CITS1402 Week 6 Lab

Gordon Royle

2023 Semester Two

This is an *individual* assessment, and so the query that you submit should be devised by you alone, typed into the submission file by you alone, tested by you alone, and then submitted entirely by yourself. You may discuss the assignment with the lab facilitators and your fellow students in *general terms only*. If you are unsure of the line between legitimate discussion and forbidden collaboration, then you can apply the following simple test: If the discussion is purely verbal and mentions no SQLite code, then it is probably legitimate. On the other hand, if you describe *actual SQLite code* in your discussion, or you *show someone* your screen, or *email someone* your files, then it is not legitimate. So you can say "I think you need to figure out which tables have the information, and then join them", but nothing more detailed. If someone asks you to *look* at their code, or to *show* them your code, then you should politely refuse.

You should submit your solutions to the selected lab questions at `secure.csse.uwa.edu.au/run/cssubmit`. The selected questions are highlighted by a `bright yellow box` containing the word `cssubmit` and then a filename.

For each of the highlighted questions, you should save your SQL query into the specified filename and then upload the resulting file (saved as plain text) to `cssubmit`. The validation system will prevent you from submitting files with incorrect names.

Make sure that you *test* your queries by

- Typing `cat A8.sql` at the Powershell/Terminal prompt to view the contents of the file, making sure that there are no formatting commands or strange characters.
- Using `sqlite3` to open the sample database, and then `.read A8.sql` at the `sqlite3` prompt, to see the output that I (my marking scripts) will see. Or use the combined command

  ```
  cat A8.sql | sqlite3 AustOpen.db
  ```

  to open `sqlite3`, connect the `AustOpen.db` database and pipe in the command contained in `A8.sql`
- Double-check the specifications to ensure that the output has the right columns, in the correct order (column order matters), and that the output rows are correct and complete.
- Double check that you have actually *submitted* the files to `cssubmit` and not just *validated* them.

Of course, replace `A8.sql` by whichever file you are testing and `AustOpen.db` by whatever database is relevant (use the filename that is *in the bright yellow box*, no matter where the question occurs in the list of lab questions).

Learning Aims

This lab is primarily concerned with *aggregate* and *summary* functions supplied by SQLite, in particular `MIN`, `MAX`, `COUNT`, `SUM` and `AVG`. There are a few others that we will not use, the full list is at

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

The aggregate functions are used by first grouping rows into groups using `GROUP BY`, summarising the groups using the aggregate functions and (optionally) filtering the summary rows using`HAVING`.

We will use two databases we have previously used, namely `classicmodels.db` and `imdb_top_250.db` and a new database `AustOpen.db`.

For this lab you may find it easier to write (and hence read) your queries if you give the tables that you use short aliases. For example, to count the number of employees located in each office of Classic Models, we could write:

```
SELECT O.city, COUNT(*) as numEmployees
FROM offices O JOIN employees USING (officeCode)
GROUP BY O.officeCode, O.city;
```

The `FROM offices O` statement assigns the table alias `O` to the table `offices` (we can also use `FROM offices AS O`). Then the name `O` is an alias for the table `offices` that can used throughout the query.

At the moment, table aliases are just a convenience, but when we are dealing with *two distinct copies* of the same table, we will need table aliases to distinguish them.

Questions

1. Using the IMDB Top-250, write a single SQL query that lists, for each *year* represented in the database (this is the column `premiered`) the *number of movies* from that year in the IMDB Top-250.

   ```
   SELECT premiered, COUNT(*)
   FROM titles
   GROUP BY premiered;
   ```

   SELF-CHECK HINT There are six movies from 1957 that are still in the Top-250.

2. There are only 2 movies released in 2005 that are still in the Top-250, whereas most other (recent) years have many more. Alter the query in the previous question so that it only outputs the data for years that have *at least* 5 movies still in the Top-250.

