COMP3311 20T3 Database Systems

Assignment 2
Queries and Functions on MyMyUNSW

Last updated: **Thursday 29th October 10:36pm**
Most recent changes are shown in red ... older changes are shown in brown.

[Assignment Spec] [Database Design] [Schema in SQL] [check1.sql] [Examples] [Fixes+Updates]

Aims

This assignment aims to give you practice in

- reading and understanding a moderately large relational schema (MyMyUNSW)
- 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 MyMyUNSW database, which contains a wealth of information about what happens at UNSW. 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.

A theme of this assignment is "dirty data". As I was building the database, using a collection of reports from UNSW's information systems and the database for the academic proposal system (MAPPS), I discovered that there were some inconsistencies in parts of the data (e.g. duplicate entries in the table for UNSW buildings, or students who were mentioned in the student data, but had no enrolment records, and, worse, enrolment records with marks and grades for students who did not exist in the student data). I removed most of these problems as I discovered them, but no doubt missed some. Some of the exercises below aim to uncover such anomalies; please explore the database and let me know if you find other anomalies.

Summary

Submission:	Login to Course Web Site > Assignments > Assignment 2 > [Submit] > upload `ass2.sql` or, on a CSE server, `give cs3311 ass2 ass2.sql`
Required Files:	`ass2.sql` (contains both SQL views and PLpgSQL functions)
Deadline:	15:00 Monday 2 November
Marks:	**16 marks** toward your total mark for this course
Late Penalty:	0.1 *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 `grieg` and run your PostgreSQL server
- load the database and start exploring
- complete the tasks below by editing `ass2.sql`
- submit `ass2.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 `/srvr` directory. You also edit your SQL files on hosts other than `grieg`. The only time that you need to use `grieg` is to manipulate your database.

Introduction

All Universities require a significant information infrastructure in order to manage their affairs. This typically involves a large commercial DBMS installation. UNSW's student information system sits behind the MyUNSW web site. MyUNSW provides an interface to a PeopleSoft enterprise management system with an underlying Oracle database. This back-end system (Peoplesoft/Oracle) is sometimes called NSS. The specific version of PeopleSoft that we use is called Campus Solutions.

Despite its successes, however, MyUNSW/NSS still has a number of deficiencies, including:

- no easy way to swap classes once enrolled
- no representation for degree program structures
- poor integration with the UNSW Online Handbook

The first point is inconvenient, since it means that the only way for a student to change tute classes is to drop the course and re-enrol into the course, selecting th new tute. If the course is already full, students would be unwilling to drop the course in case someone else grabs their place before they can re-enrol.

The second point prevents MyUNSW/NSS from being used for three important operations that would be extremely helpful to students in managing their enrolment:

- finding out how far they have progressed through their degree program, and what remains to be completed
- checking what are their enrolment options for next semester (e.g. get a list of "suggested" courses)
- determining when they have completed all of the requirements of their degree program and are eligible to graduate

In this assignment, you will be working with two instances of a database to hold information about academic matters at UNSW. The first database instance contains data from 2000 to 2015. The second database instance will contain data from 2014 to 2019. Note that all `People` data about students, and much of the `People` data about staff is synthetic.

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 Grieg. 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 `grieg`, while others can (and probably should) be done on a CSE machine other than `grieg`. In the examples below, we'll use `grieg$` to indicate that the comand must be done on `grieg` and `cse$` 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/ass2
cse$ cd /my/dir/for/ass2
cse$ cp /home/cs3311/web/20T3/assignments/ass2/ass2.sql ass2.sql
```

Note that the database dump is quite large. It's not worth copying it into your assignment directory on the CSE servers, because you only need to read it once to build your database (see below). 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 Grieg and source env as usual ...
grieg$ dropdb mymy  ... if you already had such a database
grieg$ createdb mymy
grieg$ bzcat /home/cs3311/web/20T3/assignments/ass2/mymy1.dump.bz2 | psql mymy
grieg$ psql mymy
... examine the database contents ...
```

The database loading should take less than 60 seconds on Grieg, assuming that Grieg is not under heavy load. (If you leave your assignment until the last minute, loading the database on Grieg will be considerably slower, thus delaying your work even more. The solution: at least load the database *Right Now*, even if you don't start using it for a while.) (Note that the `mymyunsw.dump` file is 50MB in size; copying it under your home directory or your `srvr/` directory is not a good idea).

