DESC (for "descending") at the end.

```
SELECT *
FROM AFLResult
ORDER BY homeScore DESC;
```

Questions

In all of these questions, a *single SQL query* means a query starting with SELECT and ending with a semicolon.

You may need to refer to Section 2 of https://www.sqlite.org/lang_expr.html to look up the syntax for arithmetic and boolean operators, and Section 1 of https://www.sqlite.org/lang_corefunc.html to look up syntax for arithmetic functions.

QUESTION 1. Write a SQL query that will list the names of the home teams that have scored *more than* 140 points. (If a team has scored more than 140 points on multiple occasions, then their name should appear once for each occasion.)

```
SELECT homeTeam
FROM AFLResult
WHERE homeScore > 140;
```

QUESTION 2. Write a SQL query that will list the names of the home teams that have scored *at least* 140 points.

```
SELECT homeTeam
FROM AFLResult
WHERE homeScore >= 140;
```

QUESTION 3. Repeat the previous query, but using `SELECT DISTINCT` instead of `SELECT`. What happens?

```
SELECT DISTINCT homeTeam
FROM AFLResult
WHERE homeScore >= 140;
```

This removes *repeated rows*, so that every row is different from each other.

QUESTION 4. Submission: `A1.sql` A *home win* is a match that is won by the team playing at home. Write a single SQL query to list the match results (just use `SELECT *` to select every column) of all the home wins in the database.

```
SELECT *
FROM AFLResult
WHERE homeScore > awayScore;
```

SELF-CHECK HINT: The last row of output should be

```
24-Mar-11|Carlton|Richmond|MCG|104|84|N
```

If you do not get this then carefully check your query.

QUESTION 5. Write a single SQL query to list the match details (use `SELECT *`) for every match that Fremantle has won when the game was played at the Melbourne Cricket Ground (MCG).

```
SELECT *
FROM AFLResult
WHERE venue == 'MCG' AND awayTeam == 'Fremantle' AND awayScore > homeScore;
```

Question 6. Optus Stadium is the home ground of both Fremantle and West Coast. Write an SQL query to find out whether any matches have been played there that *did not* involve either Fremantle or West Coast.

```
SELECT *
FROM AFLResult
WHERE venue == 'Optus Stadium'
  AND homeTeam != 'West Coast'
  AND homeTeam != 'Fremantle'
  AND awayTeam != 'West Coast'
  AND awayTeam != 'Fremantle';
```

Question 7. A "Western Derby" is any match between Fremantle and West Coast. Write a SQL command that lists the match details (use `SELECT *`) for all of the Western derbies in the table.

```
SELECT *
FROM AFLResult
WHERE (homeTeam == 'Fremantle' AND awayTeam == 'West Coast')
OR (homeTeam == 'West Coast' AND awayTeam == 'Fremantle');
```

Question 8. Write a SQL query to list the match details (just use `SELECT *`) for the matches that have been decided by *exactly one point*. In other words, the winning score is just one point more than the losing score.

Self-check Hint: The first row of your output should be

```
17-Sep-22|Sydney|Collingwood|SCG|95|94|Y
```

```
SELECT *
FROM AFLResult
WHERE (homeScore == awayScore + 1)
     OR (awayScore == homeScore + 1);
```

QUESTION 9. Submission: `A2.sql` Write a SQL command that will output a table with all same columns as `AFLResult`, but one additional column called `margin` which is the difference between the two scores. For example, if one team scores 100 and the other team scores 88, then the margin is 12.

The winning margin is always positive, so you may find it convenient to use the function `ABS` from the list of SQLite functions.

`https://www.sqlite.org/lang_corefunc.html`

SELF-CHECK HINT: The first few rows of this table should be:

```
24-Sep-22|Geelong|Sydney|MCG|133|52|Y|81
17-Sep-22|Sydney|Collingwood|SCG|95|94|Y|1
16-Sep-22|Geelong|Brisbane|MCG|120|49|Y|71
10-Sep-22|Collingwood|Fremantle|MCG|79|59|Y|20
...
```

```
SELECT *, ABS(homeScore - awayScore) AS margin
FROM AFLResult
LIMIT 5;
```

QUESTION 10. There are two AFL teams based in Western Australia, namely Fremantle and West Coast, and there are two AFL teams based in South Australia, namely Adelaide Crows and Port Adelaide. Write a single SQL query to list the match details of every match between a South Australian team and a West Australian team.

(The query is somewhat lengthy because you have to consider multiple cases — you may find it easier to start with the simpler situation where the home team is from Western Australia and the away team is from South Australia.)

```
SELECT *
FROM AFLResult
WHERE  ((homeTeam == 'Fremantle' OR homeTeam == 'West Coast')
        AND (awayTeam == 'Port Adelaide' OR awayTeam == 'Adelaide Crows'))
      OR ((awayTeam == 'Fremantle' OR awayTeam == 'West Coast')
        AND (homeTeam == 'Port Adelaide' OR homeTeam == 'Adelaide Crows'));
```

QUESTION 11. Submission: `A3.sql` Write a SQL query to list the match details of every match where West Coast *did not lose* (i.e., West Coast was the winner or the match was a draw.). You will need to use combinations of the boolean operators `AND` and `OR` and use brackets to ensure that components of the boolean expression are evaluated in the correct order.

```sql
SELECT *
FROM AFLResult
WHERE ((homeTeam == 'West Coast' AND homeScore >= awayScore)
   OR (awayTeam == 'West Coast' AND awayScore >= homeScore));
```

QUESTION 12. Write a single SQL query that lists the match results (i.e., use `SELECT *` for every game where *more than* 250 points (total) were scored, listing them in *decreasing order* of total number of points scored. Your output table should not have any extra columns.

```sql
SELECT *
FROM AFLResult
WHERE homeScore + awayScore > 250
ORDER BY homeScore + awayScore DESC;
```
