Exercises 01

DBMSs, PostgreSQL, Catalogs

Some of these questions require you to look beyond the Week 01 lecture material for answers. Some of the questions preempt material that we'll be looking at over the next few weeks. To answer some questions, you may need to look at the PostgreSQL documentation or at the texts for the course ... or, of course, you could simply reveal the answers, but where's the fun in that?

1. List some of the major issues that a relational database management system needs to concern itself with.

   Answer:

   - persistent storage of data and meta-data
   - executing SQL queries on stored data
   - maintenance of constraints on stored data
   - extensibility via views, triggers, procedures
   - query processing (optimisation and efficient execution)
   - transaction processing semantics
   - control of concurrent access by multiple users
   - recovery from failures (rollback, system failure)

2. Give an overview of the major stages in answering an SQL query in a relational database management system. For each step, describe its inputs and outputs and give a brief description of what it does.

   Answer:

   0. start with the text string of an SQL query
      - e.g. `select e.name,d.name from Employee e, Dept d where e.id=d.manager;`
   1. **parsing and translation**
      - converts an SQL query into a relational algebra expression
      - input: text string of an SQL query
      - output: expression tree for a relational algebra expression
   2. **optimisation**
      - converts RA expression into query plan
      - input: relational algebra expression tree
      - output: sequence of DBMS-specific relational operations
   3. **execution**
      - performs relational operations, via chained intermediate results
      - input: query plan (sequence of DBMS-specific relational operations)
      - output: set of result tuples, stored either in memory or on disk
   4. **output**
      - convert the result tuples into a format useful for the client
      - input: tuple data in memory buffers (and possibly on disk as well)

- output: stream of formatted tuples (format defined by library e.g. PostgreSQL's `libpq`)

3. PostgreSQL is an "object-relational database management system". What are the differences between PostgreSQL and a "conventional" relational database management system (such as Oracle)?

   Answer:

   - every database tuple has an associated object identifier
   - tables can be defined as specialisations of other tables (inheritance)
   - can define new data types and operations on those types

4. A PostgreSQL installation includes a number of different "scopes": *databases* (or catalogs), *schemas* (or namespaces), and *tablespaces*. The scopes correspond to notions from the SQL standard. Explain the difference between these and give examples of each.

   Answer:

   - **database** (or **catalog**) ... a logical scope that collects together a number of schemas; an example is `template1`, a special database that is cloned whenever a user creates a new database; details of databases are held in the `pg_database` catalog table

   - **schema** (or namespace) ... a logical scope used as a namespace; contains a collection of database objects (tables, views, functions, indexes, triggers, ...); an example is the `public` schema available as a default in all databases; details of schemas are held in the `pg_namespace` catalog table

   - **tablespace** ... a physical scope identifying a region of the host filesystem where PostgreSQL data files are stored; an example is the `pg_default` tablespace, which corresponds to the `PG_DATA` directory where most PostgreSQL data files are typically stored; details of tablespaces are held in the `pg_tablespace` catalog table

5. For each of the following command-line arguments to the `psql` command, explain what it does, when it might be useful, and how you might achieve the same effect from within `psql`:

   a. `-l`
   b. `-f`
   c. `-a`
   d. `-E`

   Answer:

   a. `psql -l`

      Generates a list of all databases in your cluster; would be useful if you couldn't remember the exact name of one of your databases.

      You can achieve the same effect from within `psql` via the command `\list` or simply `\l`

b. `psql db -f file`

Connects to the database *db* and reads commands from the file called `file` to act on that database; useful for invoking scripts that build databases or that run specific queries on them; only displays the output from the commands in `file`.

You can achieve the same effect from within `psql` via the command `\i file`

c. `psql -a db -f file`

Causes all input to `psql` to be echoed to the standard output; useful for running a script on the database and being able to see error messages in the context of the command that caused the error.

You can achieve the same effect from within `psql` via the command `\set ECHO all`

d. `psql -E db`

Connect to the database *db* as usual; for all of the `psql` catalog commands (such as `\d`, `\df`, etc.), show the SQL query that's being executed to produce it; useful if you want to learn how to use the catalog tables.

You can achieve the same effect from within `psql` via the command `\set ECHO_HIDDEN on`

6. PostgreSQL has two main mechanisms for adding data into a database: the SQL standard `INSERT` statement and the PostgreSQL-specific `COPY` statement. Describe the differences in how these two statement operate. Use the following examples, which insert the same set of tuples, to motivate your explanation:`

```
insert into Enrolment(course,student,mark,grade)
        values ('COMP9315', 3312345, 75, 'DN');
insert into Enrolment(course,student,mark,grade)
        values ('COMP9322', 3312345, 80, 'DN');
insert into Enrolment(course,student,mark,grade)
        values ('COMP9315', 3354321, 55, 'PS');

copy Enrolment(course,student,mark,grade) from stdin;
COMP9315        3312345 75      DN
COMP9322        3312345 80      DN
COMP9315        3354321 55      PS
\.
```

Answer:

Each `insert` statement is a transaction in its own right. It attempts to add a single tuple to the database, checking all of the relevant constraints. If any of the constraints fails, that particular insertion operation is aborted and the tuple is not inserted. However, any or all of the other `insert` statements may still succeed.

