Project 1 SQL and PLpgSQL

Due: Sun 27 Sept, 23:59

1. Aims

This project 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. A theme of this project 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.

2. How to do this project:

- read this specification carefully and completely
- familiarise yourself with the database schema (description, SQL schema, summary)
- make a private directory for this project, and put a copy of the proj1.sql template there
- you **must** use the create statements in proj1.sql when defining your solutions
- look at the expected outputs in the expected_qX tables loaded as part of the check.sql file
- solve each of the problems below, and put your completed solutions into proj1.sql
- check that your solution is correct by verifying against the example outputs and by using the check_qX() functions
- test that your proj1.sql file will load *without error* into a database containing just the original MyMyUNSW data
- double-check that your proj1.sql file loads in a *single pass* into a database containing just the original MyMyUNSW data
- submit the project via give.

3. 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 often called NSS.

UNSW has spent a considerable amount of money ($80M+) on the MyUNSW/NSS system, and it handles much of the educational administration plausibly well. Most people gripe about the quality of the MyUNSW interface, but the system does allow you to carry out most basic enrolment tasks online.

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

- no waiting lists for course or class enrolment
- no representation for degree program structures
- poor integration with the UNSW Online Handbook

The first point is inconvenient, since it means that enrolment into a full course or class becomes a sequence of trial-and-error attempts, hoping that somebody has dropped out just before you attempt to enrol and that no-one else has grabbed the available spot.

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

NSS contains data about student, courses, classes, pre-requisites, quotas, etc. but does not contain any representation of UNSW's degree program structures. Without such information in the NSS database, it is not possible to do any of the above three. So, in 2007 the COMP9311 class devised a data model that could represent program requirements and rules for UNSW degrees. This was built on top of an existing schema that represented all of the core NSS data (students, staff, courses, classes, etc.). The enhanced data model was named the MyMyUNSW schema.

The MyMyUNSW database includes information that encompasses the functionality of NSS, the UNSW Online Handbook, and the CATS (room allocation) database. The MyMyUNSW data model, schema and database are described in a separate document.

4. Setting Up

To install the MyMyUNSW database under your Grieg server, simply run the following two commands:

$ **createdb proj1**
$ **psql proj1 –f /home/cs9311/web/15s2/proj/proj1/mymyunsw.dump**

If you've already set up PLpgSQL in your template1 database, you will get one error message as the database starts to load:

psql:mymyunsw.dump:*NN*: ERROR: language "plpgsql" already exists

You can ignore this error message, but any other occurrence of ERROR during the load needs to be investigated.

If everything proceeds correctly, the load output should look something like:

```
SET
SET
SET
SET
SET
psql:mymyunsw.dump:NN: ERROR:  language "plpgsql" already exists
... if PLpgSQL is not already defined,
... the above ERROR will be replaced by CREATE LANGUAGE
SET
SET
SET
CREATE TABLE
CREATE TABLE
... a whole bunch of these
CREATE TABLE
ALTER TABLE
ALTER TABLE
... a whole bunch of these
ALTER TABLE
```

Apart from possible messages relating to plpgsql, you should get no error messages. The database loading should take less than 60 seconds on Grieg, assuming that Grieg is not under heavy load. (If you leave your project 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 proj1.tar.gz contains copies of the files: mymyunsw.dump, proj1.sql to get you started. You can grab the check.sql separately, once it becomes available. If you copy proj1.tar.gz to your home computer, extract 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 project.

A useful thing to do initially is to get a feeling for what data is actually there. This may help you understand the schema better, and will make the descriptions of the exercises easier to understand. Look at the schema. Ask some queries. Do it now.

Examples ...

$ **psql proj1**

... PostgreSQL welcome stuff ...
proj1=# **\d**
... look at the schema ...
proj1=# **select * from Students;**
... look at the Students table ...
proj1=# **select p.unswid,p.name from People p join Students s on (p.id=s.id);**
... look at the names and UNSW ids of all students ...
proj1=# **select p.unswid,p.name,s.phone from People p join Staff s on (p.id=s.id);**
... look at the names, staff ids, and phone #s of all staff ...
proj1=# **select count(*) from CourseEnrolments;**
... how many course enrolment records ...
proj1=# **select * from dbpop();**
... how many records in all tables ...
proj1=# **select * from transcript(3197893);**
... transcript for student with ID 3197893 ...
proj1=# ... etc. etc. etc.
proj1=# **\q**

You will find that some tables (e.g. Books, Requirements, etc.) are currently unpopulated; their contents are not needed for this project. You will also find that there are a number of views and functions defined in the database (e.g. dbpop() and transcript() from above), which may or may not be useful in this project.

