Project 2 PLpgSQL

Due: Sun 18 Oct, 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 proj2.sql template there
- you **must** use the create statements in proj2.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 proj2.sql
- check that your solution is correct by verifying against the example outputs and by using the check_qX() functions
- test that your proj2.sql file will load *without error* into a database containing just the original MyMyUNSW data
- double-check that your proj2.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 proj2
$ psql proj2 –f /home/cs9311/web/15s2/proj/proj2/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 proj2.tar.gz contains copies of the files: mymyunsw.dump, proj2.sql to get you started. You can grab the check.sql separately, once it becomes available. If you copy proj2.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 proj2**
... PostgreSQL welcome stuff ...
proj2=# **\d**
... look at the schema ...
proj2=# **select * from Students;**
... look at the Students table ...
proj2=# **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 ...
proj2=# **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 ...
proj2=# **select count(*) from CourseEnrolments;**
... how many course enrolment records ...
proj2=# **select * from dbpop();**
... how many records in all tables ...
proj2=# **select * from transcript(3197893);**
... transcript for student with ID 3197893 ...
proj2=# ... etc. etc. etc.
proj2=# **\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 proj2**
$ **psql proj2 -f /home/cs9311/web/15s2/proj/proj2/mymyunsw.dump**
$ **psql proj2**
... run some checks to make sure the database is ok
$ **mkdir *Project2Directory***
... make a working directory for Project 2
$ **cp /home/cs9311/web/15s2/proj/proj2/proj2.sql *Project2Directory***

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 proj2.sql template file for your work
- you may define as many additional functions and views as you need, provided that (a) the definitions in proj2.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 (13 marks)

You may use any combination of views, SQL functions and PLpgSQL functions in answering this question. However, you must define at least a PLpgSQL function called Q1.

UNSW staff members hold different roles at different periods of time. This information is recorded in the `Affiliation` table. Find staff who have had at least two non-concurrent roles (i.e. they must have two roles where role1 end date < role 2 start date) and display the role history of each staff member.

Write a PLpgSQL function to (a) find staff members who have had several roles over time and (b) produce a listing showing, for each staff member, their `unswid`, `name`, and the `roles` they've had. Note that the roles should be ordered by their starting date, and should be displayed as a single string with the details for each role on a separate line (terminated by `'\n'`). Dates should be presented in the format shown below. Use the following type definition and function header:

```
CREATE TYPE EmploymentRecord AS (unswid INTEGER, name TEXT, roles TEXT);
CREATE OR REPLACE FUNCTION Q1() RETURNS SETOF EmploymentRecord  ...
```

Hints: beware of ongoing roles.

Note that, in its output, PostgreSQL uses a + character to indicate an end-of-line (as well as printing `'\n'`). Note also that the staff should be printed in order of their `People.sortname.`

Sample Results:

```
# SELECT * FROM Q1();
```

unswid	name	roles
8750155	Paul Compton	Professor (1990-02-26 .. 2010-06-30) + Head of School (1996-01-01 .. 1999-06-30) + Head of School (2004-01-01 .. 2010-06-30) +
2114572	Hye-Young Paik	Lecturer (2000-01-06 .. 2003-12-20) + Lecturer (2005-07-01 .. 2010-09-30) + Senior Lecturer (2010-10-01 ..) +
…	…	…

Q2 (13 marks)

You may use any combination of views, SQL functions and PLpgSQL functions in answering this question. However, you must define at least a PLpgSQL function called Q8.

Some of the data in the database comes with annoying trailing spaces. These can lead to confusion when you search for e.g. `'EE-225'` only to discover that it's stored in the database as `'EE-225 '`). It would be useful to track down all of the examples of this, even though they may be spread over many tables and columns.

Write a PLpgSQL function that displays all instances with trailing spaces across all the fields in a given table in the database. Each tuple in the results should show (a) which table is involved, (b) which column in the table, and (c) the number of instances of values with trailing spaces in this column. You should only display result tuples for tables where the number of instances is greater than zero. Use the following type definition and function header:

```
CREATE TYPE TrailingSpaceRecord AS ("table" TEXT, "column" TEXT, nexamples
INTEGER);
CREATE OR REPLACE FUNCTION Q2(tableName TEXT) RETURNS SETOF
TrailingSpaceRecord  ...
```

Hints: You will learn about the PostgreSQL `catalog` and PLpgSQL's `EXECUTE` operation. To check whether a string has trailing spaces, simply check that it's last character is a space.

