COMP9315 19T2 DBMS Implementation

Prac Exercise 03

PostgreSQL Server Config and File Structures

Aims

This simple exercise aims to get you to:

- examine the configuration of your PostgreSQL servers
- start to understand the filesystem layout of PostgreSQL files
- start to understand the internal structure of PostgreSQL data files

You ought to get it done before the middle of week 3.

Exercise

PostgreSQL has a wide range of configuration parameters which are described in Chapter 18 of the PostgreSQL documentation. For the purposes of this lab, we are most interested in the configuration parameters related to resource usage (described in Section 18.4).

Most configuration parameters can be set by modifying the `$PGDATA/postgresql.conf` file and restarting the server. Many configuration parameters can also be set via command-line arguments to the `postgres` server when it is initially invoked. Note that you *cannot* set parameters if you invoke the server via the `pgs` script; `pgs` aims to simplify things by allowing few options and starting the server with the configuration specified in `postgresql.conf`. The standard PostgreSQL mechanism for starting the server is yet another script, called `pg_ctl` (see the `pg_ctl` section of the PostgreSQL documentation). The simplest way to invoke `pg_ctl` is one of:

```
$ pg_ctl start
server starting
$ pg_ctl stop
waiting for server to shut down.... done
server stopped
$ pg_ctl status
pg_ctl: server is running (PID: nnnnnn)
/srvr/YOU/pgsql/bin/postgres
```

The `pgs` script simply invokes `pg_ctl` to start a server, with some extra options:

```
$ pg_ctl -w start -l /srvr/YOU/pgsql/log
waiting for server to start...... done
server started
```

The `-l` option tells the PostgreSQL server which file to use to write its log messages. The log file is important, not only because it is where PostgreSQL writes error messages so that you can work out e.g. why your server wouldn't start, but also because it is where PostgreSQL writes statistical information about its performance (if requested).

The `-w` option tells `pg_ctl` to wait until the server has actually started properly before returning. If the server does not start properly, you will eventually receive a message like:

```
pg_ctl: could not start server
Examine the log output.
```

If the server fails to start, you should check your environment and the server setup (e.g. `$PGDATA/postgresql.conf`). Note that there are two aspects to consider for the environment: the contents of `/srvr/YOU/env` *and* the settings of the shell variables in your current window; the two should be consistent. A trouble-shooting guide for setting up your server appears at the bottom of Prac Exercise P01.

The primary function of the `pg_ctl` command is to invoke the `postgres` server. It can perform additional functions such as specifying the location of the log file (as we saw above) or passing configuration parameters to the server. To pass configuration parameters, you use the `-o` option and a single string containing all the server parameters. For example, the `-B` parameter to `postgres` lets you say how many shared memory buffers the server should use, and you could start `postgres` and get it to use just 16 buffers as follows:

```
$ pg_ctl start -o '-B 16' -l /srvr/YOU/pgsql/log
server starting
```

As a warm-up exercise, work out how many shared buffers the PostgreSQL server uses by default. (Hint: this is given in the `postgresql.conf` file in units of MB (not number of buffers); each buffer is 8KB long).

Answer:

The `shared_buffers` parameter controls this. The default value for this is 32MB (according to the PostgreSQL documentation). However, the value in the `postgresql.conf` file produced by `pgs` seems to be 128MB. If the value of `shared_buffers` in *N*MB, and the size of each buffer is 8KB, then the total number of buffers is given by the formula (*N**1024*1024)/8192. For 128MB, this gives 16384 buffers; for 32MB, this gives 4096 buffers.

Exercises

Start your PostgreSQL server as normal (i.e. don't change any configuration parameters) before getting started with the exercises.

Ex0: Load a new Test Database

Under the COMP9315 Pracs directory you'll find a new testing database. Create a new database to hold it, and load it up. There are two representations of the database available:

- as a PostgreSQL dump file
- as a pair of SQL files, one containing the schema and the other the data

The dump file is quicker to load, but not as "user-friendly" (i.e. not as readable) as the SQL files.

You create the database in the usual way:

```
$ createdb uni
```

I called the database `uni` because it contains (fake) data about a University. You can find out the database schema from the `schema.sql` file.

To load the database, use the following commands:

```
$ (psql uni -f /web/cs9315/19T2/pracs/p03/db.dump 2>&1) > load.out
$ grep ERR load.out
```

The first command loads the dump file and ensures that all output is written to a file called `load.out`. The second command checks for any error messages produced during the load. There may be an error message like

```
psql:/web/cs9315/19T2/pracs/p03/db.dump:16: ERROR:  language "plpgsql" already exists
```

You can ignore this. All it means is that your database already knew about the PL/pgSQL language. If there are any other errors, you should *not* ignore those, but instead try to work out what the problem is and fix it.