A `copy` statement attempts to insert all of the tuples into the database, checking constraints as it goes. If any constraint fails, the `copy` operation is halted, and none of the tuples are added to the table†.

For the above example, the `insert` statements may result in either zero or 1 or 2 or 3 tuples being inserted, depending on whether how many values are valid. For the `copy` statement, either zero or 3 tuples will be added to the table, depending on whether any tuple is invalid or not.

† A fine detail: under the `copy` statement, tuples are "temporarily" added to the table as the statement progresses. In the event of an error, the tuples are all marked as invalid and are not visible to any query (i.e. they are effectively *not* added to the table). However, they still occupy space in the table. If a very large `copy` loads e.g. 9999 or 10000 tuples and the last tuple is incorrect, space has still been allocated for the most of the tuples. The `vacuum` function can be used to clean out the invalid tuples.

7. In `psql`, the `\timing` command turns on a timer that indicates how long each SQL command takes to execute. Consider the following trace of a session asking the several different queries multiple times:

```
unsw=# \timing
Timing is on.
unsw=# select max(id) from students;
   max
---------
 9904944
Time: 112.173 ms
unsw=# select max(id) from students;
   max
---------
 9904944
Time: 0.533 ms
unsw=# select max(id) from students;
   max
---------
 9904944
Time: 0.484 ms
unsw=# select count(*) from courses;
 count
-------
 80319
Time: 132.416 ms
unsw=# select count(*) from courses;
 count
-------
 80319
Time: 30.438 ms
```

```
unsw=# select count(*) from courses;
 count
-------
 80319
Time: 34.034 ms
unsw=# select max(id) from students;
   max
---------
 9904944
Time: 0.765 ms
unsw=# select count(*) from enrolments;
  count
---------
 2816649
Time: 2006.707 ms
unsw=# select count(*) from enrolments;
  count
---------
 2816649
Time: 1099.993 ms
unsw=# select count(*) from enrolments;
  count
---------
 2816649
Time: 1109.552 ms
```

Based on the above, suggest answers to the following:

a. Why is there such variation in timing between different executions of the same command?
b. What timing value should we ascribe to each of the above commands?
c. How could we generate reliable timing values?
d. What is the accuracy of timing results that we can extract like this?

Answer:

a. Variation:

There's a clear pattern in the variations: the first time a query is executed it takes *significantly* longer than the second time its executed (e.g. the first query drops from over 100ms to less than 1ms). This is due to caching effects. PostgreSQL has a large in-memory buffer-pool. The first time a query is executed, the relevant pages will need to be read into memory buffers from disk. The second and subsequent times, the pages are already in the memory buffers.

b. Times:

Given the significantly different contexts, it's not really plausible to assign a specific time to a query. Assigning a range of values, from "cold" execution (when none of the data for the query is buffered) to "hot" execution (when as much as possible of the needed data is buffered), might be more reasonable. Even then, you would need to measure the hot and cold execution several times and take an average.

How to achieve "cold" execution multiple times? It's difficult. Even if you stop the PostgreSQL server, then restart it, effectively flushing the buffer pool, there is still some residual buffering in the Unix file buffers. You would need to read lots of other files to flush the Unix buffers.

c. Reliability:

This is partially answered in the previous question. If you can ensure that the context (hot or cold) is the same at the start of each timing, the results will be plausibly close. Obviously, you should run each test on the same lightly-loaded machine (to minimise differences caused by Unix buffering). You should also ensure that you are the only user of the database server. If multiple users are competing for the buffer pool, the times could variably substantially and randomly up or down between subsequent runs, depending on how much of your buffered data had been swapped out to service queries from other users.

d. Accuracy:

For comparable executions of the query (either buffers empty or buffers fully-loaded), it looks like it's no more accurate than +/- 10ms. It might even be better to forget about precise time measures, and simply fit queries into "ball-park" categories, e.g.

- Very fast ... $0 \le t < 100$ms
- Fast ... $100 \le t < 500$ms
- Acceptable ... $500 \le t < 2000$ms
- Slow ... $2000 \le t < 10000$ms
- Too Slow ... $t > 10000$ms

Note that the above queries were run on a PostgreSQL 8.3.5 server. More recent servers seem to be somewhat more consistent in the value returned for "hot" executions, although there is may still be a substantial difference between the first "cold" execution of a query and subsequent "hot" executions of the same query.