If you have other large databases under your PostgreSQL server on Grieg or you have large files under your `/srvr/`*`YOU`*`/` directory, it is possible that you will exhaust your Grieg disk quota. In particular, you will not be able to store two copies of the MyMyUNSW database under your Grieg server. The solution: remove any existing databases before loading your MyMyUNSW database.

If you're running PostgreSQL at home, the file `ass2.zip` contains copies of the files: `mymy1.dump.bz2`, `ass2.sql` to get you started. You can grab the `check1.sql` script separately, once it becomes available. If you copy `ass2.zip` to your home computer, unzip it, and perform commands analogous to the above, you should have a copy of the MyMyUNSW database that you can use at home to do this assignment.

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

One useful query is

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

This will give you a list of tables and the number of tuples in each. Some tables are empty, and are not relevant to this assignment. They are included for the sake of "completeness", i.e. to show what kinds of data might be stored in a real database to replace MyUNSW/NSS.

Your Tasks

Answer each of the following questions by typing SQL or PLpgSQL into the `ass2.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 `ass2.sql` as you want. However, make sure that *everything* that's required to make all of your views and functions work ends up in the `ass2.sql` file before you submit.

Q1 (1 mark)

Define an SQL view `Q1(unswid,name)` that gives the student id and name of any student who has studied more than 65 courses at UNSW. The name should be take from the `People.name` field for the student, and the student id should be taken from `People.unswid`.

Q2 (2 marks)

Define an SQL view `Q2(nstudents,nstaff,nboth)` which produces a table with a single row containing counts of:

- the total number of students (who are not also staff)
- the total number of staff (who are not also students)
- the total number of people who are both staff and student

Q3 (2 marks)

Define an SQL view `Q3(name,ncourses)` that prints the name of the person(s) who has been lecturer-in-charge (LIC) of the most courses at UNSW and the number of courses they have been LIC for. In the database, the LIC has the role of "Course Convenor".

Q4 (2 marks)

For the mymy1 database instance ... Define SQL view `Q4a(id,name)` which gives the student IDs and names of all students enrolled in in the old Computer Science (3978) degree in 05s2.

For the mymy2 database instance ... Define an SQL view `Q4b(id,name)` which gives the student IDs and names of all students enrolled in in the Computer Science (3778) degree in 17s1.

Note: the student IDs are the UNSW id's (i.e. student numbers) defined in the `People.unswid` field.

Each of the views will return an empty result in the database for which it was not designed.

Q5 (3 marks)

Define an SQL view `Q5(name)` which gives the faculty with the maximum number of committees. Include in the count for each faculty, both faculty-level committees and also the committees under the schools within the faculty. You can use the `facultyOf()` function to help with this (the function is already available in the database). You can assume that committees are defined only at the faculty and school level.

Use the `OrgUnits.name` field as the faculty name.

Q6 (1 mark)

Define an SQL function (SQL, *not* PLpgSQL) called `Q6(id integer)` that takes as parameter either a `People.id` value (i.e. an internal database identifier) or a `People.unswid` value (i.e. a UNSW student ID), and returns the name of that person. If the id value is invalid, return NULL as the result. You can assume that `People.id` and `People.unswid` values come from disjoint sets, so you should never return multiple names. (It turns out that the sets aren't quite disjoint, but I won't test any of the cases where the id/unswid sets overlap)

The function must use the following interface:

```
create or replace function Q6(id integer) returns text ...
```

Q7 (2 marks)

Define an SQL function (*not* PLpgSQL) called `Q7(subject text)` that takes as parameter a UNSW subject code (e.g. 'COMP1917') and returns a list of all offerings of the subject for which a Course Convenor is known.

