COMP3311 21T1 Database Systems

Week 05
Stored Functions in SQL and PLpgSQL

1. Write a simple PLpgSQL function that returns the square of its argument value. It is used as follows:

```
mydb=> select sqr(4);
 sqr
-----
  16
(1 row)

mydb=> select sqr(1000);
   sqr
---------
 1000000
(1 row)
```

Could we use this function in any of the following ways?

```
select sqr(5.0);
select(5.0::integer);
select sqr('5');
```

If not, how could we write a function to achieve this?

Answer:

```
create or replace function sqr(n integer) returns integer
as $$
begin
    return n * n;
end;
$$ language plpgsql;
```

This function won't square real numbers, even something like:

```
mydb=> select sqr(3.0);
ERROR:  Function sqr(numeric) does not exist
```

The defined function has type `integer`→`integer`. PostgreSQL is looking for a function with type `float`→`integer` or, more generic `numeric`→`numeric`.

However, it works if you type-cast it to an integer (second example). And PostgreSQL seems reasonable about converting strings of digits into appropriate numbers.

By declaring the function to use the generic number type `numeric`, it will handle both integers and reals correctly:

```
create or replace function sqr(n numeric) returns numeric
as $$
begin
    return n * n;
end;
$$ language plpgsql;
```

2. Write a PLpgSQL function that "spreads" the letters in some text. It is used as follows:

```
mydb=> select spread('My Text');
     spread
----------------
 M y   T e x t
(1 row)
```

Answer:

Here's a version that doesn't give a name to the parameter and uses "positional notation" (e.g. `$1`) to refer to the parmeter.

```
create or replace function spread(text) returns text
as $$
declare
    result text := '';
    i      integer;
    len    integer;
begin
    i := 1;
    len := length($1);
    while (i <= len) loop
        result := result || substr($1, i, 1) || ' ';
        i := i+1;
    end loop;
    return result;
```

```
end;
$$ language plpgsql;
```

OR, using named parameters and a `for` loop

```
create or replace function spread(str text) returns text
as $$
declare
    result text := '';
    i      integer;
begin
    i := 1;
    for i in 1..length($1) loop
        result := result || substr(str, i, 1) || ' ';
    end loop;
    return result;
end;
$$ language plpgsql;
```

Note that if you omit the initial assignment of empty string to `result`, then the value of `result` stays as `NULL` throughout the entire function execution, and `NULL` is returned (i.e., string concatenation is a `NULL`-preserving operation).

3. Write a PLpgSQL function to return a table of the first *n* positive integers.

   The fuction has the following signature:

```
create or replace function seq(n integer) returns setof integer
```

   and is used as follows:

```
mydb=> select * from seq(5);
 seq
-----
   1
   2
   3
   4
   5
(5 rows)
```

   Answer:

```
create or replace
    function seq(n integer) returns setof integer
as $$
declare
    i integer;
begin
    for i in 1 .. n
    loop
        return next i;
    end loop;
end;
$$ language plpgsql;
```

4. Generalise the previous function so that it returns a table of integers, starting from *lo* up to at most *hi*, with an increment of *inc*. The function should also be able to count down from *lo* to *hi* if the value of *inc* is negative. An *inc* value of 0 should produce an empty table. Use the following function header:

   ```
   create or replace function seq(lo int, hi int, inc int) returns setof integer
   ```

   and the function would be used as follows:

   ```
   mydb=> select * from seq(2,7,2);
    val
   -----
     2
     4
     6
   (3 rows)
   ```

   Some other examples, in a more compact representation:

   ```
   seq(1,5,1)   gives   1  2  3  4  5
   seq(5,1,-1)  gives   5  4  3  2  1
   seq(9,2,-3)  gives   9  6  3
   seq(2,9,-1)  gives   empty
   seq(1,5,0)   gives   empty
   ```

   Answer:

   ```
   create or replace function
       seq(lo int, hi int, inc int) returns setof integer
   as $$
   ```

```
declare
    i integer;
begin
    i := lo;
    if (inc > 0) then
        while (i <= hi)
        loop
            return next i;
            i := i + inc;
        end loop;
    elsif (inc < 0) then
        while (i >= hi)
        loop
            return next i;
            i := i + inc;
        end loop;
    end if;
    return;
end;
$$ language plpgsql;
```

5. Re-implement the `seq(int)` function from above as an **SQL function**, and making use of the generic `seq(int,int,int)` function defined above.