Sample Results:

```
# SELECT * FROM Q2('rooms');
```

table	column	nexmaples
rooms	longname	1
(1 row)		

Q3 (14 marks)

The transcript function supplied in the database assumes that the only way that a student can get credit towards their degree is by enrolling in a subject for which they have the pre-reqs and passing that subject. In fact, students can obtain various other kinds of "credit" towards their study to help them finish their degree:

- "advanced standing" gives students credit for some course at UNSW based on a similar course completed at another institution (or in an incomplete degree at UNSW); the student is allocated UOC for the UNSW subject, but it does not count towards their WAM; however, for purposes such as prerequisites, it is as if the student took the UNSW course
- "substitution" allows a student to take one subject in place of a core subject in their program (e.g. if the original core subject is not available and it is the student's final semester of study); the student is given the UOC for the course actually taken and the course taken counts in their WAM; however, the course taken may be used as a "stand-in" for the substituted course in determining whether they have met their degree requirements
- "exemption" is where a student is deemed to have completed a course at UNSW based on a similar course at another institution, but is not awarded any UOC for the UNSW course; however, they can use it as a pre-requisite for further study at UNSW

Information about such enrolment variations is stored in the two tables:

```
Variations(student, program, subject, vtype, intequiv, extequiv, ...)
ExternalSubjects(id, extsubj, institution, yearoffered, equivto)
```

where each Variations tuple shows a variation for one student for a given subject towards a particular program. The vtype field indicates what kind of variation it is (`'advstanding', 'substitution', 'exemption'`). The intequiv field references a UNSW subject if the variation is based on a UNSW subject. The extequiv references a tuple in the ExternalSubjects table if the variation is based on a subject studied at another institution. Only one of intequiv or extequiv will be "not null". You should examine the contents of these two tables, as well as the file called "variations.sql" containing details of some of the variations in the database.

A transcript function has already been loaded into the database, along with a definition of the TranscriptRecord type. You can grab a copy of the `transcript()` function definition using PostgreSQL's \ef command (see the PostgreSQL manual for details).

```
CREATE TYPE TranscriptRecord AS (
code CHAR(8),  -- e.g. 'COMP3311'
term CHAR(4),  -- e.g. '12s1'
name TEXT,     -- e.g. 'Database Systems'
mark INTEGER,  -- e.g. 75
grade CHAR(2), -- e.g. 'DN'
uoc iNTEGER    -- e.g. 6
);

CREATE FUNCTION
TRANSCRIPT(_sid INTEGER) RETURNS SETOF TranscriptRecord
AS $$
... PLpgSQL code ...
$$ language plpgsql;
```

You should write a new version of the `transcript()` function called `Q3()` which includes variations as well as regular course enrolments. You can use any or all of the code from the supplied `transcript()` function in developing your Q9() function. Any variations are displayed at the end of the transcript, after the regular courses, but before the WAM calculation. It should still produce the WAM and UOC count, like the original transcript function did, but they will be computed slightly

differently (see below). Note that the Q3() function has exactly the same type signature as that noted above for the `transcript()` function.

Each variation produces two TranscriptRecord tuples. The first tuple gives details of which UNSW subject is being "varied", while the second tuple gives details of the equivalent subject that is used as the basis for the variation.

The details for what first TranscriptRecord tuple contains, for the different types of variation is as follows:

advanced standing

```
first tuple: (CourseCode, null, 'Advanced standing, based on ...', null,
null, UOC)
```

- CourseCode is the course code of the UNSW course for which advanced standing is being granted
- UOC is the UOC for that particular course

Note that the UOC value should be added into the total UOC displayed in the last TranscriptRecord tuple, but should not be included in the WAM calculation.

substitution

```
first tuple: (CourseCode, null, 'Substitution, based on ...', null, null,
null)
```

- CourseCode is the course code of the UNSW course for which advanced standing is being granted

exemption

```
first tuple: (CourseCode, null, 'Exemption, based on ...', null, null, null)
```

- CourseCode is the course code of the UNSW course for which advanced standing is being granted

The second TranscriptRecord tuple for each variation will be different depending on whether the substitution is based on an internal (UNSW) or external subject (from another institution).

variation based on a UNSW subject (intequiv)

```
second tuple: (null, null, 'studying CourseCode at UNSW', null, null, null)
```

- CourseCode is code of the UNSW subject referenced by Variations.intequiv.

variation based on an external subject (extequiv)

```
second tuple: (null, null, 'study at Institution', null, null, null)
```

- Institution is name of the institution in the ExternalSubjects tuple referenced by Variations.extequiv

For the purposes of this exercise, we'll ignore whatever programs the student was enrolled in when they completed the courses. Assume that all courses were part of a single program.

Only a few students in the database have variations. Some example student IDs for testing: 3169329, 3118617, 3270322. You can find more in the "variations.sql" file.

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 proj2 proj2.sql”

The proj2.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 proj2.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 proj2** ... remove any existing DB
$ **createdb proj2** ... create an empty database
$ **psql proj2 -f /home/cs9311/web/15s2/proj/proj2/mymyunsw.dump** ... load the MyMyUNSW schema and data
$ **psql proj2 -f /home/cs9311/web/15s2/proj/proj2/check.sql** ... load the checking code
$ **psql proj2 -f proj2.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 proj2.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 proj2.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.