COMP3311 21T1 Database Systems

Prac Exercise 05

SQL Queries, Views, and Aggregates (ii)

Aims

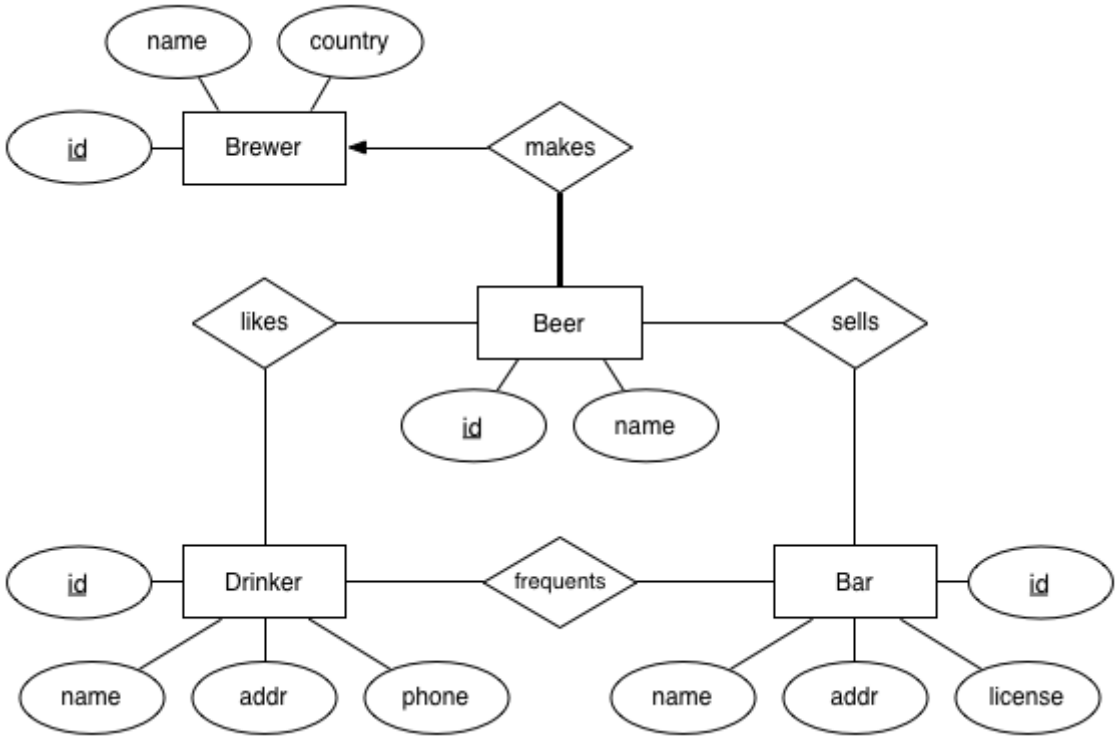
This exercise aims to give you practice in:

- asking SQL queries on a relatively simple schema
- using SQL aggregation and grouping operators
- writing SQL view definitions
- porting SQL across multiple database management systems

This exercise will **not** explain how to do everything in fine detail. Part of the aim of the exercise is that you explore how to use the PostgreSQL and SQLite systems. The documentation for these systems contains much useful information: PostgreSQL Manual, SQLite Manual. You should become familiar with where to find useful information in the documentation ASAP; you will need to know how to use PostgreSQL for the assignments and how to use SQLite for the exam.

Background

In lectures, we used a simple database about beers, bars and drinkers to illustrate aspects of querying in SQL. The database was designed to simplify queries by using symbolic primary keys (e.g., `Beers.name`). In practice, we don't use symbolic primary keys because: (a) they typically occupy more space than a numeric key, (b) symbolic names have an annoying tendency to change over time (e.g., you might change your email, and having email as a primary key creates a multitude of update problems). Therefore, we have designed a (slightly) more realistic schema for representing the same information:



The SQL schema will obviously be different to the schema used in lectures, and is available in the file:

```
/home/cs3311/web/21T1/pracs/05/schema.sql
```

This schema is written in portable SQL and so should load into both PostgreSQL and SQLite without errors.