8. Both the `pg_catalog` schema and the `information_schema` schema contain meta-data describing the content of a database. Why do we need two schemas to do essentially the same task, and how are they related?

Answer:

We don't actually need two schemas; we have two schemas as a result of history. The `information_schema` schema is an SQL standard that was developed as part of the SQL-92 standard. Most DBMSs existed before that standard and had already developed their own catalog tables, which they retained as they were often integral to the functioning of the DBMS engine. In most DBMSs the `information_schema` is implemented as a collection of views on the native catalog schema.

If you want to take a look at the definitions of the `information_schema` views in PostgreSQL, log in to any database and try the following:

```
db=# set schema 'information_schema';
SET
db=# \dS
... list of views and tables ...
db=# \d+ views
... schema and definition for "information_schema.views" ...
... which contains meta-data about views in the database ...
```

9. Cross-table references (foreign keys) in the `pg_catalog` tables are defined in terms of `oid` attributes. However, examination of the the catalog table definitions (either via `\d` in `psql` or via the PostgreSQL documentation) doesn't show an `oid` in any of the lists of table attributes. To see this, try the following commands:

```
$ psql mydb
...
mydb=# \d pg_database
...
mydb=# \d pg_authid
```

Where does the `oid` attribute come from?

Answer:

Every tuple in PostgreSQL contains some "hidden" attributes, as well as the data attributes that were defined in the table's schema (i.e. its `CREATE TABLE` statement). The tuple header containing these attributes is described in section 54.5 Database Page Layout of the PostgreSQL documentation. All tuples have attributes called `xmin` and `xmax`, used in the implementation of multi-version concurrency control. In fact the `oid` attribute is optional, but all of the `pg_catalog` tables have it. You can see the values of the hidden attributes by explicitly naming the attributes in a query on the table, e.g.

```
select oid,xmin,xmax,* from pg_namespace;
```

In other words, the "hidden" attributes are not part of the SQL `*` which matches *all* attributes in the table.

10. Write an SQL view to give a list of table names and table oid's from the public namespace in a PostgreSQL database.

Answer:

```
create or replace view Tables
as
select r.oid, r.relname as tablename
from   pg_class r join pg_namespace n on (r.relnamespace = n.oid)
```

```
where  n.nspname = 'public' and r.relkind = 'r'
;
```

11. Using the tables in the `pg_catalog` schema, write a function to determine the location of a table in the filesystem. In other words, provide your own implementation of the built-in function: `pg_relation_filepath(`*`TableName`*`)`. The function should be defined and behave as follows:

```
create function tablePath(tableName text) returns text
as $$ ... $$ language plpgsql;

mydb=# select tablePath('myTable');
          tablepath
-----------------------------
 PGDATA/base/2895497/2895518
mydb=# select tablePath('ImaginaryTable');
            tablepath
-------------------------------
 No such table: imaginarytable
```

Start the path string with `PGDATA/base` if the `pg_class.reltablespace` value is `0`, otherwise use the value of `pg_tablespace.spclocation` in the corresponding `pg_tablespace` tuple.

Answer:

```
create or replace function tablePath(tableName text) returns text
as $$
declare
        _nloc text;
        _dbid integer;
        _tbid integer;
        _tsid integer;
begin
        select r.oid, r.reltablespace into _tbid, _tsid
        from   pg_class r
                join pg_namespace n on (r.relnamespace = n.oid)
        where  r.relname = tableName and r.relkind = 'r'
                and n.nspname = 'public';
        if (_tbid is null) then
                return 'No such table: '||tableName;
        else
                select d.oid into _dbid
                from   pg_database d
```

```
                where  d.datname = current_database();
                if (_tsid = 0) then
                        _nloc := 'PGDATA/data';
                else
                        select spcname into _nloc
                        from   pg_tablespace
                        where  oid = _tsid;
                        if (_nloc is null) then
                                _nloc := '???';
                        end if;
                end if;
                return _nloc||'/'||_dbid::text||'/'||_tbid::text;
        end if;
end;
$$ language plpgsql;
```

12. Write a PL/pgSQL function to give a list of table schemas for all of the tables in the public namespace of a PostgreSQL database. Each table schema is a text string giving the table name and the name of all attributes, in their definition order (given by `pg_attribute.attnum`). You can ignore system attributes (those with `attnum < 0`). Tables should appear in alphabetical order.