Summary on Getting Started

To set up your database for this project, run the following commands in the order supplied:
$ **createdb proj1**
$ **psql proj1 -f /home/cs9311/web/15s2/proj/proj1/mymyunsw.dump**
$ **psql proj1**
... run some checks to make sure the database is ok
$ **mkdir *Project1Directory***
... make a working directory for Project 1
$ **cp /home/cs9311/web/15s2/proj/proj1/proj1.sql *Project1Directory***

The only error messages produced by these commands should be those noted above. If you omit any of the steps, then things will not work as planned.

Notes

Read these before you start on the exercises:

- the marks reflect the relative difficulty/length of each question
- use the supplied proj1.sql template file for your work
- you may define as many additional functions and views as you need, provided that (a) the definitions in proj1.sql are preserved, (b) you follow the requirements in each question on what you are allowed to define
- make sure that your queries would work on any instance of the MyMyUNSW schema; don't customise them to work just on this database; we may test them on a different database instance

- do not assume that any query will return just a single result; even if it phrased as "most" or "biggest", there may be two or more equally "big" instances in the database
- you are not allowed to use ***limit*** in answering any of the queries in this project
- when queries ask for people's names, use the Person.name field; it's there precisely to produce displayable names
- when queries ask for student ID, use the `People.unswid` field; the `People.id` field is an internal numeric key and of no interest to anyone outside the database
- unless specifically mentioned in the exercise, the order of tuples in the result does not matter; it can always be adjusted using `order by`. In fact, our check.sql will order your results automatically for comparison.
- the precise formatting of fields within a result tuple **does** matter; e.g. if you convert a number to a string using to_char it may no longer match a numeric field containing the same value, even though the two fields may look similar
- develop queries in stages; make sure that any sub-queries or sub-joins that you're using actually work correctly before using them in the query for the final view/function

An important note related to marking:

- make sure that your queries are reasonably efficient (defined as taking < 15 seconds to run)
- use psql's \timing feature to check how long your queries are taking; they must each take less than 10000 ms
- queries that are too slow will be **penalised by half of the mark for that question**, even if they give the correct result

Each question is presented with a brief description of what's required. If you want the full details of the expected output, take a look at the expected_qX tables supplied in the checking script.

5. Exercises

Q1(3 marks)

Answer this question using only *views*. You *may not* make use of any functions (either SQL ones or PLpgSQL ones). You must define at least a view called `Q1`; you may define auxiliary views if they make it easier to solve the problem.

Some UNSW students have very long and unique names, which always makes for some difficulty for the Dean when he has to read them out at graduation ceremonies. Let's try to help him out by finding the longest unusual family name of any UNSW student (if he knows the longest name, then anything else will seem relatively short). For this exercise we will consider just family names of students, and will ignore family names containing hyphens and spaces (since these are typically made up of short easy words concatenated). Also, since we're after unusual names, we want just family names that occur only once.

Devise some SQL that gives the longest unique student family name (from the `People.family` field for the student). Package the "top-level" SQL query in a view called `Q1` defined as follows:

```
create or replace view Q1(familyName) as ...
```

Hint: Write a view to give family names of students that occur only once.

Sample Results:

```
# select * from Q1;
     familyname
--------------------
Kunnawuttipreechachan
Chonbodeechalermroong
(2 rows)
```

Q2 (3 marks)

Answer this question using only *views*. You *may not* make use of any functions (either SQL ones or PLpgSQL ones). You must define at least a view called `Q2`; you may define auxiliary views if they make it easier to solve the problem.

Some courses at UNSW, contrary to what I was expecting, use A, B, C grades. The original schema only allowed the well-known two-letter grades (e.g. HD, DN, CR, etc.), so it was modified to accept these single-letter grades as well.

