Assignment 1
Queries and Functions on BeerDB

Last updated: **Wednesday 1st March 3:49pm**
Most recent changes are shown in red ... older changes are shown in brown.

Aims

This assignment aims to give you practice in

- reading and understanding a small relational schema (BeerDB)
- implementing SQL queries and views to satisfy requests for information
- implementing PLpgSQL functions to aid in satisfying requests for information

The goal is to build some useful data access operations on the BeerDB database, which contains a wealth of information about everyone's* favourite beverage, One aim of this assignment is to use SQL queries (packaged as views) to extract such information. Another is to build PlpgSQL functions that can support higher-level activities, such as might be needed in a Web interface.

* well, mine anyway ...

Summary

Submission:	Login to Course Web Site > Assignments > Assignment 1 > [Submit] > upload `ass1.sql` or, on a CSE server, `give cs3311 ass1 ass1.sql`
Required Files:	`ass1.sql` (contains both SQL views and PLpgSQL functions)
Deadline:	21:00 Friday 14 October
Marks:	**15 marks** toward your total mark for this course
Late Penalty:	0.2 *marks* off the ceiling mark for each hour late

How to do this assignment:

- read this specification carefully and completely
- create a directory for this assignment
- copy the supplied files into this directory
- login to `d2` and run your PostgreSQL server** (or run a PostgreSQL servr installed on your home machine)
- load the database and start exploring
- complete the tasks below by editing `ass1.sql`
- test your work on `d2`, which is where it's tested
- submit `ass1.sql` via WebCMS or `give`

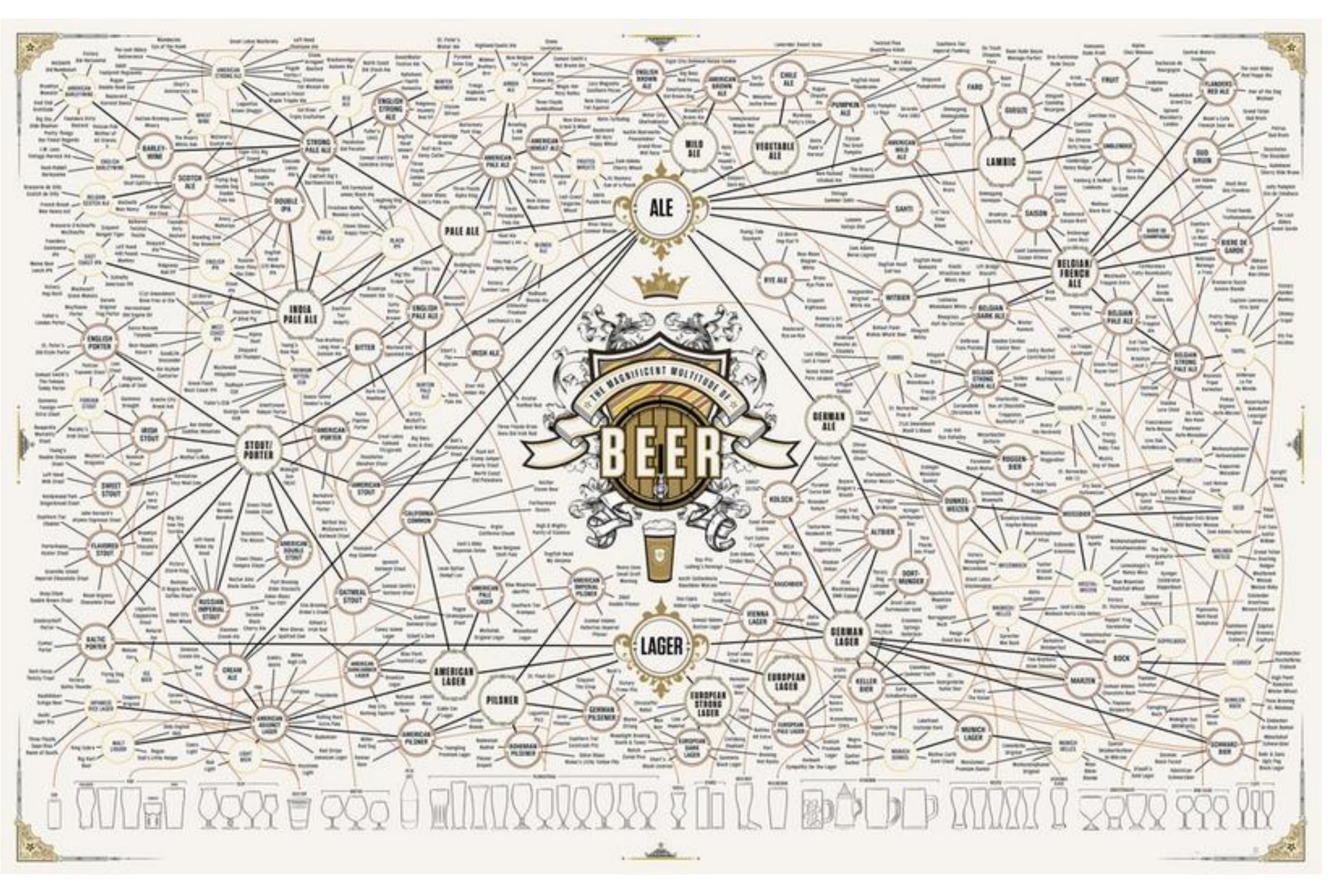
Details of the above steps are given below. Note that you can put the files wherever you like; they do not have to be under your `/localstorage` directory. You also edit your SQL files on hosts other than `d2`. The only time that you need to use `d2` is to manipulate your database. Since you can work at home machine, you don't have to use `d2` at all while *developing* your solution, but you should definitely test it there before submitting.

