COMP3311 23T3 Database Systems

Assignment 1
Queries and Functions on BeerDB

Last updated: **Sunday 24th September 3:45pm**
Most recent changes are shown in red ... older changes are shown in brown.
[Assignment Spec] [Database Design] [Examples]

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:	23:59 Friday 13 October
Marks:	**12 marks** toward your total mark for this course
Late Penalty:	0.2 *marks* off the ceiling mark for each hour late, for 5 days any submission after 5 days scores 0 marks ... UNSW late penalty policy

How to do this assignment:

- read this specification carefully and completely
- create a directory for this assignment
- copy the supplied files into this directory (see Setting Up)
- login to `vxdb2` 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 `vxdb2`, which is where it's tested
- submit `ass1.sql` via WebCMS or `give`

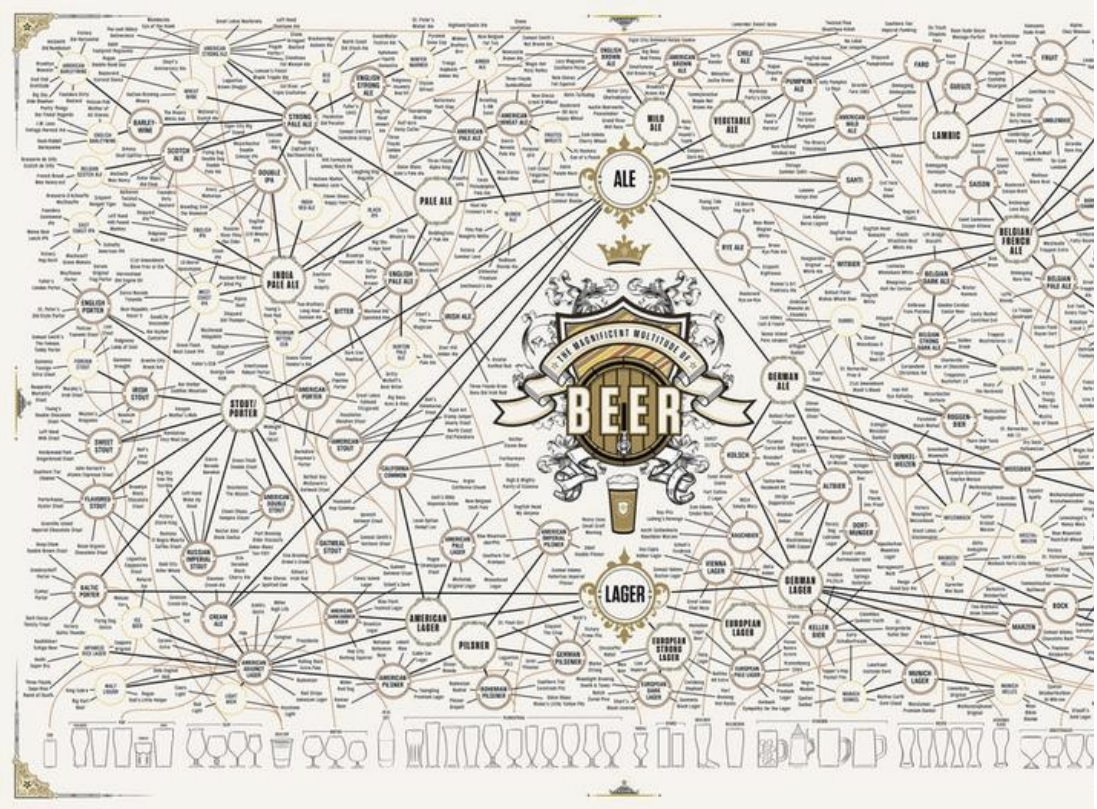
Details of the above steps are given below. Note that you can put the files wherever you like; they do not have to be under your `/localstorage` directory. You also edit your SQL files on hosts other than `vxdb2`. The only time that you need to use `vxdb2` is to manipulate your database. Since you can work at home machine, you don't have to use `vxdb2` 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 `vxdb2`. 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 `vxdb2`, while others can (and probably should) be done on a CSE machine other than `vxdb2`. In the examples below, we'll use `vxdb2` to indicate that the comand must be done on `vxdb2` 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/23T3/assignments/ass1/ass1.sql ass1.
cse$ cp /home/cs3311/web/23T3/assignments/ass1/ass1.dump ass1
```

This gives you a template for the SQL views and SQL 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.

As supplied, the `ass1.sql` template file consists entirely of comments; remove the comments from whatever question you're working on. This allows you to load the `ass1.sql` file without getting syntax errors from the not-yet-completed parts of the template. Of course, don't forget to remove the comments from all completed questions before testing and submitting.

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 vxdb2, 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
... examine the database contents
vxdb$ psql ass1
```

