COMP9311 16s2

Prac Exercise 05

Functions and Aggregates

Database Systems

Last updated: **Wednesday 31st August 12:09pm**
Most recent changes are shown in red ... older changes are shown in brown.

Aims

This exercise aims to give you practice in:

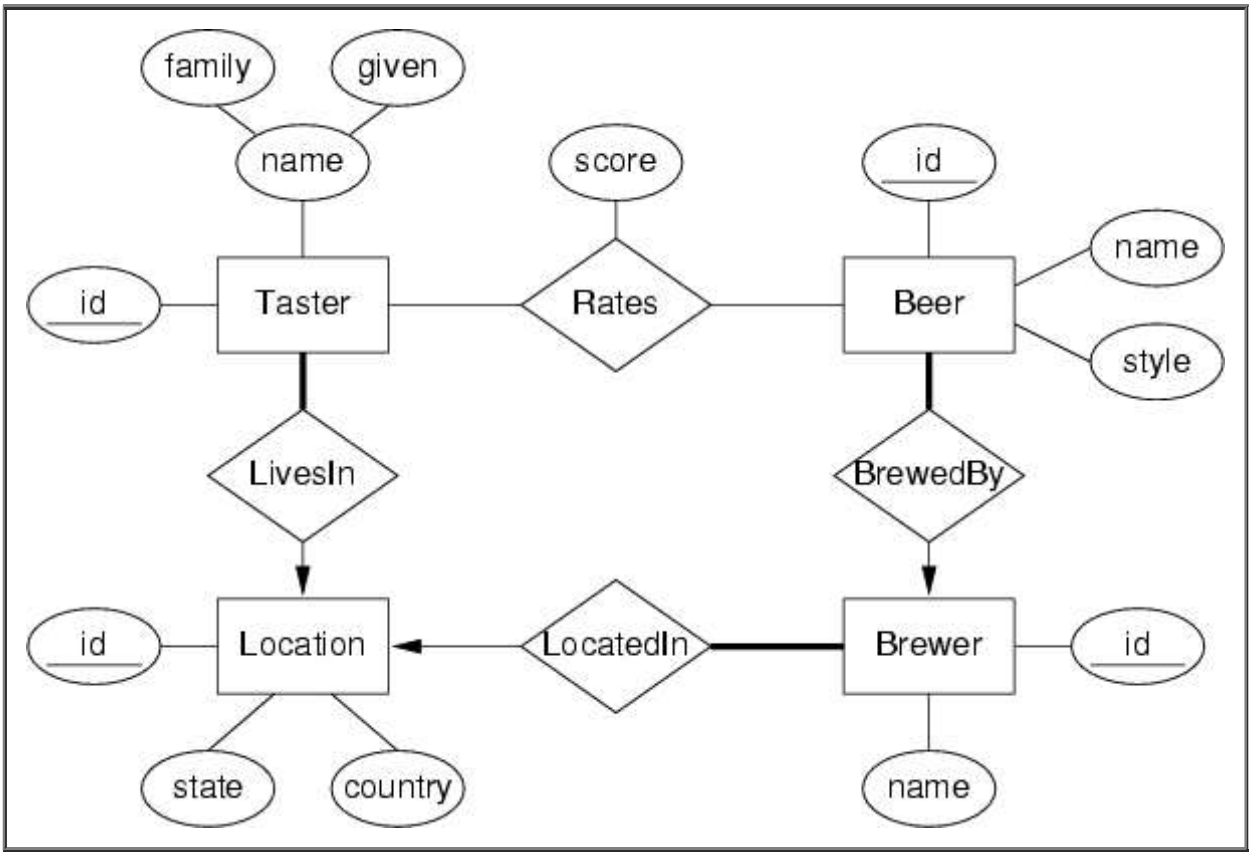
- defining views based on simple SQL queries
- defining SQL functions
- defining PLpgSQL functions
- adding new aggregate operators

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 system. A very important tool for this is the PostgreSQL Manual. In particular, the following sections of the Manual are relevant: the SQL `CREATE FUNCTION` statement; the SQL `CREATE AGGREGATE` statement; the PLpgSQL language description.