Ex1: Devise some Queries on the Test DB

The first thing to do with any database is to ensure that you understand what data is in it. Use `psql` (or some GUI tool, if you're using one) to explore the database. I've added a function that will give you counts of the number of tuples in each table:

```
uni=# select * from pop();
    table     | ntuples
-------------+---------
 assessments |   14098
 courses     |     980
 enrolments  |    3506
 items       |    3931
 people      |    1980
(5 rows)
```

You can look at the definition of the `pop()` (short for "population") either in the `pop.sql` file, or via `psql`'s `\df+` command.

Once you think you're familiar enough with the database, devise SQL queries to answer the following:

a. what is the largest staff/student id? (`People.id`)

Answer:

```
uni=# select max(id) from People;
 max
------
 5936
```

b. what is the earliest birthday of any person in the database? (`People.birthday`)

Answer:

```
uni=# select min(birthday) from People;
    min
------------
 1970-01-17
```

c. what is the maximum mark available for any assessment item? (`Items.maxmark`)

Answer:

```
uni=# select max(maxmark) from items;
 max
-----
  90
```

d. what assessment items are in each course and how many marks does each have? (`Courses.code,Items.name,Items.maxmarks`)

Answer:

```
uni=# select c.code, i.name, i.maxmark
uni-# from   Courses c, Items i
uni-# where  c.id = i.course;
   code    |     name      | maxmark
----------+--------------+---------
 ACCT1501 | Assignment 1 |      10
 ACCT1501 | Assignment 2 |      10
 ACCT1501 | Project      |      25
 ACCT1501 | Exam         |      55
 ACCT1511 | Assignment 1 |      15
 ACCT1511 | Assignment 2 |       5
 ACCT1511 | Assignment 3 |      15
 ACCT1511 | Exam         |      65
etc. etc., for 3931 items
```

e. how many students are enrolled in each course? (`Courses.code,count(Enrolments.student)`)

Answer:

```
uni=# select c.code, count(e.student)
uni-# from   Courses c, Enrolments e
uni-# where  c.id = e.course
uni-# group  by c.code
uni-# order  by c.code;
   code    | count
----------+-------
 ACCT1501 |     7
 ACCT1511 |     2
 ACCT2522 |     2
 ACCT3563 |     3
etc. etc., for 913 courses
```

If you leave out the `order by` you should get the same *set* of results, but not necessarily in the same order.

f. check that each student's assessment marks add up to the final mark for each course (`Course.code,People.name,Enrolments.mark,sum(Assessment.marks)`)

Answer:

```
uni=# select c.code, p.family||', '||p.given as name, e.mark, sum(a.mark)
uni-# from   People p, Courses c, Enrolments e, Items i, Assessments a
uni-# where  p.id = e.student and e.course = c.id and i.course = c.id
uni-#        and a.student = p.id and a.item = i.id
uni-# group  by c.code, p.family, p.given, e.mark
uni-# order  by c.code, p.family;
   code    |                    name                     | mark | sum
----------+---------------------------------------------+------+-----
 ACCT1501 | Agster, Yvan Marie                          |   68 |  68
 ACCT1501 | Bland, Daryl Robert                         |   56 |  56
 ACCT1501 | Fadaghi, Mundeep Singh                      |   47 |  47
 ACCT1501 | Gafen, Andrei                               |   56 |  56
 ACCT1501 | Mcnulty, Abu Rifat                          |   77 |  77
 ACCT1501 | Nugent, Daina                               |   55 |  55
etc. etc., for 3506 tuples
```

For the first four queries above, think about and describe the patterns of access to the data in the tables that would be required to answer them.

Answer:

a. Requires all `People.id` values to be accessed; potentially this would need a scan over all tuples in the relation, hence all pages would need to be read. However, there's an index on the `People.id` attribute (PostgreSQL makes a B-tree index on all primary keys) which contains all of the `People.id` values. In theory, the system could determine the largest value simply by looking at the index. How could we work out whether it was doing a full table scan or simply reading the index?

b. Requires all `People.birthday` values to be accessed. Since there is no index on birthdays, this will definitely require PostgreSQL to read all of the tuples/pages in the `People` table.

c. Requires all `Items.maxmark` values to be accessed; since this is not a key attribute, there are no indexes on it and all tuples/pages from the `Items` will need to be read.

d. Requires a join on the `Courses` and `Items` table; each tuple in each table will need to be accessed, possibly multiple times; in the worst-case scenario, we would read the `Courses` table once and read the entire `Items` table for each page in the `Courses` table.