Introduction

In order to work with a database, it is useful to have some background in the domain of data being stored. Here is a very quick tour of **beer**. If you want to know more, see the Wikipedia Beer Portal.

Beer is a fermented drink based on grain, yeast, hops and water. The grain is typically malted barley, but wide variety of other grains (e.g. oats, rye) can be used. There are a wide variety of beers, differing in the grains used, the yeast strain, and the hops. More highly roasted grains produce darker beers, different types of yeast produce different flavour profiles, and hops provide aroma and bitterness. To add even more variety, adjuncts (e.g. sugar, chocolate, flowers, pine needles, to name but a few) can be added.

The following diagram gives a hint of the variety of beer styles:



To build a database on beer, we need to consider:

- beer styles (e.g. lager, IPA, stout, etc., etc.)
- ingredients (e.g. varieties of hops and grains, and adjuncts)
- breweries, the facilities where beers are brewed
- beers, specific recipes following a style, and made in a particular brewery

Specific properties that we want to consider:

- ABV = alcohol by volume, a measure of a beer's strength
- IBU = international bitterness units
- each beer style has a range of ABVs for beers in that style
- for each beer, we would like to store
  - its name (brewers like to use bizarre or pun-based names for their beers)
  - its style, actual ABV, actual IBU (optional), year it was brewed
  - type and size of containers it's sold in (e.g. 375mL can)
  - its ingredients (usually a partial list because brewers don't want to reveal too much)
- for each brewery, we would like to store
  - its name, its location, the year it was founded, its website

The schema is described in more detail both as an ER model and an SQL schema in the schema page.

Doing this Assignment

The following sections describe how to carry out this assignment. Some of the instructions must be followed exactly; others require you to exercise some discretion. The instructions are targetted at people doing the assignment on `d2`. If you plan to work on this assignment at home on your own computer, you'll need to adapt the instructions to "local conditions".

If you're doing your assignment on the CSE machines, some commands must be carried out on `d2`, while others can (and probably should) be done on a CSE machine other than `d2`. In the examples below, we'll use `vxdb$` to indicate that the comand must be done on `d2` and `cse$` to indicate that it can be done elsewhere.

Setting Up

The first step in setting up this assignment is to set up a directory to hold your files for this assignment.

```
cse$ mkdir /my/dir/for/ass1
cse$ cd /my/dir/for/ass1
cse$ cp /home/cs3311/web/22T3/assignments/ass1/ass1.sql ass1.sql
cse$ cp /home/cs3311/web/22T3/assignments/ass1/ass1.dump ass1.dump
```

This gives you a template for the SQL views and PLpgSQL functions that you need to submit. You edit this file, (re)load the definitions into the database you created for the assignment, and test it there.

Speaking of the database, we have a modest sized database of all the beers that I've tasted over the last few years. We make this available as a PostgreSQL dump file. If you're working at home, you will need to copy it onto your home machine to load the database.

The next step is to set up your database:

```
... login to d2, source env, run your server as usual ...
... if you already had such a database
vxdb$ dropdb ass1
... create a new empty atabase
vxdb$ createdb ass1
... load the database, saving the output in a file called log
vxdb$ psql ass1 -f ass1.dump > log 2>&1
... check for error messages in the log; should be none
vxdb$ grep ERR log
... examine the database contents ...
vxdb$ psql ass1
```

The database loading should take less than 10 seconds on `d2`. The `ass1.dump` files contains the schema and data in a single file, along with a simple PLpgSQL function (`dbpop*()`).