Background

Consider an on-line beer rating system which describes beers, breweries, tasters, ratings, etc. Tasters are the users of the system who provide ratings of various beers. Each beer is brewed according to a particular style, by a given brewery. Since we are interested in the location of tasters and brewers, we maintain a collection of locations which is shared between the two. Ratings are based on simple 1..5 numerical scale (5 means "world classic", 1 means "barely drinkable").

An possible E/R design for this scenario is as follows:



Note that the E/R design includes numeric primary key attributes (the ones called `id`) to allow for the fact that names might clash. (Using such attributes also simplifies the derived relational model, by allowing all foreign keys to be represented by a single attribute.)

A relational schema corresponding to this E/R design is available in the file

```
/home/cs9311/web/16s2/prac/05/schema.sql
```

A file containing populated tables for this schema is available in the file

```
/home/cs9311/web/16s2/prac/05/data.sql
```

Note that the `data.sql` file is using PostgrSQL's bulk loading mechanism `COPY`; check the PostgreSQL manual for details on how this works. The `COPY` statement is a part of PostgreSQL's SQL dialect, and can only be used by the PostgreSQL super-user (which is you when you're running your own server, as you are on Grieg). Also, since the `id` attributes are serial primary keys, they have an associated sequence, which needs to be set to the correct value after the tuples are loaded (so that any new tuples will have appropriate sequence values generated).

There is also a template file giving headers for the views, functions and aggregates you need to write.

```
/home/cs9311/web/16s2/prac/05/prac05.sql
```

Grab a copy of this file to save yourself some typing.

The relational model attempts to capture all of the semantics of the E/R design. However, there is one difference between the relational model and the E/R design in that beer styles have been converted into entities. This is primarily to ensure that all beer style information looks consistent (e.g. we don't have some beers called "lager" and others called "laager").

You should create a new database called "beers" and load the schema and data into this database. The following commands wil do this:

```
$ createdb beers
CREATE DATABASE
$ psql beers -f /home/cs9311/web/16s2/prac/05/schema.sql
... which will produce lots of NOTICE messages ...
$ psql beers -f /home/cs9311/web/16s2/prac/05/data.sql
... which will produce messages about table ids ...
```

Once the schema and data are loaded, check that everything is in order by running the following queries and seeing whether you get the same results:

```
beers=# select count(*) from Ratings;
 count
-------
    32
(1 row)

beers=# select given from Taster order by given;
 given
--------
 Adam
 Geoff
 Hector
 Jeff
 John
 Peter
 Raghu
 Ramez
 Rose
 Sarah
(10 rows)
```

If the database doesn't look correct, try to work out what went wrong and then try to load the data correctly. Once you're statisfied that the database is correct, continue with the exercises.

Exercises

You might want to collect together all your view definitions in a file. A template is available for this, to save some typing.

1. Find out who made what ratings.

   If you look at the `Ratings` table, you'll see that it contains just a bunch of numbers. This is good for being compact, but hard to work out what it all means. Write an SQL query that will put together the data from the `Ratings` table with the `Taster` and `Beer` tables so that you get a better idea of who rated which beer. The result should display the taster's name (given name only), along with the name of the beer, its brewer and the rating. Order the table by the tasters' names; for a given taster, sort in descreasing order of rating (i.e. highest-rated beer first).

   Place your query into a view definition like this:

   ```
   create or replace view AllRatings(taster,beer,brewer,rating)
   ```

   Note that the attributes to the view definitions supply names for whatever values appear in the `select` clause of the SQL query. If you want to think of the view as a "virtual table", then these represent the names of the attributes of that table.