Ex2: Explore the Files of the Test DB

Now that you've used the database, let's take a look at how the data is stored in the file system. All data is for a given database is stored under the directory (folder):

```
$PGDATA/base/OID
```

where `$PGDATA` is the location of the PostgreSQL data directory as set in your `env` file, and the*`OID`* is the unique internal id of the database from the `pg_database` table. Work out, using the PostgreSQL catalog, which directory corresponds to your newly-created database. (Hint: the `pg_database` table will help here. Also, `psql`'s `\dS` command will tell you the names of all catalog tables).

Answer:

The following SQL query will help you work out what is the *`OID`* for your database:

```
select oid, datname from pg_database
```

This will give you a list of databases, including `template1`, `template0` and `postgres`, each with an associated OID. There should also be a tuple for your `uni` database; the OID value should also appear as the name of a directory in `pgsql/data/base/`.

Change into the relevant directory and run the `ls` command. This will show dozens of files. Most of these files contain local data from system catalog tables, while others contain your `uni` data. Recall from lectures that data files associated with a table are named after the *`OID`* of that table. Use the PostgreSQL catalog to work out which files correspond to your tables.

Answer:

The following SQL query will do it:

```
select c.oid,c.relname
from   pg_class c, pg_namespace n
where  c.relkind='r' and c.relnamespace=n.oid and n.nspname='public';
```

If you omit the last condition in the query, you'll get all of the system tables as well, which will help you work out what all of the other files in the directory are.

All of the data files in this directory are in binary format, so you can't read them with a text editor or the standard Unix file pagers (like `more` and `less`). Sometime, however, you can get *some* information from a binary file via the `strings` command, which prints any text-strings that it finds in the file. Try this on the file corresponding to the `Courses` table and you should get a list of course codes and course titles, with a few "junk" characters. Since this generates a lot of output, you might want to use something like the following command:

```
$ strings OID_of_Courses_data_file | less
BENV2254;Theories of Colour and Light
BENV2228?C20 Arch:Modernity-Deconstruc.
BENV22241Architectural Studies 3
etc. etc. etc.
```

Note that you won't necessarily see *exactly* the output shown above. The order that tuples are inserted into a page depends on many factors that vary from system to system. What you are guaranteed to see are some strings containing data relevant to courses.

An alternative way to examine binary data files is via the Unix `od` command (read the `man` entry if you don't know what it does). Examine the files corresponding to the `People` table and the `Assessments` table to see if you can observe the data they contain and also to see if you can work out how the data is laid out within the pages of the file. You can can get assistance with understanding the intra-page data layout from the source code files:

```
/srvr/YOU/postgresql-11.3/src/include/storage/bufpage.h
/srvr/YOU/postgresql-11.3/src/backend/storage/page/bufpage.c
```

You'll probably notice some other files with similar *OID*s to the data files, and other files with the same *OID*s but with added suffixes. Suggest what might be contained in these files. (Searching for suffixes in the source code might help for those files with suffixes).

Answer:

The files with `_fsm` suffixes contain free space maps which indicate where space is available in the data file (see Section 53.3 of the PostgreSQL documentation).

The files with `_vm` suffixes contain visibility maps which indicate pages that contain tuples visible to all active transactions; this allows vacuuming to be optimised (see Section 53.4 of the PostgreSQL documentation).

The files with OIDs close to those of the table data files, but without any free space maps are index files (each table has an index on its defined primary key).

While you're examining the data files, return to `psql` and write a query to print the number of data pages in each relation. This is a simple modification of the query above to get the table OIDs.

Answer:

```
uni=# select c.relname,c.relpages
uni-# from   pg_class c, pg_namespace n
uni-# where  c.relkind='r' and c.relnamespace=n.oid and n.nspname='public';
   relname    | relpages
-------------+----------
 assessments |       70
 people      |       27
 courses     |        9
 enrolments  |       19
 items       |       26
```

Once you've got the page counts in the catalog, check that they're consistent with the file sizes in the directory for the `uni` database (assuming an 8KB page size).

Answer:

A quick example of how to do this:

```
$ psql uni
psql (11.3)
Type "help" for help.

uni=# select oid,relpages from pg_class where relname='Courses';
 oid | relpages
-----+----------        # Ooops ... PostgreSQL uses all-lower-case table names internally
(0 rows)

uni=# select oid,relpages from pg_class where relname='courses';
  oid  | relpages
-------+----------
 NNNNN |        9       # NNNNN is the oid of the Courses table
(1 row)                 # and is also the name of its data file
                        # This also tells us that the table has 9 * 8KB pages
uni=# \q
$ ls -l NNNNN
-rw------- 1 YOU YOU 73728 2011-08-03 14:04 NNNN
# 73728 is the number of bytes in the file NNNNN
$ bc -l
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
9 * 8192
73728
# type control-d to exit the bc command
```

Try this on some other tables. Try to explain any anomalies you find.

End of Prac

Let me know via the forums, or come to a consultation if you have any problems with this exercise ... *jas*