Exercises 02

Storage Management and Catalogs

For any questions that require you to use the PostgreSQL catalog, you should try to solve them by referring to the catalog section in the PostgreSQL documentation before looking at the solutions. An important aim of these exercises is for you to become familiar with the catalog.

1. What is the purpose of the storage management subsystem of a DBMS?

   Answer:

   The primary purpose of the storage manager is to organise the persistent storage of the DBMS's data and meta-data, typically on a disk device. The storage manager contains a mapping from user-level database objects (such as tables and tuples) to files and disk blocks. Its primary functions are performing the mapping from objects to files and transferring data between memory and disk.

2. Describe some of the typical functions provided by the storage management subsystem.

   Answer:

   Note that these functions are merely suggestive of the kinds of functions that might appear in a storage manager. They bear no relation to any real DBMS (and they are not drawn from the PostgreSQL storage manager, although similar kinds of functions will be found there). The function descriptions could have been less detailed, but I thought it was worth mentioning some typical data types as well.

   Some typical storage management functions ...

   - `RelnDescriptor *openRelation(char *relnName)`
     - initiates access to a named table/relation
     - determines which files correspond to the named table
     - sets up a data structure (`RelnDescriptor`) to manage access to those files
     - the data structure would typically contain file descriptors and a buffer
   - `DataBlock getPage(TableDescriptor *table, PageId pid)`
     - fetch the content of the `pid`th data page from the open `table`
     - `DataBlock` is a reference to a memory buffer containing the data
   - `Tuple getTuple(TableDescriptor *table, TupleID tid)`
     - fetch the content of the `pid`th tuple from the open `table`
     - `Tuple` is an in-memory data structure containing the values from the tuple
     - this function would typically determine which page contained the tuple, then call `getPage()` to retrieve the page, and finally extract the data values from the page buffer; it may also need to open other files and read e.g. large data values from them

   Other functions might include `putPage`, `putTuple`, `closeTable`, etc.

3. 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 ...
```

4. 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.

5. 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'
;
```

6. 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;
```

7. 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;
```

8. 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

9. **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.