   When you invoke the view, you should see output that looks like this:

```
 taster |          beer           |          brewer          | rating
--------+-------------------------+--------------------------+--------
 Adam   | Old                     | Toohey's                 |      4
 Adam   | Victoria Bitter         | Carlton and United       |      1
 Adam   | New                     | Toohey's                 |      1
 Geoff  | Redback                 | Matilda Bay Brewing      |      4
 Geoff  | James Squire Pilsener   | Maltshovel Brewery       |      4
 Geoff  | Empire                  | Carlton and United       |      3
 Hector | Sierra Nevada Pale Ale  | Sierra Nevada            |      4
 Hector | Fosters                 | Carlton and United       |      3
 Jeff   | Sierra Nevada Pale Ale  | Sierra Nevada            |      4
 Jeff   | Burragorang Bock        | Scharer's Little Brewery |      3
 Jeff   | Rasputin                | North Coast Brewing      |      1
 John   | Sierra Nevada Pale Ale  | Sierra Nevada            |      5
 John   | 80/-                    | Calendonian Brewing      |      4
 John   | Rasputin                | North Coast Brewing      |      4
 John   | Empire                  | Carlton and United       |      3
 John   | Chimay Red              | Chimay                   |      3
 John   | Crown Lager             | Carlton and United       |      2
 John   | Victoria Bitter         | Carlton and United       |      1
 Peter  | XXXX                    | Castlemaine/Perkins      |      5
 Raghu  | Old Tire                | New Glarus Brewing       |      5
 Raghu  | Sierra Nevada Pale Ale  | Sierra Nevada            |      3
 Raghu  | Rasputin                | North Coast Brewing      |      3
 Ramez  | Sierra Nevada Pale Ale  | Sierra Nevada            |      4
 Ramez  | Bigfoot Barley Wine     | Sierra Nevada            |      3
 Rose   | Redback                 | Matilda Bay Brewing      |      5
 Sarah  | Burragorang Bock        | Scharer's Little Brewery |      4
 Sarah  | James Squire Amber Ale  | Maltshovel Brewery       |      3
 Sarah  | James Squire Pilsener   | Maltshovel Brewery       |      3
 Sarah  | Old                     | Toohey's                 |      3
 Sarah  | Scharer's Lager         | Scharer's Little Brewery |      3
 Sarah  | New                     | Toohey's                 |      2
 Sarah  | Victoria Bitter         | Carlton and United       |      1
(32 rows)
```

2. Find out what is my favourite beer.

   Clearly, you can work out the answer to this once you've solved the query above. However, try to write a query that returns a single tuple with the name of the beer(s) and brewer(s) for the beer(s) that John Shepherd rates highest.

   Place your query into a view definition like this:

   ```
   create or replace view JohnsFavouriteBeer(brewer,beer)
   ```

   When you invoke the view, you should see output that looks like this:

   ```
   beers=# select * from JohnsFavouriteBeer ;
       brewer     |   beer
   ---------------+----------
    Sierra Nevada | Pale Ale
   (1 row)
   ```

   (Hint: adapt the view from the previous question in defining this new view)

3. Find out anyone's favourite beer.

   The above view seems to provide a useful operation, but seems a little restrictive. Surely I don't always want to know what is John's favourite beer. Maybe I want to know what is Adam's favourite beer, or Sarah's favourite beer. The queries to do this would be almost the same as the one used in the view above, but with the name changed. Which naturally raises the question "Can view definitions be parameterised?". The answer is "No" in standard SQL, but PostgreSQL provides functions, which can be used to implement something like this.

   PostgreSQL functions can be defined in a number of languages, including SQL. An SQL function typically contains a single SQL query, into which parameters to the function can be interpolated, thus providing a parameterisable query. (Note that SQL functions can contain an arbitrary sequence of SQL statements, including updates, separated by semi-colons. The result of such a function is the the result of the last SQL statement.)

   For the favourite beer example, define a function which takes as input the complete name of a taster (as a text string) and returns one or more tuples containing the name of the brewer and the beer, as in the above example.