If you're working on your laptop (and not via `putty`), you can grab copies of all files used in this Prac in the ZIP archive:

```
/home/cs3311/web/21T1/pracs/05/prac.zip
```

Setting up the PostgreSQL Database

Login to a machine with a PostgreSQL server running. If you already have a `beer2` database and you want to replace it, you will, of course, need to drop it first:

```
$ dropdb beer2
```

Then do the following:

```
$ createdb beer2
... make a new empty database ...
$ psql beer2 -f schema.sql
... load the schema ... produces CREATE TABLE, etc. messages ...
$ psql beer2 -f data.sql
... load the tuples ... poduces INSERT messages ...
```

You now have a database that you can use via:

```
$ psql beer2
... run SQL commands ...
```

Setting up the SQLite Database

Login to a machine with SQLite installed, change to the directory containing the `schema.sql` and `data.sql` files mentioned above, and do the following:

```
% sqlite3 beer2.db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite> .read schema.sql
sqlite> .read data.sql
sqlite> .quit
%
```

This will create a file called `beer2.db` in the current directory. You now have a database that you can use via:

```
$ sqlite3 beer2.db
... run SQL commands ...
```

Exercises

Use the two databases you created above to do the exercises below. The same view definitions should work in both databases. Perhaps you could alternate between developing and testing the view first in PostgreSQL and then testing it in SQLite, and vice versa. The aim is to get practice in building queries in both databases.

In the questions below, you are required to produce SQL queries to solve a range of data retrieval problems on this schema. For each problem, create a view called `Q`*n* which holds the "top-level" SQL statement that produces the answer (this SQL statement may make use of any views defined earlier in the Prac Exercise). In producing a solution for each problem, you may define as many auxiliary views as you like.

To simplify the process of producing these views, a template file (`queries.sql`) is available. While you're developing your views, you might find it convenient to edit the views in one window (i.e. edit the `queries.sql` file containing the views) and copy-and-paste the view definitions into another window running `psql` or `sqlite3`.

Note that the order of tuples in the results does not matter. As long as you have the same set of tuples, your view is correct. Remember that, in theory, the output from an SQL query is a set. Some test queries use an explicit ordering, but that should not be included in the view definition.

Note also that the sample outputs typically use column names that are different to the column names in the table. You should use the column names given in the sample output; treat them as part of description of the question.

Once you have completed each of the view definitions, you can test it simply by typing:

```
beer2=# select * from Qn;
```

and observing whether the result matches the expected result given below. Note that I'll give all the results in PostgreSQL format; the SQLite tuples don't look precisely the same, but it should be clear enough that it *is* the same set of tuples.

Queries on Beer Database v2

Write an SQL view to answer each of the following queries. Note that *none* of your queries should contain internal `id` values; all references to entities in queries should be via their name.

1. ***What beers are made by Toohey's?***

   In PostgreSQL, the results should look like:

   ```
   beer2=# select * from Q1;
       beer
   -------------
    New
    Old
    Red
    Sheaf Stout
   (4 rows)
   ```

2. ***Show beers with headings "Beer", "Brewer".***

   In PostgreSQL, the results should look like:

   ```
   beer2=# select * from Q2;
              Beer            |       Brewer
   ---------------------------+--------------------
    80/-                      | Caledonian
   ```

```
 Amber Ale                | James Squire
 Bigfoot Barley Wine      | Sierra Nevada
 Burragorang Bock         | George IV Inn
 Chestnut Lager           | Bridge Road Brewers
 Crown Lager              | Carlton
 Fosters Lager            | Carlton
 India Pale Ale           | James Squire
 Invalid Stout            | Carlton
 Melbourne Bitter         | Carlton
 New                      | Toohey's
 Nirvana Pale Ale         | Murray's
 Old                      | Toohey's
 Old Admiral              | Lord Nelson
 Pale Ale                 | Sierra Nevada
 Pilsener                 | James Squire
 Porter                   | James Squire
 Premium Lager            | Cascade
 Red                      | Toohey's
 Sink the Bismarck        | Brew Dog
 Sheaf Stout              | Toohey's
 Sparkling Ale            | Cooper's
 Stout                    | Cooper's
 Tactical Nuclear Penguin | Brew Dog
 Three Sheets             | Lord Nelson
 Victoria Bitter          | Carlton
(26 rows)
```

3. ***Find the brewers whose beers John likes.***

   In PostgreSQL, the results should look like:

```
beer2=# select * from Q3;
    brewer
---------------
 Brew Dog
 James Squire
 Lord Nelson
 Sierra Nevada
 Caledonian
(5 rows)
```

4. ***How many different beers are there?***

In PostgreSQL, the results should look like:

```
beer2=# select * from Q4;
 #beers
--------
     26
(1 row)
```

5. ***How many different brewers are there?***

   In PostgreSQL, the results should look like:

```
beer2=# select * from Q5;
 #brewers
----------
       12
(1 row)
```

