Lab Exercise 01

PostgreSQL: a client-server RDBMS

Aims

This exercise aims to get you to:

- install your PostgreSQL database server at CSE
- create, populate and examine a very small database

Prerequisites

For this lab, you will need to:

- have a laptop/PC running Windows/MacOS/Linux, etc.
- a good network connection to connect CSE computers

To attend a lab, you need to:

- bring your own laptop, make sure it is fully charged. (there may be charging ports in the lab room, you have to check it yourself)
- or borrow a laptop from CSE

The instructions are quite detailed and should be helpful for you. Please read them carefully *before* jumping into entering commands into your computer.

Important note: To do this lab, you will need to first connect to CSE using VLab following the instructions in Connecting to CSE using VLAB

Background

PostgreSQL runs on most platforms, but the installation instructions will be different for each one. There are downloadable binaries for many platforms, but they place restrictions on how your server is configured. Other installations come with a GUI front-end; I think this moves you too far away from the server, and simply gets in the way ... but maybe that's just me.

My general advice on using a server-side software is 'TRY TO GET USED TO USING COMMAND LINE INTERFACE. FAST/EFFICIENT/FLEXIBLE/KEEPS YOUR BRAIN ACTIVE' !!

This Lab Exercise describes how to set PostgreSQL up on your CSE account. You can still work from home by accessing your server through VLab or `ssh`.

If you *do* want to install PostgreSQL on your home machine, you'll need to work out how to do that yourself. There are plenty of online resources describing how to do this for different operating systems (type "install postgresql" at Google). It probably doesn't matter if you use a slightly different version at home, since we'll be using a subset of SQL and PLpgSQL that hasn't changed for a while; any version around 11/12/13/14/15 is ok for the lab work in this course.

The lab work for the assignments can be carried out on a CSE machine called `d.cse.unsw.edu.au` . You run your own PostgreSQL server on this machine and are effectively the database administrator of this server. This machine has been configured to run large numbers* of PostgreSQL servers.

> * Note: "large numbers" is around 300. If you leave your work to the last minute, and find 500 other students all trying to run PostgreSQL on `d` , performance will be sub-optimal.

You should *NOT* use other CSE servers such as `wagner` or `vx05` for running PostgreSQL; if you do, your PostgreSQL server will most likely be terminated automatically not long after it starts.

> Reminder: You should *always* test your work on the CSE machines before you submit assignments, since that's where we'll be running our tests to award your marks.

For brevity, we'll refer to " `d.cse.unsw.edu.au` " as " `d` " in the rest of this document.

What `d` provides is a large amount of storage and compute power that is useful for students studying databases. You should have access to `d` because you're enrolled in COMP9311. You can access `d` using either `ssh` or VLab.

In the examples below, we have used the `$` sign to represent the prompt from the command interpreter (shell). In fact, the prompt may look quite different on your machine (e.g., it may contain the name of the machine you're using, or your username, or the current directory name). All of the things that the computer types are in `this font` . The commands that **you** are supposed to type are in **`this bold font`** . Some commands use `$USER` as a placeholder for your CSE username (e.g., z1234567). When you type in the commands, you should replace `$USER` with your own username (e.g., z1234567).

Exercises

Stage 0: Basic Linux Commands

Before you begin working with PostgreSQL on `d.cse.unsw.edu.au` , it's important to familiarize yourself with some basic Linux commands. These commands will help you navigate and interact with the Linux command-line interface.

1. Opening a Terminal

To get started, open a terminal or command prompt on your local machine. This is where you'll enter Linux commands.

2. Navigation

- `cd` (Change Directory): Use this command to navigate between directories. For example:

  ```
  $ cd /path/to/directory
  ```

- `ls` (List Files and Directories): To view the contents of a directory, use:

  ```
  $ ls
  ```

- `pwd` (Print Working Directory): To display the current directory's path, type:

  ```
  $ pwd
  ```

3. File Operations

- `touch` (Create a New File): You can create an empty file with:

  ```
  $ touch filename.txt
  ```

- `mkdir` (Create a New Directory): To create a new directory, use:

  ```
  $ mkdir directory_name
  ```

- `cp` (Copy): Use this command to copy files or directories:

  ```
  $ cp source destination
  ```

- `mv` (Move/Rename): Move or rename files and directories with:

  ```
  $ mv source destination
  ```

- `rm` (Remove/Delete): To delete files or directories, use:

  ```
  $ rm filename
  ```

4. Working with Text Files

- `cat` (Concatenate and Display): To display the content of a text file, use:

  ```
  $ cat filename.txt
  ```

- `gedit` (Text Editor): To edit a text file, you can use the `gedit` text editor:

  ```
  $ gedit filename.txt
  ```

  To save changes in `gedit` , click the "Save" button in the graphical interface.