If you're running PostgreSQL at home, you'll need to load both `ass1.sql` and `ass1.dump`.

Think of some questions you could ask on the database (e.g. like the ones in the lectures) and work out SQL queries to answer them.

One useful query is

```
ass1=# select * from dbpop();
```

This will give you a list of tables and the number of tuples in each. The `dbpop()` function is written in PLpgSQL, and makes use of the PostgreSQL catalog. We'll look at this later in the term.

Your Tasks

Answer each of the following questions by typing SQL or PLpgSQL into the `ass1.sql` file. You may find it convenient to work on each question in a temporary file, so that you don't have to keep loading *all* of the other views and functions each time you change the one you're working on. Note that you can add as many auxuliary views and functions to `ass1.sql` as you want. However, make sure that *everything* that's required to make all of your views and functions work is in the `ass1.sql` file before you submit.

Note #1: many of the queries are phrased in the singular e.g. "Find the beer that ...". Despite the use of "beer" (singular), it is possible that multiple beers satisfy the query. Because of this you should, in general, avoid the use of `LIMIT 1`.

Note #2: the database is not a complete picture of beers in the Real World. Treat each question as being prefaced by "According to the BeerDB database ...".

Note #3: you can assume that the names for styles and breweries are unique; you *cannot* assume this for beer names.

There are examples of the results of each view and function in the Examples page.

Q0 (2 marks)

Given that you've alread y taken multiple programming courses, we should be able to assume that you'll express your code with good style conventions. But, just in case ...

You must ensure that your SQL queries follow a consistent style. The one I've been using in the lectures is fine. An alterntive, where the word `JOIN` comes at the start of the line, is also OK. The main thing is to choose one style and use it consistently.

Similarly, PLpgSQL should be laid out like you would lay out any procedural programming language. E.g. bodies of loops should be indented from the `FOR` or `WHILE` statement that introduces them. E.g. the body of a function should be indented from the `BEGIN`...`END`.

Ugly, inconsistent layout of SQL queries and PLpgSQL functions will be penalised.

Q1 (2 marks)

There has been an explosion in the number of new small breweries over the last decade, and the Sydney area is no exception. Define an SQL view `Q1(brewery,suburb)` that gives the names and suburbs of all breweries founded in the Sydney metropolitan area in 2020. It is possible that we may not know the suburb (`town` field) for all breweries.

Q2 (2 marks)

Brewers generally come up abstract or whimiscal names for their beers. Sometimes, though, they simply name the beer after its style (e.g. a lager beer called "Lager"). Define an SQL view `Q2(beer,brewery)` that gives the names and breweries of all beers whose name is the same as their style.

Q3 (2 marks)

It is generally considered that the craft beer revolution began in California in the 1970's/80's. Define a view `Q3(brewery,founded)` that gives the name and foundation year of the oldest brewery located in California. Presumably, this brewery is "where it all started".

Q4 (2 marks)

Nowadays, the most common beer style is IPA (India Pale Ale). It has become so popular that almost every brewery makes their own variation on the style, and there are now a proliferation of styles based on IPA. All of these styles have the string "IPA" somewhere in the style name, always in all upper-case. Define a view `Q4(style,count)` that gives a list of all IPA variations and a count of the number of beers brewed in that style.

Q5 (2 marks)

Brewery locations in this database have varying levels of accuracy. For all beers, we know the country. We may know the region, if the country has regions. If the brewery is located somewhere in a metropolitan area (e.g. Sydney), we might record that. If we know precisely which town/suburb it is located in, we would record that. Define a view `Q5(brewery,location)` that gives the name and most precise location of each brewery located in California. If we know the town/suburb name, use that. If not, give the name of the metropolitan area (e.g. San Diego) where it's located.

You can assume that every location will have at least a metro or a town. As an exercise, develop a table constraint that could be applied to `Locations` to ensure this.

Q6 (3 marks)

Some beers are aged in wooden barrels (often oak barrels previously used for whisky) after being fermented and before being bottled. This information is recorded in the `notes` field of the beer, but may not always be recorded consistently; it is guaranteed to have the two words "barrel" and "aged", but they may be separated within the string, may occur in either order, and may be a mix of upper- and lower-case characters, Define a view `Q6(beer,brewery,abv)` that shows the strongest barrel-aged beer(s), giving the beer name, the brewery name(s), and the strength (ABV = percentage alcohol by volume).