   ```
   SELECT premiered, COUNT(*)
   FROM titles
   GROUP BY premiered
   HAVING COUNT(*) >=5;
   ```

3. Repeat the previous question, but now list the output rows in decreasing order of number of Top-250 movies and name the two columns as `ReleaseYear` and `NumMovies`. The year 1995 was particularly good, with 8 Top-250 movies released in that year, more than any other year, and so (in SQLite Studio or in sqlite3 with headers turned on), the first rows of the output should be:

   ```
   ReleaseYear    NumMovies
   1995           8
   ```

```sql
SELECT premiered AS ReleaseYear, COUNT(*) AS NumMovies
FROM titles
GROUP BY premiered
HAVING NumMovies >=5
ORDER BY NumMovies DESC;
```

4. Using Classic Models, write a single SQL query that lists, for each city, the *name* of the city, and the *number of employees* based in that city.

   SELF-CHECK HINT There are four employees in Sydney.

```sql
SELECT city, COUNT(*)
FROM offices JOIN employees
USING (officeCode)
GROUP BY city;
```

5. Submission: `A8.sql` Using `AustOpen.db`, write a single SQL query that for each year in the database lists the *year* and the *number of matches* won by Australian players in the Women's Singles for that year. (Remember that the data for Women's Singles is in the table `WTAResult`.)

```sql
SELECT yr, COUNT(*)
FROM WTAResult
WHERE winnerCountry = 'AUS'
GROUP BY yr;
```

   SELF-CHECK HINT 2022 was a bumper year, because Australian women won a total of 10 matches.

6. Using `AustOpen.db`, write a single SQL query lists the *year*, *player name* and *number of matches won* for each of the Australian players in the Women's Singles. (You need not consider players who did not win any matches — this substantially simplifies the query because you only need to consider names in the `winnerName` column.)

```sql
SELECT yr, winnerName, COUNT(*)
FROM WTAResult
WHERE winnerCountry = 'AUS'
GROUP BY yr, winnerName;
```

   SELF-CHECK HINT In 2022, Ashleigh Barty was the Women's Champion, which she achieved by winning the maximum of 7 matdes, so make sure that one of your rows is (2022, Ashleigh Barty, 7).

7. Using Classic Models, write a single SQL query that lists, for each product line, the *product line* and the *price paid* by Classic Models for the most expensive product in that product line. Recall that the price paid by Classic Models is in the `products.buyPrice` column. (Note that you do not have the list the actual product, just its price.)

```sql
SELECT productLine, MAX(buyprice)
FROM products
GROUP BY productLine;
```

   SELF-CHECK HINT The most expensive product in the `Ships` line costs $82.34, so one of the rows of the output table should be:

   `Ships 82.34`

8. Submission: `A9.sql` Using Classic Models, write a single SQL query that lists, for each order, its *order number*, the *number of line-items* in the order, and the *total cost* of the order, rounded to

two decimal places (look up and use the `ROUND` function for this). The output table should have three columns, and should list more expensive orders before cheaper ones.

Avoid using more tables than you need.

```
SELECT orderNumber, COUNT(*), ROUND(SUM(quantityOrdered*priceEach),2) as totalCost
FROM orderDetails
GROUP BY orderNumber
ORDER BY totalCost DESC;
```

SELF-CHECK HINT Order 10165 contains 18 line-items and has a total cost of 67392.85.

9. We want to find out whether *younger players* have an advantage in the Australian Open Men's Singles. Write a single SQL query that calculates the percentage of matches won by the younger player. For each year, the output table should list the *year* and the *percentage of matches* won by the younger player, with the percentage rounded to 2 decimal places.

   Each year there are exactly 127 matches in the Australian Open so you may directly hard-code the value 127 into your query.

```
SELECT yr, ROUND(COUNT(*) / 127.0 * 100, 2)
FROM ATPResult
WHERE winnerAge < loserAge
GROUP BY yr;
```

   SELF-CHECK HINT In 2001, 46.46 % of the matches were won by the younger of the two players.