   Answer:

```
create or replace function
    seq(n int) returns setof integer
as $$
select * from seq(1,n,1);
$$ language sql;
```

6. Create a factorial function based on the above sequence returning functions.

```
create function fac(n int) returns integer
```

   Implement it as an **SQL function** (not a PLpgSQL function). The obvious solution to this problem requires a `product` aggregate, analogous to the `sum` aggregate. PostgreSQL does not actually have a `product` aggregate, but for the purposes of this question, you can assume that it does, and has the following interface:

```
product(list of integers) returns integer
```

Answer:

```
create function fac(n int) returns integer
as $$
select product(seq) from seq(n);
$$
language sql;
```

Note that the `sum()` aggregate actually returns a value of type `bigint`, and so, probably, should `product()`.

Use the old Beers/Bars/Drinkers database in answering the following questions. A summary schema for this database:

```
Beers(name:string, manufacturer:string)
Bars(name:string, address:string, license#:integer)
Drinkers(name:string, address:string, phone:string)
Likes(drinker:string, beer:string)
Sells(bar:string, beer:string, price:real)
Frequents(drinker:string, bar:string)
```

Primary key attributes are in **bold**. Foreign key attributes are in ***bold italic***.

The examples below assume that the user is connected to a database called `beer` containing an instance of the above schema.

7. Write a PLpgSQL function called `hotelsIn()` that takes a single argument giving the name of a suburb, and returns a text string containing the names of all hotels in that suburb, one per line.

```
create function hotelsIn(_addr text) returns text
```

The function is used as follows:

```
beer=> select hotelsIn('The Rocks');
    hotelsin
-----------------
 Australia Hotel+
 Lord Nelson    +

(1 row)
```

Can you explain what the `'+'` at the end of each line is? And why it says `(1 row)`?

Note that the output from functions returning a single `text` string and looks better if you turn off `psql`'s output alignment (via `psql`'s **`\a`** command) and column headings (via `psql`'s **`\t`** command).

Compare the aligned output above to the unaligned output below:

```
beer=> \a
Output format is unaligned.
beer=> \t
Showing only tuples.
beer=> select hotelsIn('The Rocks');
Australia Hotel
Lord Nelson
```

From now on, sample outputs for functions returning `text` will assume that we have used **`\a`** and **`\t`**.

Answer:

Reminder: this function returns a single string and not a list of tuples.

```
create or replace function
    hotelsIn(_addr text) returns text
as $$
declare
    r   record;
    out text := '';
begin
    for r in select * from bars where addr = _addr
    loop
        out := out || r.name || e'\n';
    end loop;
    return out;
end;
$$ language plpgsql;
```

8. Write a new PLpgSQL function called `hotelsIn()` that takes a single argument giving the name of a suburb and returns the names of all hotels in that suburb. The hotel names should all appear on a single line, as in the following examples:

```
beer=> select hotelsIn('The Rocks');
Hotels in The Rocks:  Australia Hotel  Lord Nelson

beer=> select hotelsIn('Randwick');
Hotels in Randwick:  Royal Hotel
```

```
beer=> select hotelsIn('Rendwik');
There are no hotels in Rendwik
```

Answer:

```
create or replace function
    hotelsIn (_addr text) returns text
as $$
declare
    howmany integer;
    pubnames text;
    p record;
begin
    select count(*) into howmany from Bars where addr = _addr;
    if (howmany = 0) then
        return 'There are no hotels in '|| _addr || e'\n';
    end if;
    pubnames:= 'Hotels in ' || _addr || ':';
    for p in select * from Bars where addr = _addr
    loop
        pubnames := pubnames||'  '||p.name;
    end loop;
    pubnames := pubnames||e'\n';
    return pubnames;
end;
$$ language plpgsql;
```