Note that this means just the Course Convenor role, not Course Lecturer or any other role associated with the course. Also, if there happen to be several Course Convenors, they should all be returned (in separate tuples). If there is no Course Convenor registered for a particular offering of the course, then that course offering should not appear in the result.

The function must use the following interface:

```
create or replace function Q7(subject text)
  returns table (subject text, term text, convenor text)
```

The `course` field in the result tuples should be the UNSW course code (i.e. that same thing that was used as the parameter). If the parameter does not correspond to a known UNSW course, then simply return an empty result set.

Q8 (5 marks)

Define a PLpgSQL function `Q8(zid integer)`, which takes a student id and produces a transcript as a table of `TranscriptRecords`. Each transcript record should contain information about the student's attempt at a course. Records should appear ordered by term; within a term, they should be ordered by subject code.

Use the following definition for the transcript tuples:

```
create type TranscriptRecord as (
        code  char(8), -- UNSW-style course code (e.g. COMP1021)
```

```
        term  char(4), -- semester code (e.g. 98s1)
        prog  char(4), -- program being studied in this semester
        name  text,    -- shortened name of the course's subject
        mark  integer, -- numeric mark acheived
        grade char(2), -- grade code (e.g. FL,UF,PS,CR,DN,HD)
        uoc   integer  -- units of credit available for the course
);
```

Note that this type is already defined in the database.

All subject names should be truncated to 20 characters (hint: use PostgreSQL's `substr()` function).

If no `mark` or `grade` is available for a course, set those fields as null in the `TranscriptRecord`. Only display a UOC value if the student actually passed the course (i.e. has a grade from the set {SY, PT, PC, PS, CR, DN, HD, A, B, C}) or has an XE grade (for credit from exchange) or a T grade (transferred credit) or a PE grade (professional experience) or a RC or RS grade (research courses)). Otherwise, leave the `uoc` field as null. A null grade or any grade other than those just mentioned should not be treated as a pass.

At the end of the transcript, add an extra `TranscriptRecord` which contains

```
(null, null, null, 'Overall WAM/UOC', wamValue, null, UOCpassed)
```

where the *`wamValue`* and *`UOCpassed`* are computed as follows:

```
wamValue = weightedSumOfMarks / totalUOCattempted
UOCpassed = sum of UOC for all subjects passed
totalUOCattempted = sum of UOC for all subjects in transcript
weightedSumOfMarks = sum of mark*UOC for all subjects in transcript
```

Do not include in the WAM calculations any course that has a null grade. If a course ~~has a null mark but~~ has a SY or XE T or PE grade, include the UOC in the UOCpassed only. Round the WAM value to the nearest integer.

If no courses have been completed, add the following tuple at the end of the transcript:

```
(null, null, null, 'No WAM available', null, null, null)
```

If a student switches degrees part-way through their study, just treat the transcript as all being related to a single program. Even if they credit transferred from the first to the second program (entries with a T grade) just count them all towards the total UOC.

How can you find interesting transcripts to look at? The answer is to think of some properties of a transcript that might make it interesting, and then ask a query to get information about any students who have these kinds of transcripts. I used the following query to find some students. Work out what it does and then try variations to find other kinds of "interesting" students:

```
select p.unswid,pr.code,termName(min(pe.term)),count(*)
from   People p
         join Program_enrolments pe on (pe.student=p.id)
         join Programs pr on (pe.program=pr.id)
where  pr.code like '3%'
group  by p.unswid,pr.code
having count(*) > 5;
```

Q9 (5 marks)

An important part of defining academic rules in MyMyUNSW is the ability to define groups of academic objects (e.g. groups of subjects, streams or programs) In MyMyUNSW, groups can be defined in three different ways:

- `enumerated` by giving a list of objects in a *`X`*`_members` table

- `pattern` by giving a pattern that identifies all relevant objects
- `query` by storing an SQL query which returns a set of object ids

In all cases, the result is a set of academic objects of a specific type (given in the `gtype` attribute).

Write a PLpgSQL function `Q8(gid integer)` that takes the internal ID of an academic object group and returns the codes for all members of the academic object group, including any child groups. Associated with each code should be the type of the corresponding object, either `subject`, `stream` or `program`.