5. Executing Bash Scripts

In Linux, you can execute Bash scripts to automate tasks or perform a series of commands. There are two ways to run a Bash script: by making it executable and running it directly, or by using the `bash` command.

1. Create or obtain a Bash script file (e.g., `myscript.sh`).
2. **Option 1:** Making the Script Executable
    1. Make the script file executable using the `chmod` command. Replace `myscript.sh` with the name of your script:

       ```
       $ chmod +x myscript.sh
       ```

    2. Run the script using the following syntax, replacing `myscript.sh` with the name of your script:

       ```
       $ ./myscript.sh
       ```

3. **Option 2:** Using the `bash` Command
    1. Execute the script directly using the `bash` command, specifying the script's filename:

       ```
       $ bash myscript.sh
       ```

Executing a Bash script allows you to automate tasks, configure your environment, or perform various actions without manually typing each command.

Now that you have a basic understanding of these Linux commands, you're ready to proceed to Stage 1 and set up PostgreSQL on `d.cse.unsw.edu.au` .

Stage 1: Getting started on d.cse.unsw.edu.au

Log in to `d`. If you're not logged into `d` nothing that follows will work.

The `d` server has two diffreent names and you need to use the names in an appropriate context before you can login.

Within the CSE network (e.g. logged in to `vx02`), you access `d` as:

```
$ ssh nw-syd-vxdb
```

Outside the CSE network, you access `d` as:

```
$ ssh d.cse.unsw.edu.au
```

You can log into `d` from a command-line (shell) window on any CSE machine (including `vlab`) via the command:

```
$ ssh nw-syd-vxdb
```

If you're doing this exercise from home, you can use any `ssh` client, but you'll need to refer to `d` via its fully-qualified name:

```
$ ssh YourZID@d.cse.unsw.edu.au
```

From home, an alternative is to use VLab. This requires a VNC client (e.g. TigerVNC). Use the VNC server

```
d.cse.unsw.edu.au:5920
```

You can check whether you're actually logged in to `d` by using the command:

```
$ hostname
nw-syd-vxdb
```

Your home directory at CSE is directly accessible from `d`. Run the `ls` command to check that you are indeed in your CSE home directory.

The first time you log in to `d`, it automatically creates a directory to hold your databases:

```
$ ls -al /localstorage/$USER
```

(Don't forget to replace $USER with your own username)

This directory is initially empty, but we're about to put the files for a PostgreSQL server into it.

Stage 2: Setting up your environment

PostgreSQL requires precise configuration settings to ensure it functions correctly. To get started, assuming that you have logged into `d`, you can run the following commands to set up your PostgreSQL Server. First, you visit the root directory represented by `~`, and then execute the bash script `setup.sh`.

```
$ cd ~
$ bash /home/cs9311/web/24T2/lab/01/setup.sh
```

You will see some outputs like this.

```
initializing the postgresql setup
The files belonging to this database system will be owned by user ZID.
This user must also own the server process.

The database cluster will be initialized with locale "C.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

creating directory /localstorage/ZID/pgsql/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Australia/Sydney
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok
```

```
initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D /localstorage/ZID/pgsql/data -l logfile start

Finished modifying the config file
```

After this, the postgreSQl is installed in the directory `/localstorage/$USER/` , where $USER is your zID. You can check this by listing what's inside this directory:

```
$ ls /localstorage/$USER/
```

You should see outputs like this.

```
$ env  pgsql
```

"env" is a directory for environment-specific configurations, and "pgsql" is the PostgreSQL data directory for storing database files and configurations. Note that for your current configurations to take effect, you need to log out of `d.cse.unsw.edu.au` via the command `exit` .

```
$ exit
```

In case you want to delete the postgreSQL server and start from scratch, run the following commands:

```
$ cd ~
$ bash /home/cs9311/web/24T2/lab/01/clean.sh
```

Stage 3: Using your PostgreSQL Server

In this section, we assume that you have completed Stage 2 and now have a directory `/localstorage/$USER/` for PostgreSQL on `d`

When you want to do some work with PostgreSQL: login to `d` , start your server, do your work, and then stop the server before logging off.