9. Write a PLpgSQL procedure `happyHourPrice` that accepts the name of a hotel, the name of a beer and the number of dollars to deduct from the price, and returns a new price. The procedure should check for the following errors:

   - non-existent hotel (invalid hotel name)
   - non-existent beer (invalid beer name)
   - beer not available at the specified hotel
   - invalid price reduction (e.g. making reduced price negative)

   Use `to_char(price,'$9.99')` to format the prices.

```
beer=> select happyHourPrice('Oz Hotel','New',0.50);
There is no hotel called 'Oz Hotel'

beer=> select happyHourPrice('Australia Hotel','Newer',0.50);
There is no beer called 'Newer'
```

```
beer=> select happyHourPrice('Australia Hotel','New',0.50);
The Australia Hotel does not serve New

beer=> select happyHourPrice('Australia Hotel','Burragorang Bock',4.50);
Price reduction is too large; Burragorang Bock only costs $ 3.50

beer=> select happyHourPrice('Australia Hotel','Burragorang Bock',1.50);
Happy hour price for Burragorang Bock at Australia Hotel is $ 2.00
```

Answer:

Checking for existence of some tuples could be done using either `select count(*)` followed by a check for zero, or by using the `FOUND` variable (which is set after each query). This solution combines both approaches to show the range of possiblities.

In general, you could use `count(*)` whenever you knew that you were not interested in collecting any other information from the table; you'd try to collect the information and use `FOUND` in all other circumstances.

```
-- using positional notation for parameters
create or replace function
    happyHourPrice (_hotel text, _beer text, _discount real) returns text
as $$
declare
    counter integer;
    std_price real;
    new_price real;
begin
    select count(*) into counter from Bars where name = _hotel;
    if (counter = 0) then
        return 'There is no hotel called '|| _hotel ||e'\n';
    end if;
    select * from Beers where name = _beer;  -- any results vanish
    if (not found) then
        return 'There is no beer called '|| _beer ||e'\n';
    end if;
    select price into std_price
    from   Sells s
    where  s.beer = _beer and s.bar = _hotel;
    if (not found) then
        return 'The '|| _hotel || ' does not serve '||_beer;
    end if;
    new_price := std_price - _discount;
    if (new_price < 0) then
```

```
            return 'Price reduction is too large; '
                       || _beer || ' only costs '
                       || to_char(std_price, '$9.99');
        else
            return 'Happy hour price for '
                       || _beer || ' at '|| _hotel ||' is '
                       || to_char(new_price, '$9.99');
        end if;
end;
$$ language plpgsql;
```

10. The `hotelsIn` function above returns a formatted string giving details of the bars in a suburb. If we wanted to return a table of records for the bars in a suburb, we could use a view as follows:

```
beer=> create or replace view HotelsInTheRocks as
    -> select * from Bars where addr = 'The Rocks';
CREATE VIEW
beer=> select * from HotelsInTheRocks;
      name        |   addr    | license
-----------------+-----------+---------
 Australia Hotel | The Rocks |  123456
 Lord Nelson     | The Rocks |  123888
(2 rows)
```

Unfortunately, we need to specify a suburb in the view definition. It would be more useful if we could define a "parameterised view" which we could use to generate a table for any suburb, e.g.

```
beer=> select * from HotelsIn('The Rocks');
      name        |   addr    | license
-----------------+-----------+---------
 Australia Hotel | The Rocks |  123456
 Lord Nelson     | The Rocks |  123888
(2 rows)
beer=> select * from hotelsIn('Coogee');
       name        |  addr   | license
------------------+---------+---------
 Coogee Bay Hotel | Coogee |  966500
(1 row)
```

Such a parameterised view can be implemented via an SQL function, defined as:

```
create or replace function hotelsIn(text) returns setof Bars
as $$ ... $$ language sql;
```

Complete the definition of the SQL function.

Answer:

```
create or replace function
    hotelsIn(text) returns setof Bars
as $$
select * from Bars where addr = $1;
$$ language sql;
```

11. The function for the previous question can also be implemented in PLpgSQL. Give the PLpgSQL definition. It would be used in the same way as the above.