6. ***Find pairs of beers by the same manufacturer*** (but no (a,b) and (b,a) pairs, or (a,a))

   In PostgreSQL, the results should look like:

```
beer2=# select * from Q6;
        beer1         |          beer2
----------------------+-------------------------
 Amber Ale            | Porter
 Amber Ale            | Pilsener
 Amber Ale            | India Pale Ale
 Bigfoot Barley Wine  | Pale Ale
 Crown Lager          | Victoria Bitter
 Crown Lager          | Melbourne Bitter
 Crown Lager          | Invalid Stout
 Crown Lager          | Fosters Lager
 Fosters Lager        | Victoria Bitter
 Fosters Lager        | Melbourne Bitter
 Fosters Lager        | Invalid Stout
 India Pale Ale       | Porter
 India Pale Ale       | Pilsener
 Invalid Stout        | Victoria Bitter
 Invalid Stout        | Melbourne Bitter
 Melbourne Bitter     | Victoria Bitter
 New                  | Sheaf Stout
 New                  | Red
```

```
 New                 | Old
 Old                 | Sheaf Stout
 Old                 | Red
 Old Admiral         | Three Sheets
 Pilsener            | Porter
 Red                 | Sheaf Stout
 Sink the Bismarck   | Tactical Nuclear Penguin
 Sparkling Ale       | Stout
(26 rows)
```

7. ***How many beers does each brewer make?***

   In PostgreSQL, the results should look like:

```
beer2=# select * from Q7 order by brewer;
       brewer         | nbeers
----------------------+--------
 Brew Dog             |      2
 Bridge Road Brewers  |      1
 Caledonian           |      1
 Carlton              |      5
 Cascade              |      1
 Cooper's             |      2
 George IV Inn        |      1
 James Squire         |      4
 Lord Nelson          |      2
 Murray's             |      1
 Sierra Nevada        |      2
 Toohey's             |      4
(12 rows)
```

8. ***Which brewer makes the most beers?***

   In PostgreSQL, the results should look like:

```
beer2=# select * from Q8;
 brewer
---------
 Carlton
(1 row)
```

9. ***Beers that are the only one by their brewer.***

In PostgreSQL, the results should look like:

```
beer2=# select * from Q9;
       beer
------------------
 80/-
 Burragorang Bock
 Chestnut Lager
 Nirvana Pale Ale
 Premium Lager
(5 rows)
```

10. ***Beers sold at bars where John drinks.***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q10 order by beer;
       beer
-------------------
 Burragorang Bock
 New
 Old
 Old Admiral
 Pale Ale
 Sink the Bismarck
 Sparkling Ale
 Three Sheets
 Victoria Bitter
(9 rows)
```

    You might like to consider a variation on this query to find just the beers that John likes that are sold in the bars where he drinks. The solution is given in the solutions file.

11. ***Bars where either Gernot or John drink.***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q11 order by bar;
      bar
------------------
 Australia Hotel
 Coogee Bay Hotel
 Local Taphouse
```

```
 Lord Nelson
 Royal Hotel
(5 rows)
```

12. ***Bars where both Gernot and John drink.***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q12;
     bar
-------------
 Lord Nelson
(1 row)
```

13. ***Bars where John drinks but Gernot doesn't***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q13;
       bar
------------------
 Australia Hotel
 Local Taphouse
 Coogee Bay Hotel
(3 rows)
```

14. ***What is the most expensive beer?***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q14;
       beer
-------------------
 Sink the Bismarck
(1 row)
```

15. ***Find bars that serve New at the same price as the Coogee Bay Hotel charges for VB.***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q15;
     bar
-------------
```

```
 Royal Hotel
(1 row)
```

16. ***Find the average price of common beers*** (where "common" = served in more than two hotels)

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q16;
      beer       | AvgPrice
-----------------+----------
 Victoria Bitter |     2.40
 New             |     2.59
 Old             |     2.68
(3 rows)
```

17. ***Which bar sells 'New' cheapest?***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q17;
     bar
--------------
 Regent Hotel
(1 row)
```

18. ***Which bar is most popular?*** (Most drinkers)

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q18;
       bar
------------------
 Coogee Bay Hotel
 Lord Nelson
(2 rows)
```

19. ***Which bar is least popular?*** (May have no drinkers)

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q19;
       bar
-----------------
```

```
 Local Taphouse
 Marble Bar
 Regent Hotel
 Australia Hotel
 Royal Hotel
(5 rows)
```

20. ***Which bar is most expensive?*** (Highest average price)

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q20;
      bar
----------------
 Local Taphouse
(1 row)
```

21. ***Which beers are sold at all bars?***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q21;
 beer
------
(0 rows)
```

    i.e. no beers are sold at all bars.