You need to consider two cases. One where the word "barrel" comes before the word "aged" (e.g. "whisky-barrel aged"), The other where "barrel" comes after "aged" (e.g. "aged in bourbon barrels"). In this case "barrel" may be singular or plural.

Q7 (2 marks)

Hops are an essential ingredient in beer brewing. They impart aroma, flavour and bitterness, and also act as a preservative. Some hop varieties are very popular, while others are rarely used. Define a view `Q7(hop)` that finds the most frequently used hop variety.

Q8 (3 marks)

The most common beer styles nowadays are IPAs, Lagers and Stouts. Define a view `Q8(brewery)` that finds breweries that do not make any of these common beer styles. Find beer styles that include precisely the string "IPA" or "Lager" or "Stout" somewhere in the style name.

Q9 (3 marks)

Define a view `Q9(grain)` that gives the name of the most commonly used grain in Hazy IPA beers. Find beers whose style is exactly the string "Hazy IPA".

Q10 (3 marks)

Define a view `Q10(unused)` that gives the names of any ingredients that are not used in making any of the beers in the database. We are not interested in the type of the ingredient, just its name.

Q11 (4 marks)

Write a PLpgSQL function that takes a country name as argument and returns an `ABVrange` tuple giving the minimum and maximum ABVs of all beers brewed in that country.

```
create or replace function Q11(_country text) returns ABVrange ...
```

The `_country` paramter must *exactly match* the name of the country in the database. If it matches no country, return the tuple (0,0).

The supplied `ass1.sql` contains a definition for the `ABVrange` which should work each time you load `ass1.sql` into the database, although it does print a `NOTICE` (*not* an error) about the dropping operation cascading to the function. Do not change this definition.

In order to make your output consistent with the format used in the examples, cast the final values to `numeric(4,1)`. Let's say that your minimum ABV value ends up in a variable called `min_abv`. Then, in the statement where it ends up in the result tuple, write it as `min_abv::numeric(4,1)`.

There are examples of how the function should behave in the Examples page.

Q12 (6 marks)

Write a PostgreSQL function that takes a string as argument and gets information about all beers that contain that string somewhere in their name (use case-insensitive matching).

```
create or replace function Q12(partial_name text) returns setof BeerData ...
```

The `BeerData` type has three components:

- `beer`: the name of the beer
- `brewer`: the name of the brewery/breweries who make the beer
- `info`: the ingredients used in making the beer

Note that some beers involve two breweries who collaborate in making the beer. These beers should *not* be shown twice, once for each brewer. Instead, the `brewer` column should contain the names of all breweries in alphabetical order, and separated by `' + '`. There are examples of this in the Examples page.

The `info` should presented as a single text string, formatted as up to three lines: one containing a comma-separated list of hops, one containing a comma-separated list of grains, and one containing a comma-separated list of adjuncts. If no information is available about one of these types of ingredients, do not include a line for that type. Do not include a final `'\n'` character in the result string.

An example of what the `info` should look like for a beer that uses all ingredient types:

```
Hops: Bravo,Centennial,Mosaic
Grain: Oats,Pale,Rye,Treticale,Wheat
Extras: Lactose,Vanilla
```

The comma-separated ingredient lists should be in alphabetical order.

Note that `psql` put a `'+'` at the end of each line. Ignore this; it's an output artifact.

There are more examples of how the function should behave in the Examples page. In particular, if there are no beers matching the `partial_name`, simply return an empty table (0 rows).

Submission and Testing

We will test your submission as follows:

- create a testing subdirectory
- create a new database *TestingDB* and initialise it with `ass1.dump`
- load your work via the command: `psql TestingDB -f ass1.sql` (using your `ass1.sql`)
- run auto-marking on your views and functions using a script that we will eventually make available to you

Note that there is a time-limit on the execution of queries. If any query takes longer than 3 seconds to run (you can check this using the **\timing** flag) your mark for that query will be reduced.

Your submitted code must be *complete* so that when we do the above, your `ass1.sql` will load without errors. If your code does not work when installed for testing as described above and the reason for the failure is that your `ass1.sql` did not contain all of the required definitions, you will be penalised by a 2 mark penalty. If you code does not load because your definitions are in the wrong order, there will be a 1 mark penalty.

Before you submit, it would be useful to test out whether the files you submit will work on `d2`, by doing the following:

```
vxdb2$ dropdb ass1
vxdb2$ createdb ass1
vxdb2$ psql ass1 -f ass1.dump
vxdb2$ psql ass1 -f ass1.sql
```

These commands may produce information messages, but they *should not* produce any errors.

Have fun, *jas*