Answer:

```
create or replace function
    hotelsIn(_addr text) returns setof Bars
as $$
declare
    r record;  -- could also be declared r Bars%rowtype;
begin
    for r in select * from Bars where addr = _addr
    loop
       return next r;
    end loop;
    return;
end;
$$ language plpgsql;
```

Use the Bank Database in answering the following questions. A summary schema for this database:

```
Branches(location:text, address:text, assets:real)
Accounts(holder:text, branch:text, balance:real)
Customers(name:text, address:text)
Employees(id:integer, name:text, salary:real)
```

The examples below assume that the user is connected to a database called `bank` containing an instance of the above schema.

12. For each of the following, write both an SQL and a PLpgSQL function to return the result:

a. salary of a specified employee

Answer:

```
-- Salary of a specified employee
--   Allows employee to be determined by name or id
--    using overloading on the function name
--   Assume name or id identifies only one employee

create or replace function empSal(text) returns real
as $$
select salary from employees where name = $1
$$ language sql;

create or replace function empSal(integer) returns real
as $$
    select salary from employees where id = $1
$$ language sql;

create or replace function
    empSal1(_name text) returns real
as $$
declare
    _sal real;
begin
    select salary into _sal
    from employees where name = _name;
    return _sal;
end;
$$ language plpgsql;

create or replace function
    empSal1(_id integer) returns real
as $$
declare
    _sal real;
begin
    select salary into _sal
    from employees where id = _id;
    return _sal;
```

```
end;
$$ language plpgsql;
```

b. all details of a particular branch

Answer:

```
-- All details of a particular branch
--   Example of PLpgSQL function returning a record

create or replace function branchDetails(text) returns Branches
as $$
    select * from Branches where location = $1;
$$ language sql;


create or replace function branchDetails1(_bname text) returns Branches
as $$
declare
    _tup Branches;
begin
    select * into _tup
    from Branches where location = _bname;
    return _tup;
end;
$$ language plpgsql;
```

c. names of all employees earning more than $*sal*

Answer:

```
-- Names of all employees earning more than $sal
--   Example of PLpgSQL function returning a set of atomic values

create or replace function empsWithSal(real) returns setof text
as $$
    select name from employees where salary > $1;
$$ language sql;

create type EmpName as ( name text );

create or replace function empsWithSal1(_minSal real) returns setof EmpName
as $$
```

```
declare
    _en EmpName;
begin
    for _en in select name
           from employees where salary > _minSal
    loop
        return next _en;
    end loop;
    return;
end;
$$ language plpgsql;
```

d. all details of highly-paid employees

Answer:

```
-- All details of highly-paid employees
--   Example of PLpgSQL function returning a set of atomic values

create or replace function richEmps(real) returns setof Employees
as $$
    select * from employees where salary > $1;
$$ language sql;

create or replace function emps1(_minSal real) returns setof Employees
as $$
declare
    _e Employee;
begin
    for _e in select *
              from employees where salary > _minSal
    loop
        return next _e;
    end loop;
    return;
end;
$$ language plpgsql;
```

13. Write a PLpgSQL function to produce a report giving details of branches:

- name and address of branch
- list of customers who hold accounts at that branch
- total amount in accounts held at that branch

Use the following format for each branch:

```
Branch: Clovelly, Clovelly Rd.
Customers:  Chuck Ian James
Total deposits: $   8860.00
```

Answer:

```
create or replace function branchList() returns text
as $$
declare
    a   record;
    b   record;
    tot integer;
    qry text;
    out text := e'\n';
begin
    for b in select * from Branches
    loop
        out := out || 'Branch: ' || b.location || ', ';
        out := out || b.address || e'\n' || 'Customers: ';
        tot := 0;
        for a in select * from Accounts where branch=b.location
        loop
            out := out || ' ' || a.holder;
            tot := tot + a.balance;
        end loop;
        select sum(balance) into tot
        from Accounts where branch=b.location;
        out := out || E'\nTotal deposits: ';
        out := out || to_char(tot,'$999999.99');
        out := out || E'\n---\n';
    end loop;
    return out;
end;
$$ language plpgsql;
```

It's also possible to implement this more efficiently using just one SQL query (rather than nested-loop queries). The more efficient solution invloves ordering the Accounts tuples by branch, and keeping track of when the current branch changes to a new one.