10. Submission: `A10.sql` Do some male Australian Open players take significantly longer than others to win the matches that they do win? Write a single SQL query that lists, for each winner, the *name* of the player, the *number of matches* they have won, and the *average length* of those matches, rounded to 2 decimal places. Restrict the output to players who have won at least 10 matches, and sort the rows so that the fastest players are at the top. Note that we are only considering the matches that a player wins and remember that "at least 10" includes 10.

```
SELECT winnerName, COUNT(*), ROUND(AVG(minutes),2) AS avg
FROM ATPResult
GROUP BY winnerName
HAVING COUNT(*) > 9
ORDER BY avg;
```

   SELF CHECK HINT Andre Agassi is the fastest in this list, having won 30 matches with an average duration of just 108.13 minutes per match.

11. Submission: `A11.sql` Using Classic Models, write a single SQL query that lists, for each combination of order and product line, the *order number*, the *product line*, the *number* of line-items of that product line in the order, and the *total cost* (to the customer) of those line items, rounded to 2 decimal places. (You only need to list the product lines that actually occur in the order.)

```
SELECT O.orderNumber, P.productLine, COUNT(*), ROUND(SUM(O.quantityOrdered*O.priceEach),2)
FROM orderDetails O JOIN products P
ON O.productCode = P.productCode
GROUP BY O.orderNumber, P.productLine;
```

   SELF-CHECK HINT: Figure 1 shows that Order 10103 contains 3 line-items of Classic Cars, 7 of Trucks and Buses and 6 of Vintage Cars. The three line-items of Classic Cars cost a total of 14548.88.

12. Using Classic Models, write a single SQL query that lists each *order number* along with the *net profit*, rounded to 2 decimal places, that Classic Models made from that order. The table should have two columns named `orderNumber` and `netProfit`.

10100	Vintage Cars	4	10223.83
10101	Vintage Cars	4	10549.01
10102	Vintage Cars	2	5494.78
10103	Classic Cars	3	14548.88
10103	Trucks and B	7	20987.03
10103	Vintage Cars	6	14683.04
10104	Classic Cars	7	22003.45
10104	Trains	2	4476.87
10104	Trucks and B	4	13725.88

Figure 1: Orders broken down by product lines

```
SELECT orderNumber, ROUND(SUM(quantityOrdered*(priceEach-buyPrice)),2) AS netProfit
FROM orderDetails JOIN products USING (productCode)
GROUP BY orderNumber;
```

SELF-CHECK HINT: Order 10422 has two line items only, so is a good choice to check your code. This order consists of 51 units of `S18_1342`, which were sold for $91.44 each, and 25 units of `S18_1367` which were sold for $47.44 each, for a total cost of $5849.44. As Classic Models paid $60.62 for each `S18_1342` and $24.26 for each `S18_1367`, the items in this order cost Classic Models a total of $3698.12. Therefore the profit margin on this particular order is $5849.44-$3698.12 = $2151.32.

13. Classic Models has a "Important Customer" list, which is a list of all the customers who have made orders that (added together) have a total cost of at least $500000 (half a million dollars). Write a single SQL query that lists the *customer name* and *total order value* of these Important Customers.

```
SELECT C.customerName, SUM(D.quantityOrdered*D.priceEach) AS totalOrderCost FROM
    Customers C JOIN Orders O JOIN orderDetails D
WHERE C.customerNumber = O.customerNumber
AND O.orderNumber = D.orderNumber
GROUP BY C.customerName
HAVING totalOrderCost >= 500000;
```

The tables WTAResult and ATPResult contain data about individual matches in a different way to AFLResult,

In WTAResult and ATPResult it is easy to identify the winner, because the winner's name is always in the same column, namely winnerName. But in AFLResult the name of the winning team is sometimes in the column homeTeam and sometimes in the column awayTeam, and of course sometimes there is no winner at all, because the game can end in a draw.

So for AFLResult it is not clear how to write a SELECT statement that chooses the winning team — it requires some sort of *conditional execution* that changes the column being selected according to the values in other columns. In a procedural programming language, this would be easy to write, with something like the following:

```
if homeScore > awayScore:
  return homeTeam
elif homeScore < awayScore:
  return awayTeam
else:
  return 'DRAW'
```

But SQL is a declarative language, and hence has no direct "control flow" statements. Instead there are a couple of special commands that permit the user some limited control to alter the value returned based on a boolean condition.