   Why "one or more tuples"? Maybe I have several equally favourite beers. Always consider this possibility when faced with an information request like "Find the largest ..." or "Find the most expensive ..."; there may be a number of equally large/expensive things in the database. Also, do not assume that a given taster's maximum rating will be equal to 5; there may be people who rate beers harshly and never give a better rating than 4.

   Make sure that you test your function on all tasters, to ensure that it's working properly for the people who have several equal favourite beers, or those who haven't given a rating of 5.

   Before we can define a function, we need to define a type for the return tuples. Note that this happens automatically for views, but not for functions. The tuple type can be defined as:

   ```
   create type BeerInfo as (brewer text, beer text);
   ```

   The function would then be defined like this:

   ```
   create or replace function FavouriteBeer(taster text) returns set of BeerInfo
   ```

   and could be used as follows:

   ```
   beers=# select * from FavouriteBeer('John');
       brewer     |   beer
   ---------------+----------
    Sierra Nevada | Pale Ale
   ```

```
(1 row)

beers=# select * from FavouriteBeer('Adam');
  brewer  | beer
----------+------
 Toohey's | Old
(1 row)
```

Note that since the function returns a set of tuples, it can be treated like a dynamic table (somewhat like a view) and needs to be used in the context where a table would normally be used, i.e. in the `from` clause of a `select` statement. In fact, you can use such a function in the `select` clause, and it gives a plausible result; try it out and see if you can explain what PostgreSQL is doing, e.g.

```
beers=# select FavouriteBeer('John');
```

Some things to note about SQL function definitions:

- the body of every PostgreSQL function (including SQL functions) is defined as a single long string, wrapped in unusual quotes `$$ ... $$`. Because of their unusual quoting, they are different from other strings in PostgreSQL in that embedded single quotes do *not* need to be doubled.
- we have given a name to the parameter of the function, but within the function body, this name may not be used; in SQL functions, parameters must be referred to via positional notation `$1`, `$2`, etc. (named parameters may, however, be used in other kinds of functions e.g. plpgsql ones)
- an SQL function consists of a single SQL statement, similar to a view; unlike a view, the function has parameters and the SQL statement may refer to these parameters (using positional notation)

4. What style is that beer?

Sometimes beers are named after their style (e.g. Sierra Nevada Pale Ale). Other times, imaginitive names are used (e.g. Rooftop Red, Old Peculier). For these latter ones, we may know the name and want to discover what the style is. Write an SQL function that takes two text string arguments (the name of a brewer and the name of a beer) and returns the text string giving the style of the beer). The function should be defined as:

```
create or replace function BeerStyle(brewer text, beer text) returns text
```

and used as e.g.

```
beers=# select BeerStyle('Sierra Nevada','Pale Ale');
 beerstyle
-----------
 Pale Ale
(1 row)
```

The function should give the correct result, regardless of the text case of the input parameters, so the following query should also work:

```
beers=# select BeerStyle('sierra nevada','pale ale');
 beerstyle
-----------
 Pale Ale
(1 row)
```

However, spelling mistakes or unknown names, will cause a null result to be returned, e.g.

```
beers=# select BeerStyle('sieera nevada','pale ale');
 beerstyle
-----------

(1 row)
```

5. Consider the following PostgreSQL SQL function to produce a representation of a taster's address:

```
create or replace function TasterAddress(text) returns text
as $$
   select loc.state||', '||loc.country
   from   Taster t, Location loc
   where  t.given = $1 and t.livesIn = loc.id
$$ language sql;
```

This function would be used as follows:

```
beers=# select tasterAddress('John');
       tasteraddress
---------------------------
 New South Wales, Australia
(1 row)

beers=# select tasterAddress('Jeff');
   tasteraddress
--------------------
 California, U.S.A.
(1 row)
```

The function works ok for people who have both a state and a country, but fails for people who have just a country specified; it gives a null address, when clearly some address information is known. To observe the bug, try to find Sarah's address. Modify the function so that it produces a sensible result when either the country or the state is null. Once you've fixed this bug, you ought to be able to get a result like:

```
beers=# select tasterAddress('Sarah');
 tasteraddress
---------------
 England
(1 row)
```

Hint: take a look at the case construct and the coalesce() function.