Do not leave your PostgreSQL server running while you are not using it.

A typical session with PostgreSQLwould begin with you logging in to the `d` . You would then do something like ...

```
$ ssh nw-syd-vxdb
  ... logging into the course server ...
$ p1
  ... start the PostgreSQL server ...
$ psql SomeDatabase
  ... work with a database ...
$ p0
  ... stop the PostgreSQL server ...
```

Note that `p1` and `p0` are abbreviations defined in the `env` . `p1` starts the PostgreSQL server and `p0` stops the PostgreSQL server.

After using `p1` , you can check whether your server is running via the command:

```
$ psql -l
```

Note: `l` is lower-case L, not the digit 1.

Try starting, checking, and stopping the server a few times.

Things occasionally go wrong, and knowing how to deal with them will save you lots of time. There's a discussion of common problems at the end of this document; make sure that you read and understand it.

Once your PostgreSQL server is running, you can access your PostgreSQL databases via the `psql` command. You normally invoke this command by specifying the name of a database, e.g.

```
$ psql MyDatabase
```

If you type `psql` command without any arguments, it assumes that you are trying to access a database with the same name as your login name. Since you probably won't have created such a database, you're likely to get a message like:

```
psql: FATAL:  database "${USER}" does not exist
```

You will get a message like this any time that you try to access a database that does not exist.

If you're not sure what databases you have created, `psql` can tell you via the `-l` option

```
$ psql -l
```

If you run this command now, you ought to see output that looks like:

```
SET
                               List of databases
   Name    | Owner | Encoding | Collate    |    Ctype    | Access privileges
-----------+-------+----------+------------+-------------+-------------------
 postgres  | jas   | UTF8     | C          | en_AU.UTF-8 |
 template0 | jas   | UTF8     | C          | en_AU.UTF-8 | =c/jas           +
           |       |          |            |             | jas=CTc/jas
 template1 | jas   | UTF8     | en_US.utf8 | en_US.utf8  |
(3 rows)
```

Of course, it will be *your* username, and not `jas`.

Note that PostgreSQL commands like `psql` and `createdb` are a lot noisier than normal Linux commands. In particular, they all seem to print `SET` when they run; you can ignore this. Similarly, if you see output like `INSERT 0 1`, you can ignore that as well.

The above three databases are created for use by the PostgreSQL server; you should not modify them. At this stage, you don't need to worry about the contents of the other columns in the output. As long as you see at least three databases when you run the `psql -l` command, it means that your PostgreSQL server is up and running ok.

Note that you are the administrator for your PostgreSQL server (add "database administrator" to your CV) and you can create as many databases as you like, within the limits of your disk quota.

From within `psql`, the fact that you are an administrator is indicated by a prompt that looks like

```
dbName=#
```

rather than the prompt for database users

```
dbName=>
```

which you may have seen in textbooks or notes.

Note that you can only access databases created as above while you're logged into `d`. In other words, you must run the `psql` command on `d`.

Note that the **only** commands that you should run on `db.cse` are the commands to start and stop the server, the `psql` command to start an interactive session with a database, and the other PostgreSQL clients such as `createdb`. Do not run other processes such as web browsers or drawing programs or games on `d`. Text editors are OK; VScode is not.

All of the PostgreSQL client applications are documented in the PostgreSQL manual, in the "PostgreSQL Client Applications" section. While there are quite a few PostgreSQL client commands, `psql` will be the one that you will mostly use.

Mini-Exercise: a quick way to check whether your PostgreSQL server is running is to try the command:

```
$ psql -l
```

Try this command now.

If you get a response like:

```
psql: command not found
```

then you haven't set up your environment properly; `source` the `env` file.