Devise some SQL that gives a list of the courses that use A, B, C grades. For each course, give its subject code (e.g. `COMP3311`) and the semester when it was offered, in the format `11S1`. Package the "top-level" SQL query in a view called `Q2` defined as follows:

```
create or replace view Q2(subject,semester) as ...
```

Note that you cannot assume that every such course will use all of the single-letter grades in a given semester (e.g. maybe they had a tough exam and nobody scored an A). Note also that the database does not store an indication of the kind of grading being used in a given course, so the only way you can work it out is to look at the actual grades awarded.

Hint: you will need to use some of PostgreSQL's functions on strings to build the semester name.

Sample Results:

```
# select * from Q2 order by semester,subject;
 subject  | semester
----------+----------
MNGT5201 | 09S1
MNGT5211 | 09S1
MNGT5232 | 09S1
... others omitted to save space ...
MNGT5388 | 10X1
MNGT5395 | 10X1
MNGT5521 | 10X1
(25 rows)
```

Q3 (4 marks)

Answer this question using only *views*. You may not make use of any functions (either SQL ones or PLpgSQL ones). You must define at least a view called `Q3`; you may define auxiliary views if they make it easier to solve the problem.

In the lecture last week, it was claimed that having both `uoc` and `eftsload` in the `Subjects` table was unnecessary, since one could be determined from the other (i.e. `uoc/48 == eftsload`). We could test this calculating the ratio of `uoc/eftsload` for all subjects to see whether they adhere to the `uoc/eftsload` ratio being constant. Since the `eftsload` values in my data files are truncated to three decimal places, some of the ratios won't be exact; to allow for this, take only one decimal place in your ratio calculations.

Devise some SQL to produce a list of distinct `uoc/eftsload` values and the number of subjects which has each value. Package the "top-level" SQL query in a view called `Q3` defined as follows:

```
create or replace view Q3(ratio,nsubjects) as ...
```

Hint: use `numeric(4,1)` to restrict the precision of the ratios, and be careful about typecasting between integers and reals. You can ignore any subjects with `eftsload` values that are either `NULL` or zero.

Sample Results:

```
# select * from Q3 order by ratio;
 ratio | nsubjects
-------+-----------
```

```
 18.5 |         1
 20.0 |         2
 21.3 |         1
... others omitted to save space ...
 48.0 |      9200
 50.3 |         2
 80.0 |         1
(10 rows)
```

Q4 (6 marks)

Answer this question using only *views*. You *may not* make use of any functions (either SQL ones or PLpgSQL ones). You must define at least a view called `Q4`; you may define auxiliary views if they make it easier to solve the problem.

In loading the organisational units (`OrgUnits`), I expected that all of them would be associated with a parent OrgUnit (even the faculties have a parent OrgUnit which is UNSW). However, some OrgUnits appear to be "orphaned"; they are not registered as having any parent in the `UnitGroups` table.

Devise some SQL that gives give the name (`OrgUnits.longname`) of all orphaned organisational units. Package the "top-level" SQL query in a view called `Q4` defined as follows:

```
create or replace view Q4(orgunit) as ...
```

Note that there may be some units which had been stored more than once in the system due to data inconsistency. You could use distinct to remove the duplicates in the final results.

Sample Results:

```
# select * from Q4 order by orgunit;
                      orgunit
-------------------------------------------------
Aboriginal Research and Resource Centre
Applied Science Program
Asia Pacific Health Res Centre
 ... others omitted to save space ...
UNSW Key Centre for Mines
Unisearch
Violent Studies Unit
(110 rows)
```

Q5 (6 marks)

Answer this question using only *views*. You *may not* make use of any functions (either SQL ones or PLpgSQL ones). You must define at least a view called `Q5`; you may define auxiliary views if they make it easier to solve the problem.

Devise some SQL to produce a list of COMP subjects that were offered in 2008 but were not offered in 2009 or 2010. The assumption here is that if they weren't offered in either of those years (2009/2010), then they're probably no longer offered. Package the "top-level" SQL query in a view called `Q5` defined as follows:

```
create or replace view Q5(code,title) as ...
```

Note that the `lastoffer` field in the `Subjects` table is not reliable, so you *should not* use it in answering this question. Also, please use `subjects.longname` as title.