22. ***Price of cheapest beer at each bar?***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q22;
       bar         | min_price
-------------------+-----------
 Coogee Bay Hotel  |      2.25
 Local Taphouse    |      7.50
 Royal Hotel       |      2.30
 Australia Hotel   |      3.00
 Regent Hotel      |      2.20
 Marble Bar        |      2.80
 Lord Nelson       |      3.00
(7 rows)
```

23. ***Name of cheapest beer at each bar?***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q23;
       bar         |      beer
-------------------+-----------------
 Australia Hotel   | New
 Coogee Bay Hotel  | New
 Lord Nelson       | New
 Marble Bar        | New
 Marble Bar        | Victoria Bitter
 Regent Hotel      | New
 Regent Hotel      | Victoria Bitter
 Royal Hotel       | Victoria Bitter
 Royal Hotel       | New
 Local Taphouse    | Pale Ale
(10 rows)
```

24. ***How many drinkers are in each suburb?***

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q24;
   addr    | count
-----------+-------
 Randwick  |     1
 Mosman    |     1
 Newtown   |     1
 Clovelly  |     1
(4 rows)
```

25. ***How many bars in suburbs where drinkers live?*** (must include suburbs with no bars)

    In PostgreSQL, the results should look like:

```
beer2=# select * from Q25;
   addr    | #bars
-----------+-------
 Randwick  |     1
 Mosman    |     0
 Newtown   |     0
```

```
 Clovelly |     0
(4 rows)
```

You should attempt the above exercises before looking at the PostgreSQL sample solutions.

Queries in SQLite

Now that you've attempted the above exercises in PostgreSQL, let's consider how things would work in SQLite. This provides an interesting exercise in SQL portability, since both databases support "standard SQL".

Make a copy of your `queries.sql` file and start testing the views that you created for PostgreSQL for SQLite. If you want to make the query results from SQLite look more like those from PostgreSQL, run the following commands at the start of your SQLite session:

```
$ sqlite3 beer2.db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite> .headers on
sqlite> .mode column
sqlite> .width 20 20 20 20
sqlite> ... continue with your SQL queries ...
```

Note that SQLite is not quite as smart as PostgreSQL when it comes to choosing column widths. All columns will be 20 characters wide if you use the above settings. Note that any values that are wider than 20 characters will be truncated. If you can't be bothered typing these commands each time, put them in a file called `.sqliterc` in your home directory and they'll be executed each time you run `sqlite3`.

As we noticed in the previous Prac Exercise, SQLite doesn't support the definition of views via:

```
create or replace view ViewName as ...
```

The fix for this is simple enough. Change all of the view definitions to something like:

```
drop view if exists ViewName;
create view ViewName as ...
```

Note that this approach would cause problems in PostgreSQL, which records which views depend on which other views. Inevitably, you would try to drop a view defined early in the SQL file that is used by view later in the file. In PostgreSQL, you can add the keyword `cascade` to ensure that you not only drop the view you asked to drop, but also all the views that make use of it. SQLite doesn't do this dependency checking, and will allow you to drop a view, even if other views use it. You won't notice until you try to use one of the remaining views and will then be told that the view you dropped is undefined. If you redefine the view straight away (as in the above), then the dependency problem never arises.

If you're a `vim` user, the following command will convert all of the current `create view` statements into an appropriate form. If you're not a `vim` user, read and weep ... or post the equivalent for your editor of choice.

```
:s/^create or replace view \([^ ]*\)/drop view if exists \1;^Mcreate view \1/
```

Note: the ^M is achieved by typing the two-character sequence control-V control-M. Also, if you attempt this in `vim` and mess it up, you can undo the effects simply by typing the character `u`.

To avoid the fact that SQLite doesn't support view definitions of the form:

```
create view ViewName(attr1, attr2, ...)
as
select expr1, expr2, ...
```

simply use the equivalent form:

```
create view ViewName
as
select expr1 as attr1, expr2 as attr2, ...
```

Once you've made the above changes, many of the views will work correctly in both PostgreSQL and SQLite.

One "problem" is that SQLite doesn't have quite the same formatting options for result values. PostgreSQL allows you to cast to a `numeric` value to control the number of decimal places in real number values; unfrotunately, SQLite doesn't quite allow the same. Don't worry if your real results don't look the same in SQLite.

You should attempt the above exercises before looking at the SQLite sample solutions.