The first of these is the CASE expression which has the following syntax:

```
CASE  WHEN <boolean_1> THEN <expr_1>
      WHEN <boolean_2> THEN <expr_2>
      WHEN ...
      ELSE <expr_3>
END
```

This entire expression is used as part of the SELECT statement to specify *one of the columns* of the output table. When a row is processed, the value of this column is the *first expression* for which the associated boolean condition is true. The ellipsis (dots) just indicate that you can use as many WHEN clauses as necessary.

An example makes this clearer:

```
SELECT daydate,
CASE
 WHEN homeScore > awayScore THEN homeTeam
 WHEN awayScore > homeScore THEN awayTeam
 ELSE NULL
END
FROM AFLResult;
```

This code defines a table with *two columns* — the first column is called daydate and is just copied from the row being processed, while the second column is *calculated* according to the specifications of the CASE statement. This compares homeScore to awayScore and chooses either homeTeam, awayTeam or NULL according to whether homeScore is greater than, less than, or equal to awayScore.

You can rename the column defined by the CASE statement in the usual way.

```
SELECT daydate,
CASE
 WHEN homeScore > awayScore THEN homeTeam
 WHEN awayScore > homeScore THEN awayTeam
```

```
 ELSE NULL
END AS winner
FROM AFLResult;
```

There is a second way to do this that has only recently been added to SQLite (version 3.32 or higher), namely the `IIF` statement, which is perhaps easier to read when there are only two choices.

> `IIF` was introduced in SQLite 3.32, so will not work if you are using an earlier version of SQLite. You can check which version you have if you use the dot command `.version` in the `sqlite3` terminal program, and you can install a more recent version if you wish.

This code lists the *older player* from each match, by simply testing whether `winnerAge` is greater than `loserAge` or not. (This ignores the possibility that both players have exactly the same age.)

```
SELECT yr,
       IIF(winnerAge > loserAge, winnerName, loserName) as olderPlayer
FROM ATPResult;
```

14. Using `AFLResult.db`, use a single SQL query to find the *average winning score* for each of the years in the database. Your output table should have two columns, one for the *year* and another for the *average winning score* for all the matches in that year.

    You will need to *calculate* the year from the `daydate` string which displays dates in the style `24-Sep-22`. To deduce that this date is in 2022, you need to extract the *last two digits* from the string and then add it to the number 2000.

    The SQLite function `SUBSTR` allows you to extract a *substring* from a string argument, you will need to look up the details for the syntax (at `https://www.sqlite.org/lang_corefunc.html#substr`) and experiment a bit.

    You will be helped by the fact that the `daydate` always uses 2 characters for the day, 3 characters for the month, and 2 characters for the year and separates them by hyphens. This means that the two characters representing the year always start at the 8th position in the string.

```
SELECT 2000+SUBSTR(day date,8,2) AS yr,
AVG(IIF(homeScore > awayScore, homeScore, awayScore))
FROM AFLResult
GROUP BY yr;
```

    Self-Check Hint: The average winning score in 2021 was 94.79227053140097.

15. We suspect that the playoff games at the end of the season might have lower winning scores than the games during the regular season. Write a single SQL query based on the one you wrote for the previous question, but now calculate the average winning score separately for the playoff and non-playoff games. So each row should contain a *year*, an *average winning score* and then either `'N'` or `'Y'` depending on whether the average is taken over playoff games or not.

    Figure 2 shows the first few rows of output. For example, in 2011 the average winning score in the regular season was about 113 points, while in the playoffs it was about 104 points.

```
SELECT 2000+SUBSTR(daydate,8,2) AS yr, playOff,
AVG(IIF(homeScore > awayScore, homeScore, awayScore))
FROM AFLResult
GROUP BY yr, playOff;
```

2011	N	113.11764705882354
2011	Y	104.22222222222223
2012	N	112.84848484848484
2012	Y	100.22222222222223
2013	N	111.5909090909091
2013	Y	95

Figure 2: Average wining scores