Sample Results:

```
# select * from Q5 order by code;
   code   |              title
----------+--------------------------------
```

```
COMP2091 | Computing 2
COMP3241 | Real -Time Systems: Specification , Design & Imple
COMP4211 | Advanced Architectures and Algorithms
 ... others omitted to save space ...
COMP9314 | Next Generation Database Systems
COMP9416 | Knowledge Based Systems
COMP9912 | Project (24 Units of Credit)
(9 rows)
```

Q6 (8 marks)

Answer this question using only *views* and plpgsql functions. You must define at least a plpgsql function called `Q6`; you may define auxiliary views and plpgsql functions if they make it easier to solve the problem.

Every course in the MyMyUNSW database is rated by students. Associated with each enrolment record is a field called `stuEval` which allows the student to record an evaluation for the course on a 1...6 scale (1 being "poor" and 6 being "excellent"). The overall rating of a course is determined by taking the mean of all non-null evaluations. In order that ratings be statistically significant, only courses with more than 10 students and where at least 1/3 of the students fill out the evaluation should be considered.

Write a plpgsql function to find the best-rated course(s) in a given semester. Use the following type definition and function header:

```
create type EvalRecord as (code text, title text, rating numeric(4,2));
create or replace function Q6(integer,text) returns setof EvalRecord ...
```

The first parameter is a year value (e.g. `2005`), and the second parameter is a session name (e.g. `'S1'`). The function returns a set of `EvalRecord` for the best-rated course(s) in the given semester. Each row contains a `Subjects.code` value (as `code`), a `Subjects.name` value (as `title`), and a numeric value giving the average of all the `stueval` values for students enrolled in the course. (Note that we return a set because it's possible that several courses might be equally highly rated). If invalid values are supplied for the parameters, the function should simply return an empty set of results. It is also possible to return an empty set if there are no courses that are given a rating in a particular session.

Hints: The SQL `avg` aggregate conveniently ignores `NULL` values. If *nS* is the number of students enrolled in a course and *nE* is the number of evaluations filled out, then we are only interested in courses satisfying: *nS* > 10 AND (3**nE*) > *nS*. If you don't use precisely this expression, you may get different answers to the expected ones, even if overall your logic is correct. Also, you should avoid reducing precision in your rating values until as late as possible in the computation.

Sample Results:

```
# select * from Q6(2007,'S2');
   code   |            title              | rating
----------+-------------------------------+--------
COMP9311 | Database Systems               |   4.27
COMP3331 | Computer Networks&Applications |   4.27
(2 rows)
```

6. Submission

Submit this project by doing the following:

- Ensure that you are in the directory containing the file to be submitted.

- Type “give cs9311 proj1 proj1.sql”

The proj1.sql file should contain answers to all of the exercises for this project. It should be completely self-contained and able to load in a single pass, so that it can be auto-tested as follows:

- a fresh copy of the MyMyUNSW database will be created (using the schema from mymyunsw.dump)
- the data in this database may be **different** to the database that you're using for testing
- a new check.sql file will be loaded (with expected results appropriate for the database)
- the contents of your proj1.sql file will be loaded
- each checking function will be executed and the results recorded

Before you submit your solution, you should check that it will load correctly for testing by using something like the following operations:

$ **dropdb proj1** ... remove any existing DB
$ **createdb proj1** ... create an empty database
$ **psql proj1 -f /home/cs9311/web/15s2/proj/proj1/mymyunsw.dump** ... load the MyMyUNSW schema and data
$ **psql proj1 -f /home/cs9311/web/15s2/proj/proj1/check.sql** ... load the checking code
$ **psql proj1 -f proj1.sql** ... load your solution

Note: if your database contains any views or functions that are not available in a file somewhere, you should put them into a file before you drop the database.

If your code does not load without errors, fix it and repeat the above until it does.

You must ensure that your proj1.sql file will load correctly (i.e. it has no syntax errors and it contains all of your view definitions in the correct order). If I need to manually fix problems with your proj1.sql file in order to test it (e.g. change the order of some definitions), you will be fined via a 2 mark penalty for each problem.

7. Late Submission Penalty

10% reduction for the 1st day, then 30% reduction.