6. Define a function that produces a summary of beer tasting.

The function `BeerSummary` returns a summary for each beer consisting of the name of the beer, followed by its average rating (to 1 decimal place), followed by a comma-separated list of the given names of the people who tasted and rated the beer.

Use the following header in defining the function:

```
create or replace function BeerSummary() returns text
```

The function should produce the following results on the example database:

```
beers=# select BeerSummary();
            beersummary
-----------------------------------------

 Beer:    80/-
 Rating:  4.0
 Tasters: John

 Beer:    Bigfoot Barley Wine
 Rating:  3.0
 Tasters: Ramez
```

```
 Beer:    Burragorang Bock
 Rating:  3.0
 Tasters: Jeff, Sarah

 Beer:    Chimay Red
 Rating:  3.0
 Tasters: John

 ... a bunch of text omitted here to save space ...

 Beer:    Sierra Nevada Pale Ale
 Rating:  4.0
 Tasters: Hector, Jeff, John, Raghu, Ramez

 Beer:    Victoria Bitter
 Rating:  1.0
 Tasters: Adam, John, Sarah

 Beer:    XXXX
 Rating:  5.0
 Tasters: Peter

(1 row)
```

You should make use of the `AllRatings` view in producing the summary. You should **not** make use of the `concat()` aggregate in this function; note that `concat()` produces the taster list in a different format to that given above (i.e. `concat()` doesn't put spaces after the commas in its taster lists).

7. Define a new aggregation operator to concatenate a column of strings.

   Define the aggregation operator as follows:

   ```
   create aggregate concat ( ... )
   ```

   You could use this operator to get a list of taster names as follows:

   ```
   beers=# select concat(given) from Taster ;

                        concat
   ------------------------------------------------------
    John,Adam,Jeff,Sarah,Raghu,Ramez,Hector,Geoff,Peter,Rose
   (1 row)
   ```

8. Define a view that produces a summary of beer tasting.

   The view should return a set of tuples where each tuple contains the name of the beer, the average rating (to 1 decimal place), and a comma-separated list of the given names of people who tasted and rated the beer.

   Use the following header in defining the view:

   ```
   create or replace view BeerSummary(beer,rating,tasters)
   ```

   The view should produce the following results on the example database:

   ```
   beers=# select * from BeerSummary;
            beer          | rating |           tasters
   -----------------------+--------+-----------------------------
    Redback               |    4.5 | Geoff,Rose
    Fosters               |    3.0 | Hector
    New                   |    1.5 | Adam,Sarah
    Empire                |    3.0 | Geoff,John
   ```

```
 Old Tire                 |  5.0  | Raghu
 Old                      |  3.5  | Adam,Sarah
 80/-                     |  4.0  | John
 Chimay Red               |  3.0  | John
 Crown Lager              |  2.0  | John
 James Squire Amber Ale   |  3.0  | Sarah
 Sierra Nevada Pale Ale   |  4.0  | Hector,Jeff,John,Raghu,Ramez
 Rasputin                 |  2.7  | Jeff,John,Raghu
 Burragorang Bock         |  3.5  | Jeff,Sarah
 XXXX                     |  5.0  | Peter
 Scharer's Lager          |  3.0  | Sarah
 Bigfoot Barley Wine      |  3.0  | Ramez
 Victoria Bitter          |  1.0  | Adam,John,Sarah
 James Squire Pilsener    |  3.5  | Geoff,Sarah
(18 rows)
```

You should make use of the `AllRatings` view in producing the summary.

[Sample Solutions]