Project 2 PLpgSQL

<u>Due: Fri 20 May 23:59</u>

Section 1

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

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 COMP3311 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/cs3311/web/16s1/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 Course_Enrolments;
... how many course enrolment records ...
proj2=# select * from dbpop();
... how many records in all tables ...
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() 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/cs3311/web/16s1/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/cs3311/web/16s1/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 < 60 seconds to run)
- use psql's \timing feature to check how long your queries are taking; they must each take less than 60000 ms
- queries that are too slow will be **penalized 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

<u>Note that the mymyunsw.dump used in project 2 is different from that used in project 1, please confirm that you load the correct database when you start your work.</u>

Q1 (6 marks)

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

Write a PLpgSQL function Q1(integer) to generate transcript record for the given student. Each transcript tuple should contain the following information as follows: code, term, course, prog, name, mark, grade, uoc, rank and totalEnrols. The meaning of each attribute is explained in the following.

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)
        course integer,      -- course ID
        prog    char(4),      -- 4-digit program code for the program being studied when the
                                 course was studied
        name    text,         -- 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 awarded for the course
        rank integer,          -- the rank of the student in the corresponding course
        totalEnrols integer,   -- the total number of students enrolled in this course with
                                  non-null mark
);
```

Sample results (details can be found in check.sql):

proj2=#select * from q1(2237675)

code	term	course	prog	name	mark	grade	uoc	rank	totalEnrols
GBAT9101	06s2	23343	8616	Project Management	80	DN	6	5	19
GBAT9106	07s1	27143	8616	Information Systems	73	CR	6	5	9
…	…	…	…	…	…	…	…	…	…

Q2 (6 marks)

You may use any combination of views, SQL functions and PLpgSQL functions in this question. However, you must define at least a PLpgSQL function called Q2.

In some scenarios, we are interested in finding the number of tuples that have a given pattern (represented by regular expression) for each attribute in a table. For example, given a table 'Subjects' and a pattern 'COMP\d{3}', we would like to know the columns of 'Subjects' that 'COMP\d{3}' appears in the instances of these columns and also the number of such instances for each column(here we only count the number of matched rows).

Write a PLpgSQL function that displays all instances with the given pattern 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 in this column. You should only display the columns whose number of instances is greater than zero. Use the following type definition and function header:
create type MatchingRecord as ("table" text, "column" text, nexamples integer);
create or replace function Q2 (tableName text, pattern text) returns setof MatchingRecord ...

Sample results (details can be found in check.sql):
proj2=#select * from q2('subjects', 'COMP\d\d\d')

table	column	nexamples
subjects	code	265
subjects	syllabus	25
…	…	…

Q3 (7 marks)

You may use any combination of views, SQL functions and PLpgSQL functions in this question. However, you must define at least a PLpgSQL function called Q3.

UNSW staff members hold different roles at different periods of time. This information is recorded in `Affiliations` table. Given the id of an organization, we want to find all the staff members who have had at least two non-concurrent roles in the given organization (i.e. they must have two roles where role1′s ending date ≤ role 2′s starting date). Note that you need to consider that the given organization may have lots of sub-organizations. For example, the faculty of engineering has 9 schools, such as biomedical engineering and CSE.

Give the id of an organization, write a PLpgSQL function to (a) find all the staff members who have had several roles in the given organization over time and (b) produce a list showing, for each staff member, their unswid, name, and the roles they've had in the given organization (includes the name of staff role, the name of organization, and starting date and ending date). 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').
Use the following type definition and function header:
create type EmploymentRecord as (unswid integer, staff_name text, roles text);
create or replace function Q3(integer) returns setof EmploymentRecord ...
Note that PostgreSQL uses a + character to indicate an end-of-line in its output (as well as printing '\n'). And the staff should be printed in order of their People.sortname.

Sample results (details can be found in check.sql):

proj2=#select * from q3(661)

unswid	name	roles
3140956	Thomas Loveday	Senior Lecturer, Architecture Program (2011-09-05..2011-09-05) + Senior Lecturer, Interior Architecture Program (2011-11-25..) +