The database loading should take less than 2 seconds on `vxdb2`, and will produce messages like:

```
CREATE DOMAIN
CREATE TYPE
SET
CREATE TABLE
COPY number
ALTER TABLE
```

The `ass1.dump` file contains the schema and data in a single file, along with a simple PLpgSQL function (`dbpop()`), which counts the number of tuples in

each table.

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 not use `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.

Note #4: do not worry about the order of tuples in your result; we will apply `order by` when testing the queries (see the ordering in the Examples page).

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

Q0 (2 marks)

Style mark

Given that you've already 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 alternative, 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 (1 mark)

Write a view `Q1(state,nbreweries)` that returns a list of Australian states and a count of the number of breweries in each state.

The columns in the result are:

- `state` = the name of a state (region)
- `nbreweries` = a count of the number of breweries in that state

Q2 (1 mark)

Write a view `Q2(style,min_abv,max_abv)` that determines which style(s) have the largest difference between their minimum and maximum ABV values (i.e. the widest range of alcohol levels).

The columns in the result are:

- `style` = the name of the style
- `min_abv` = the minimum ABV for the style (`Styles.min_abv`)
- `max_abv` = the maximum ABV for the style (`Styles.max_abv`)

Q3 (2 marks)

Write a view `Q3(style,lo_abv,hi_abv,min_abv,max_abv)` that gives a list of beer styles satisfying the properties:

- the minimum and maximum ABVs for the style are different
- some beer brewed in the style has an ABV outside the min/max range

The columns in the result are:

- `style` = the name of the style
- `lo_abv` = lowest ABV of any beer made in that style
- `hi_abv` = highest ABV of any beer made in that style
- `min_abv` = the minimum ABV for the style (`Styles.min_abv`)
- `max_abv` = the maximum ABV for the style (`Styles.max_abv`)

Q4 (2 marks)

Write a view `Q4(brewery,rating)` that returns the brewery (or breweries) with the maximum average rating for all their beers.

The columns in the result are:

- `brewery` = the name of the brewery
- `rating` = average rating for rated beers by that brewery

To avoid beweries with a single high-rated beer coming out on top, we only consider breweries that have at least five rated beers. If beers are brewed collaboratively, give the rating to both the breweries involved. Make sure that you compute the average rating using floating point numbers; if you use integers, PostgreSQL truncates to an integer like C does. Don't include beers with no rating (i.e. `Beers.rating is NULL`).

Q5 (2 marks)

Alcohol consumption is often measured in term of "standard drinks". This can be calculated by the formula: <

volume × ABV × 0.0008

The number of standard drinks is usually printed on the container, but just in case ...

Write an SQL function `Q5(pattern text)` whose argument is a pattern representing part of a beer name. The function returns a set of 0 or more tuples

for any beers whose name contains the pattern, and with the following fields:

- `beer` = full name of a beer
- `container` = container that the beer is sold in (e.g. `440ml can`)
- `std_drinks` = the number of standard drinks in the container

You must define the function as follows

```
... Q5(pattern text) returns table(beer text, container text,
```

The standard drinks value is of type `numeric(3,1)`. You will need to determine how to construct the `container` string.

Q6 (3 marks)

Write an SQL function `Q6(pattern text)` whose argument is a string representing part of a country name. The function returns a set of 0 or more tuples for any countries whose name contains the pattern (case-insensitive matching), and with the following fields:

- `country` = full name of a country
- `first` = the earliest year that a beer was brewed in that country
- `nbeers` = the number of beers brewed in that country
- `rating` = the average rating of beers brewed in that country

You must define the function as follows

```
... Q6(pattern text) returns
      table(country text, first integer, nbeers integer, rat
```

The `rating` value should be cast to type `numeric(3,1)` within the function.