Use the following database schema, which is somehwat similar to the schema for Assignment 2. The schema is too large to give a complete summary here, but we provide some details for some tables:

```
Term(id:integer, year:integer, session:('S1','S2','X1','X2'), ...)
Subject(id:integer, code:text, ..., name:text, ... uoc:integer, ...)
Course(id:integer, subject:integer, term:integer, lic:integer, ...)
OrgUnit(id, utype, name, longname, ...)
OrgUnitType(id, name)
Person(id:integer, ..., name:text, ...)
Student(id:integer, sid:integer, stype:('local','intl'))
Staff(id:integer, sid:integer, office:integer, ...)
StaffRole(id, descript)
Affiliation(staff, orgunit, role, fraction)
```

Note that there is an example database unsw.dump (3.5MB) that you could load into a newly created database to help with these problems, although you should be able to solve them without reference to a specific database instance. Note that all of the people data in this database is synthetic and the various enrolment tables have been cleared to save space.

The examples below assume that the user is connected to a database called `unsw` containing an instance of the above schema.

14. Write a PLpgSQL function to produce the complete name of an organisational unit (aka OrgUnit), given the OrgUnit's internal id:

```
function unitName(_ouid integer) returns text
```

This will need to make use of the `OrgUnit` and `OrgUnitType` tables. The `OrgUnitType` table contains a list of unit types (e.g. faculty, school, institute) via *(id,name)* tuples. The `OrgUnit` table has a foreign key to the `OrgUnitType` table to indicate what kind of unit it is. The attribute contains the useful name of the unit (the `name` attribute is a very abbreviated version of the unit's name). The `longname` attribute for faculties already contains the words "Faculty of". For other kinds of `OrgUnit`, you need to prepend the name of its `OrgUnitType`.

The function returns the complete name using the rules:

- the university is denoted by UNSW
- a faculty is denoted using its base name (not all faculty names start with Faculty)
- a school is denoted School of XYZ
- a department is denoted Department of XYZ
- a centre is denoted Centre for XYZ
- an institute is denoted Institute of XYZ
- other kinds of OrgUnits are treated as having no name (i.e. return null)

Some examples of usage (assuming `\a` and `\t`):

```
unsw=> select unitName(0);
UNSW

unsw=> select unitName(2);
Faculty of Arts and Social Sciences
```

```
unsw=> select unitName(4);
Faculty of Law

unsw=> select unitName(9);
Faculty of Engineering

unsw=> select unitName(11);
Faculty of Science

unsw=> select unitName(36);
School of Chemistry

unsw=> select unitName(44);
School of Computer Science and Engineering

unsw=> select unitName(75);
Centre for Human Geography

unsw=> select unitName(92);
Department of Korean Studies

unsw=> select unitName(999);
ERROR:  No such unit: 999
```

Answer:

```
create or replace function unitName(_ouid integer) returns text
as $$
declare
    _outype text;
    _ouname text;
begin
    -- check whether the orgunit ID is valid
    select * from OrgUnit where id = _ouid;
    if (not found) then
        raise exception 'No such unit: %',_ouid;
    end if;

    select t.name,u.longname into _outype,_ouname
    from   OrgUnitType t, OrgUnit u
    where  u.id = _ouid and u.utype = t.id;

    -- debugging output
    -- raise notice 'Type:%, Name:%',_outype,_ouname;

    if (_outype = 'UNSW') then
        return 'UNSW';
    elsif (_outype = 'Faculty') then
        return _ouname;
    elsif (_outype = 'School') then
        return 'School of '||_ouname;
    elsif (_outype = 'Department') then
        return 'Department of '||_ouname;
    elsif (_outype = 'Centre') then
        return 'Centre for '||_ouname;
    elsif (_outype = 'Institute') then
        return 'Institute of '||_ouname;
    else
        return null;
    end if;
end;
$$ language plpgsql;
```