9226425	Stephen Ward	Program Head, Industrial Design Program (2001-01-01..2011-09-27) + Lecturer, Industrial Design Program (2011-09-27..) +

Section 2

1. Introduction

Graph is a data structure heavily used to represent complex data. A graph **G=(V,E)** consists of a set of nodes and a set of edges, where each node **v** refers to a data entity, and each edge **(u, v)** that connects two nodes **u** and **v** represents the relationships between the two associated nodes. Graphs have become a popular structure to represent data and its relationships. The social networks are graphs, where you can view a person as a node, and the friendship relation between two persons as an edge. In this project we only consider the simplest version of a graph - the **undirected**, **self-loop-free**, and **multi-edge-free** graph. We explain the three terms as follows.

- **Undirected**: Each edge of the graph has no direction, so the edge (u, v) is the same as the edge (v, u);
- **Self-loop-free**: There is no edge connecting a vertex to itself;
- **Multi-edge-free**: There is at most an edge between each pair of vertices.

A question naturally emerges in the social network: how many people among my friends' friends are my friends? To answer this question, we need to explore a very special structure, triangle, in the graph. A triangle in an undirected graph is a set of three vertices such that each possible edge between them is present is the graph. Triangle is the most studied structure in the area, related to not only answering our intuitive question - friends' friends are my friends - but many fundamental graph-related problems, such as structure study, graph simulation, pattern recognition, to just name a few. In this project, we will explore the triangle listing problem - to list all triangles in a given graph - by using SQL (PL/pgSQL) queries.

To express the idea, we need to see the graph as an edge relation table E(v_1, v_2), where v_1, v_2 stand for the two nodes of an edge. In this project, each node is represented with a unique integer, referred as the ***node id***. We denote *id(v)* as the id of a given node *v*. We show an example of the edge table in Example-1.

v_1,	v_2
1,	2
1,	3
1,	4
2,	3
2,	4
3,	4

Example 1

Note that the edge is given in a canonical format, in which the id of the first node is smaller than the second. In order to solve triangle listing, we enforce two joins based on the edge relation as follows,

$$\begin{aligned} W(v_1, v_2, v_3) &= \Pi_{(E_1.v_1, E_1.v_2, E_2.v2)} E_1(v_1, v_2) \bowtie_{(E_1.v1 = E_2.v_1)} E_2(v_1, v_2) \\ T(v_1, v_2, v_3) &= W(v_1, v_2, v_3) \bowtie_{(W.v_2 = E.v_1 \wedge W.v_3 = E.v_2)} E(v_1, v_2), \end{aligned} \tag{1}$$

where $W(v_1, v_2, v_3)$ is an intermediate relation called ***wedge*** - a path of length 2 that is centered on v_1, and $T(v_1, v_2, v_3)$ is the triangle. We show both structures in Figure-1. T is obtained from W by closing the two ends $W.v_2$ and $W.v_3$ with another edge. Note that the above E_1 and E_2 refer to the same edge table E, while we use the subscript for clarity. From the above equation, we notice that the wedge relation W is obtained by self-joining two edge relations, while the triangle relation T is processed by joining W to an extra edge relation E. In this project, you are required to use the above equations to implement the triangle listing task.

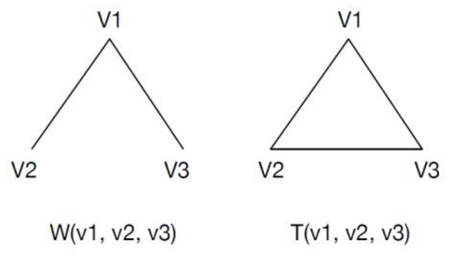

Figure 1: Wedge $W(v_1, v_2, v_3)$ and Triangle $T(v_1, v_2, v_3)$.

Given a triangle with three nodes as (v_1, v_2, v_3), it can be discovered in six ways, namely (v_1, v_2, v_3), (v_1, v_3, v_2), (v_2, v_1, v_3), (v_2, v_3, v_1), (v_3, v_1, v_2) and (v_3, v_2, v_1). In the following way, we will say (v_1, v_2, v_3) as the ***instance***, and the rests as the ***ghosts*** (Note that you can name any of them as the instance and the rest will be the ghosts). In other words, we can always obtain a ghost by permutating the node orders in the instance. In the triangle listing, we only keep the instance, while removing all ghosts, as they are simply duplicates of the instance.

A way to remove the ghosts is node orientation (ordering). Suppose we predefine an order < among all nodes. In other words, given any two nodes u, v, we can always tell whether u < v or v < u based on the order. For a given instance $T(v_1, v_2, v_3)$ where $v_1 < v_2 < v_3$, if we confine the order of the outputted triangle (v_1, v_2, v_3) as $v_1 < v_2 < v_3$, we will immediately rule out the ghosts as they clearly violate the order constraints. Therefore, we require in this project that each outputted triangle $T(v_1, v_2, v_3)$ must have $v_1 < v_2 < v_3$.

In terms of using two different orderings, we have the following two exercises.

2. Exercises

Q1 (3 marks)

Ordering based on the node id. Given node u and v, we say u < v if and only if *id(u)* < *id(v)*. Since it is required that each outputted triangle $T(v_1, v_2, v_3)$ satisfies $v_1 < v_2 < v_3$, we immediately have $id(v_1) < id(v_2) < id(v_3)$ in the id-based ordering.

<u>Q1</u>: In this problem, you are required to implement a function, **triangle_naive(dataset_name)**, which has the "dataset_name" (graph) as input, and outputs the number of valid triangles $T(v_1, v_2, v_3)$ with $id(v_1) < id(v_2) < id(v_3)$ in the graph.