    The function should have following header:

    ```
    create or replace function tableSchemas() returns setof text ...
    ```

    and is used as follows:

    ```
    uni=# select * from tableschemas();
                                     tableschemas
    ---------------------------------------------------------------------------------
     assessments(item, student, mark)
     courses(id, code, title, uoc, convenor)
     enrolments(course, student, mark, grade)
     items(id, course, name, maxmark)
     people(id, ptype, title, family, given, street, suburb, pcode, gender, birthday, country)
    (5 rows)
    ```

    Answer:

    This function makes use of the `tables` view defined in Q6.

    ```
    create or replace function tableSchemas() returns setof text
    as $$
    ```

```
declare
        tab record; att record; ts text;
begin
        for tab in
                select * from tables order by tablename
        loop
                ts := '';
                for att in
                        select * from pg_attribute
                        where  attrelid = tab.oid and attnum > 0
                        order  by attnum
                loop
                        if (ts <> '') then ts := ts||', '; end if;
                        ts := ts||att.attname;
                end loop;
                ts := tab.tablename||'('||ts||')';
                return next ts;
        end loop;
        return;
end;
$$ language plpgsql;
```

And, just for fun, a version that uses the `information_schema` views, and, in theory, should be portable to other DBMSs that implement these views.

```
create or replace function tableSchemas2() returns setof text
as $$
declare
        tab record; att record; ts text;
begin
        for tab in
                select table_catalog,table_schema,table_name
                from   information_schema.tables
                where  table_schema='public' and table_type='BASE TABLE'
                order  by table_name
        loop
                ts := '';
                for att in
                        select c.column_name
                        from   information_schema.columns c
                        where  c.table_catalog = tab.table_catalog
                                and c.table_schema = tab.table_schema
                                and c.table_name = tab.table_name
```

```
                        order  by c.ordinal_position
                loop
                        if (ts <> '') then ts := ts||', '; end if;
                        ts := ts||att.column_name;
                end loop;
                ts := tab.table_name||'('||ts||')';
                return next ts;
        end loop;
        return;
end;
$$ language plpgsql;
```

13. Extend the function from the previous question so that attaches a type name to each attribute name. Use the following function to produce the string for each attribute's type:

```
create or replace function typeString(typid oid, typmod integer) returns text
as $$
declare
        typ text;
begin
        typ := pg_catalog.format_type(typid,typmod);
        if (substr(typ,1,17) = 'character varying')
        then
                typ := replace(typ, 'character varying', 'varchar');
        elsif (substr(typ,1,9) = 'character')
        then
                typ := replace(typ, 'character', 'char');
        end if;
        return typ;
end;
$$ language plpgsql;
```

The first argument to this function is a `pg_attribute.atttypid` value; the second argument is a `pg_attribute.atttypmod` value. (Look up what these actually represent in the PostgreSQL documentation).

Use the same function header as above, but this time the output should look like (for the first three tables at least):

```
 assessments(item:integer, student:integer, mark:integer)
 courses(id:integer, code:char(8), title:varchar(50), uoc:integer, convenor:integer)
 enrolments(course:integer, student:integer, mark:integer, grade:char(2))
```

Answer:

```
create or replace function tableSchemas() returns setof text
as $$
declare
        t record; a record; ts text;
begin
        for t in
                select * from tables order by tablename
        loop
                ts := '';
                for a in
                        select * from pg_attribute
                        where  attrelid = t.oid and attnum > 0
                        order  by attnum
                loop
                        if (ts <> '') then ts := ts||', '; end if;
                        ts := ts||a.attname||':'||typeString(a.atttypid,a.atttypmod);
                end loop;
                ts := t.tablename||'('||ts||')';
                return next ts;
        end loop;
        return;
end;
$$ language plpgsql;

create or replace function typeString(typid oid, typmod integer) returns text
as $$
declare
        tname text;
begin
        tname := format_type(typid,typmod);
        tname := replace(tname, 'character varying', 'varchar');
        tname := replace(tname, 'character', 'char');
        return tname;
end;
$$ language plpgsql;
```

Note that `format_type()` is a built-in function defined in the PostgreSQL documentation in section 9.23. System Information Functions

14. The following SQL syntax can be used to modify the length of a `varchar` attribute.

```
alter table TableName alter column ColumnName set data type varchar(N);
```

where *N* is the new length.

If PostgreSQL did not support the above syntax, suggest how you might be able to achieve the same effect by manipulating the catalog data.

Answer:

One possible approach would be:

```
update pg_attribute set atttypmod = N
where  attrelid = (select oid from pg_class where relname = 'TableName')
       and attname = 'ColumnName';
```

This is somewhat like what PostgreSQL does when you use the above `ALTER TABLE` statement.

Making the length longer causes no problems. What do you suppose might happen if you try to make the length shorter than the longest string value already stored in that column?

The `ALTER TABLE` statement rejects the update because some tuples have values that are too long for the new length. However, if you use the `UPDATE` statement, it changes the length, but the over-length tuples remain.