An alternative, using an SQL `CASE` expression:

```
create or replace function unitName(_ouid integer) returns text
as $$
declare
    _ouname text;
begin
    -- check whether the orgunit ID is valid
    select * from OrgUnit where id = _ouid;
    if (not found) then
        raise exception 'No such unit: %',_ouid;
    end if;

    select case
           when t.name = 'UNSW' then 'UNSW'
           when t.name = 'Faculty' then t.longname
           when t.name = 'School' then 'School of '||t.longname
           when t.name = 'Department' then 'Department of '||t.longname
           when t.name = 'Centre' then 'Centre for '||t.longname
           when t.name = 'Institute' then 'Institute of '||t.longname
           else null
           end into _ouname
    from   OrgUnitType t, OrgUnit u
    where  u.id = _ouid and u.utype = t.id;
    return _ouname;
end;
$$ language plpgsql;
```

If you didn't care about error-checking on the OrgUnit ID, then this could be done as an SQL function.

15. In the previous question, you needed to know the internal ID of an `OrgUnit`. This is unlikely, so write a function that takes part of an `OrgUnit.longname` and returns the ID or `NULL` if there is no such unit. If there is more than one matching unit, return the ID of the first matching unit. Implement this as an SQL function, which allows case-insensitive matching:

```
create or replace function unitID(partName text) returns integer
as $$ ... $$ language sql;
```

Examples of usage:

```
unsw=> select unitName(unitID('law'));
Faculty of Law

unsw=> select unitName(unitID('arts'));
Faculty of Arts and Social Sciences

unsw=> select unitName(unitID('information'));
School of Information Management

unsw=> select unitName(unitID('information sys'));
School of Information Systems

unsw=> select unitName(unitID('chem'));
Department of Biochemistry

unsw=> select unitName(unitID('computer'));
School of Computer Science (ADFA)

unsw=> select unitName(unitID('comp%sci%eng'));
School of Computer Science and Engineering

unsw=> select unitName(unitID('korean'));
Department of Korean Studies
```

We use `unitName()` as a way of checking the result Note that such a simple text-based search can produce unexpected results.

Answer:

A simple solution (and probably the only one possible in plain SQL:

```
create or replace function unitID(partName text) returns integer
as $$
   select id from OrgUnit where longname ilike '%'||partname||'%';
$$ language sql;
```

16. Write a PLpgSQL function which takes the numeric identifier of a given OrgUnit and returns the numeric identifier of the parent faculty for the specified OrgUnit:

```
function facultyOf(_ouid integer) returns integer
```

Note that a faculty is treated as its own parent. Note also that some OrgUnits don't belong to any faculty; such OrgUnits should return a null result from the function.

Examples of use:

```
unsw=> select unitName(facultyof(2));
Faculty of Arts and Social Sciences

unsw=> select unitName(facultyof(9));
Faculty of Engineering

unsw=> select unitName(facultyof(36));
Faculty of Science

unsw=> select unitName(facultyof(44));
Faculty of Engineering

unsw=> select unitName(facultyof(75));
Faculty of Science

unsw=> select unitName(facultyof(92));
Faculty of Arts and Social Sciences

unsw=> select unitName(facultyof(999));
ERROR:  No such unit: 999
```

Answer:

```
create or replace function facultyOf(_ouid integer) returns integer
as $$
declare
    _count integer;
    _tname text;
    _parent integer;
begin
    select count(*) into _count
    from orgUnit where id = _ouid;
    if (_count = 0) then
        raise exception 'No such unit: %',_ouid;
    end if;

    select t.name into _tname
    from OrgUnit u, OrgUnitType t
    where u.id = _ouid and u.utype = t.id;

    if (_tname is null) then
        return null;
    elsif (_tname = 'University') then
        return null;
    elsif (_tname = 'Faculty') then
        return _ouid;
    else
        select owner into _parent
        from UnitGroups where member = _ouid;
        return facultyOf(_parent);
    end if;
end;
$$ language plpgsql;
```

An alternative way of checking the existence of the specified organisational unit would be:

```
    select * from OrgUnit where id = _ouid;
    if (not found) then
        raise exception 'No such unit: %',_ouid;
    end if;
```