You should return distinct codes (i.e. ignore multiple versions of any object), and there is no need to check whether the academic object is still being offered.

The function is defined as follows:

```
create or replace function Q9(gid integer) returns setof AcObjRecord
```

where `AcObjRecord` is already defined in the database as follows:

```
create type AcObjRecord as (
        objtype text,  -- academic object's type e.g. subject, stream, program
        objcode text   -- academic object's code e.g. COMP3311, SENGA1, 3978
);
```

Groups of academic objects are defined in the tables:

- `acad_object_groups(id, name, gtype, glogic, gdefby, negated, parent, definition)`
  where the most important fields are:
  - `gtype` ... what kind of objects in the group
  - `gdefby` ... how the group is defined
  - `definition` ... where queries or patterns are given
- `program_group_members(program, ao_group)` ... for enumerated program groups
- `stream_group_members(stream, ao_group)` ... for enumerated stream groups
- `subject_group_members(subject, ao_group)` ... for enumerated subject groups

There are a wide variety of patterns. You should explore the `acad_object_groups` table yourself to see what's available. To give you a head start, here are some existing patterns and what they mean:

- `COMP2###` ... any level 2 computing course (e.g. COMP2911, COMP2041)
- `COMP[34]###` ... any level 3 or 4 computing course (e.g. COMP3311, COMP4181)
- `####1###` ... any level 1 course at UNSW
- `(COMP|SENG|BINF)2###` ... any level 2 course from CSE
- `COMP1917,COMP1927` ... core first year computing courses
- `COMP1###,COMP2###` ... same as `COMP[12]###`

You do *not* need to handle any of the following types of academic object groups:

- any groups defined using a query (`gdefby='query'`)
- any groups defined using negation (`negated=true`)
- any groups defined by a pattern which includes `'FREE'`, `'GEN'` or `'F='` as a substring

For any group like the above, simply return no reults (zero rows). You can also ignore the `glogic` field; treat them all as `or` groups.

If any group has a child group containing FREE, ignore just the child group. For pattern groups, you do not need to check whether codes used in the pattern correspond to real objects in the relevant table (e.g. a pattern string may contain a subject code which does not exist in the `Subjects` table)

Your function should be able to expand any pattern element from the above classes of patterns (i.e. pattern elements that include #, `[...]` and `(...|...)`). If patterns include `{xxx;yyy;zzz}` alternatives, include all of the alternatives as group members (i.e. as if they were `xxx,yyy,zzz`). If patterns have

child patterns, include all of the acdemic objects in the child patterns. You can recognise that a group with `id=X` has child groups, by the existnce of other academic groups with `parent=X`.

Hint: In order to solve this, you'll probably need to look in the PostgreSQL manual at Section 9.4 "String Functions and Operators" and Section 42.5.4 "Executing Dynamic Commands".

Q10 (4 marks)

Define a PLpgSQL function that takes a subject code and returns the set of all subjects that include this subject in their pre-reqs. This kind of function could help in planning what courses you could study in subsequent semesters.

The function is defined as follows:

```
create or replace function Q10(code text) returns setof text ...
```

You only need to consider literal subject codes (e.g. COMP1234) in the pre-reqs. If a pre-req object group contains a pattern, ignore the pattern.

Hint: This function can probably make use of (a variation of) the `Q9()` function.

Submission and Testing

We will test your submission as follows:

- create a testing subdirectory
- create a new database *`TestingDB`* and initialise it with `mymy1.dump`
- run the command: `psql` *`TestingDB`* `-f ass2.sql` (using your `ass2.sql`)
- load and run the tests in the `check1.sql` script
- repeat the above for `mymy2.dump` and `check2.sql`

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 wiil be reduced.

Your submitted code must be *complete* so that when we do the above, your `ass2.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 `ass2.sql` did not contain all of the required definitions, you will be penalised by a 3 mark administrative penalty.

Before you submit, it would be useful to test out whether the files you submit will work by following a similar sequence of steps to those noted above.

Have fun, *jas*