<u>Hint</u>: You will implement the two joins shown in Equation-1. Make sure that you output the wedge $W(v_1, v_2, v_3)$ in the order $id(v_1) < id(v_2) < id(v_3)$, as is required for $T(v_1, v_2, v_3)$. You might want to use the dynamic sql - the *Execute* clause - in PL/pgSQL.

Q2 (3 marks)

Ordering based on the degree. We say u is v's **neighbor** in the graph, if (u, v) or (v, u) is an edge. Let's use *deg(v)* to represent the degree - the number of neighbors - of the node v.
As shown in Example-1, the node 1's neighbors are {2, 3, 4}, and 2's neighbors are {1, 3, 4}, et al. And *deg(1)* = *deg(2)* = *deg(3)* = *deg(4)* = 3.

The performance of the triangle listing depends heavily on the number of intermediate results, aka the wedges. As you might have noticed, it is not necessary that a wedge turns into a triangle. If not, futile load is introduced. The more such futile load is, the slower the algorithm is. As a result, the id-based ordering sometimes works poorly, especially in a real graph. In the real graph, we often expect that some nodes have extremely large degree. Take the social network Twitter as an example. Normally people have hundreds of followers, and someone famous attaches more, while super stars like Justin Bieber and Barack Obama draw millions. Suppose a super-star node v like Obama with 10^6 followers ($deg(v) = 10^6$) is assigned with id = 1 (the smallest id). According to the id-based order, v will be the smallest node in any wedge that is potentially a triangle. Therefore, any two followers of v out of 10^6 can form a wedge centered on v, rendering $10^{12}/2$ wedges as a whole, among which only a few will finally become valid triangles (considering as an example that most of Obama's followers do not necessarily know each other). We hence need to apply more advanced ordering. Given two nodes v_1 and v_2, we say $v_1 < v_2$ *if* $deg(v_1) < deg(v_2)$ or $deg(v_1) = deg(v_2)$ $\wedge$

id(v_1) < id(v_2). In other words, we will order the nodes via their degrees, with id to break the tie.

What is the benefit of doing so? Let's go back to the example of super-star node v, where deg(v) = 10^6. As the degree of v is so large, we can hardly find any its neighbors having larger degree (larger order). As the wedges centered on v must have v as the smallest node, the degree-based ordering avoids the generation of a large number of unwanted wedges centered on the super-star nodes.

Q2: In this problem, you are required to implement a function, **triangle_in_order(dataset_name)**, which has the "dataset_name" (graph) as input, and outputs the number of valid triangles (T(v_1, v_2, v_3) with $v_1 < v_2 < v_3$) according to the degree ordering in the graph.

Hint: You should first generate the degree for each node, and then attach the degree to both ends of the edge, which allows you to do node comparison. You can compare the number of wedges generated here to that is generated in part 1. A large reduction should be expected.

Specifications: we provide several files to help you finish this question:

- graph.sql. a template file for your work.
- store_graph.sh. The script helps create a database "graph", a table "tbl_dataset0", and insert the edge data specified in "dataset_0.txt" into the table "tbl_dataset0". Note that we will test several datasets other than ``dataset0'', so you are required to provide a parameter "dataset_name" in both functions. And the "dataset_name" specified in the functions is related to the edge table "tbl_{dataset_name}". For example, in the test dataset: "dataset0", the edge table is "tbl_dataset0", and you should process the query by calling the function as "select * from triangle_naive(dataset0)" and "select * from triangle_in_order(dataset0);", respectively. As a result, you have to check that the table"tbl_{dataset_name}" exists in both cases.
- dataset_0.txt, the graph dataset
- check_graph.sql. This is the test file. While implementing both functions, you can use "check_graph.sql" to check whether the results are correct. Simply load the "check_graph.sql" into the "graph" database by calling "psql graph -f check_graph.sql", and in the "graph" database, you can use "select * from check_all();'' to query the results. The testing outputs would be as follows when everything is correct:

dataset	test	result
dataset0	triangle_naive	correct
dataset0	triangle_in_order	correct

Submission

You can submit this project by doing the following:

- Ensure that you are in the directory containing the files to be submitted, includes proj2.sql and graph.sql.
- Submit the project by typing "give cs3311 proj2 proj2.sql graph.sql".
- If you submit your project more than once, the last submission will replace the previous one.
- **To prove successful submission, please take a screenshot as assignment submission manual shows and keep it by yourself.**
- If you have any problems in submissions, please email to **longyuan@cse.unsw.edu.au**. You can also ask questions about this project in our project group, we will answer your question as soon as possible.

Before you submit your solution, you should check that it will load correctly. If we need to manually fix problems with your file in order to test it, you will be fined via half of the mark penalty for each problem.

Late Submission Penalty

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