Q7 (3 marks)

Write a PLpgSQL function `Q7(_beerID integer)` whose argument is an integer representing a beer ID (`Beers.id`) value. The function returns a single text string, formatted as follows:

- if the ID does not exist in the `Beers` table, simply return the string

  ```
  No such beer (_beerID)
  ```

- if the ID does exist, the first line contains the name of the beer enclosed in `"..."`
- if the beer has no ingredients recorded, simply add a second line to the return string and return the two-line string, e.g.

  ```
  "Name of beer"
    no ingredients recorded
  ```

  Note that there are exactly two spaces before the `no ingredients`.
- otherwise append a string containing a list of ingredients, one per line, in order of the ingredient names:
  - each line starts with exactly four spaces
  - then the name of the ingredient
  - then the type of the ingredient in `(...)`

Note that the return value of this function is a single text string. It will likely contain embedded newline characters, but there should be **no** trailing newline character. There are examples of the return values in the Examples page, in case the above is not clear enough.

Note also that, in its output, `psql` uses a plus `+` symbol to indicate that there is an embedded newline. The plus symbol is simply an artifact of the way `psql` displays strings. Do not embed plus symbols into your return string.

Q8 (4 marks)

Write a PLpgSQL function `Q8(pattern text)` whose argument is a string representing part of a beer name. The function returns a set of 0 or more tuples for any beers whose name contains the pattern (case-insensitive matching), and with the following fields:

- `beer` = full name of a beer
- `brewery` = which brewery (or breweries) the beer was made in
- `hops` = a comma-separated list of the names of hops used in the beer

If the beer is a collaboration beer (`brewed_by` more than one brewery), the names of all breweries involved must be included, separated by `'+'` characters, and appear in alphabetical order of the brewery names.

The names of the hops in the list of hops contained in the beer must be separated by `','` characters, and must appear in alphabetical order. If a beer has no hop ingredients recorded, then the hops string should be set to:

```
no hops recorded
```

The function needs to be defined differently to the functions above that returned `table(...)`. For this function, we need to define a new tuple type and have the function return a `setof` tuples of that type:

```
drop type if exists BeerHops cascade;
create type BeerHops as (beer text, brewery text, hops text);

create or replace function
    Q8(pattern text) returns setof BeerHops
...
```

The `drop type` statement is included so that you can reload the `ass1.sql` file multiple times without generating errors. It does, however, generate a `NOTICE`, which looks like an error, but is actually harmless.

Q9 (4 marks)

Write a PLpgSQL function `Q9(breweryID integer)` whose argument is an integer, possibly containing a `Breweries.id` value. The function returns a set of `Collab` tuples (see below) for the brewery, and with the following fields:

- `brewery` = full name of a brewery, or `NULL`
- `collaborator` = full name of another brewery, the string `'none'`

The `Collab` tuples give the names, in alphabetical order, of all the breweries that the indicated brewery (`breweryID`) has made collaboration beers with.

There are a number of different cases for what appears in the `Collab` tuples:

- `breweryID` is not a valid `Breweries.id` value
  - `brewery` has the value `'No such brewery (`*`breweryID`*`)'`
  - `collaborator` has the value `'none'`
- the brewery has never made a collaboration beer
  - `brewery` contains the name of the brewery whose id is `breweryID`
  - `collaborator` has the value `'none'`
- the brewery has collaborated with exactly one other brewery
  - `brewery` contains the name of the brewery whose id is `breweryID`
  - `collaborator` contains the name of the collaborating brewery
- the brewery has collaborated with more than one other brewery

- there are multiple tuples for this brewery
- in the first tuple, `brewery` contains the name of the brewery whose id is `breweryID`
- in all subsequent tuples, `brewery` has the value `NULL`
- in all tuples, `collaborator` contains the name of a collaborating brewery

The function needs to be defined differently to the functions above that returned `table(...)`. For this function, we need to define a new tuple type and have the function return a `setof` tuples of that type:

```
drop type if exists Collab cascade;
create type Collab as (brewery text, collaborator text);

create or replace function
    Q9(_bid integer) returns setof Collab
...
```

The `drop type` statement is included so that you can reload the `ass1.sql` file multiple times without generating errors. It does, however, generate a `NOTICE`, which looks like an error, but is actually harmless.

Note that the function always returns at least one tuple, even if `breweryID` is not valid. Note also that this is different to question Q6, where the list of list of ingredients was formed by string concatentation; in this case, each collaborating brewery is in a separate tuple.

In case the above description is not clear, there are many examples in the Examples page.

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
- drop the *`TestingDB`* database
- create a new database *`TestingDB`* and initialise it with an unseen dump file (same schema, different data)
- load your work via the command: `psql TestingDB -f ass1.sql` (using your `ass1.sql`)
- run auto-marking on your views and functions

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 whether the files you submit will work on `vxdb2`, 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*