If you get a response like:

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "....s.PGSQL.5432"?
```

then the server isn't running.

If you get a list of databases, like the example above, then this means your server is running ok and ready for use.

Cleaning up

After you've finished a session with PostgreSQL, it's essential that you shut your PostgreSQL server down (to prevent overloading `d`). Recall that your this via the command

```
$ p0
```

PostgreSQL generates log files that can potentially grow quite large. If you start your server using `p1` , the log file is called

```
/localstorage/$USER/pgsql/data/log
```

It would be worth checking every so often to see how large it has become. To clean up the log, simply stop the server and remove the file. Note: if you remove the logfile while the server is running, you may not remove it at all; its link in the filesystem will be gone, but the disk space will continue to be used and grow until the server stops.

> **Mini-Exercise:** Try starting and stopping the server a few times, and running `psql` both when the server is running and when it's not, just to see the kinds of messages you'll get.

Exercise #1: Making a database

Start by logging in to `d` and setting the environment.

Once the PostgreSQL server is running, try creating a database by running the command:

```
$ createdb mydb
```

which will create the database, or give an error message if it can't create it for some reason. (A typical reason for failure would be that your PostgreSQL server is not running.)

Now use the `psql -l` command to check that the new database exists.

You can access the database by running the command:

```
$ psql mydb
```

which should give you a message like

```
SET
psql (13.3 (Debian 13.3-1))
Type "help" for help.

mydb=#
```

Note that `psql` lets you execute two kinds of commands: SQL queries and updates, and `psql` “meta”-commands. The `psql` “meta”-commands allow you to examine the database schema, and control various aspects of `psql` itself, such as where it writes its output and how it formats tables.

Getting back to the `psql` session that you just started, the `mydb` database is empty, so there's not much you can do with it. The `\d` (describe) command allows you to check what's in the database. If you type it now, you get the unsurprising response

```
mydb=# \d
No relations found.
```

About the only useful thing you can do at the moment is to quit from `psql` via the `\q` command.

```
mydb=# \q
$ ... now waiting for you to type Linux commands ...
```

Note: it is common to forget which prompt you're looking at and sometimes type Linux commands to `psql` or to type SQL queries to the Linux shell. It usually becomes apparent fairly quickly what you've done wrong, but can initially be confusing when you think that the command/query is not behaving as it should. Here are examples of making the above two mistakes:

```
$ ... Linux command interpreter ...
$ select * from table;
-bash: syntax error near unexpected token `from'
$ psql mydb
... change context to PostgreSQL ...
mydb=# ls -l
mydb-# ... PostgreSQL waits for you to complete what it thinks is an SQL query ...
mydb-# ;    ... because semi-colon finishes and then executes an SQL query ...
ERROR:  syntax error at or near "ls"
```

```
LINE 1: ls -l
        ^
mydb=# \q
$ ... back to Linux command interpreter ...
```

Exercise #2: Populating a database

Once the `mydb` database exists, the following command will create the schemas (tables) and populate them with tuples:

```
$ psql mydb -f /home/cs9311/web/24T2/lab/01/mydb.sql
```

Note that this command produces quite a bit of output, telling you what changes it's making to the database. The output should look like:

```
SET
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

The lines containing `CREATE TABLE` are, obviously, related to PostgreSQL creating new database tables (there are four of them). The lines containing `INSERT` are related to PostgreSQL adding new tuples into those tables.

Clearly, if we were adding hundreds of tuples to the tables, the output would be very long. You can get PostgreSQL to stop giving you the `INSERT` messages by using the `-q` option to the `psql` command.

PostgreSQL's output can be verbose during database loading. If you want to ignore everything *except* error messages, you could use a command like:

```
$ psql mydb -f /home/cs9311/web/24T2/lab/01/mydb.sql 2>&1 | grep ERROR
```

If you don't understand the fine details of the above, take a look at the documentation for the Linux/Unix shell.

The `-f` option to `psql` tells it to read its input from a file, rather than from standard input (normally, the keyboard). If you look in the `mydb.sql` file, you'll find a mix of table (relation) definitions and statements to insert tuples into the database. We don't expect you to understand the contents of the file at this stage.

If you try to run the above command again, you will generate a heap of error messages, because you're trying to insert the same collection of tables and tuples into the database, when they've already been inserted.

Note that the tables and tuples are now permanently stored on disk. If you switch your PostgreSQL server off, when you restart it the contents of the `mydb` database will be available, in whatever state you left them from the last time you used the database.

Exercise #3: Examining a database

One simple way to manipulate PostgreSQL databases is to use the `psql` command. A useful way to start exploring a database is to find out what tables it has. We saw before that you can do this with the \d (describe) command. Let's try that on the newly-populated `mydb` database.

```
mydb=# \d
         List of relations
 Schema |   Name     | Type  | Owner
```

```
--------+-----------+-------+-------
 public | courses   | table | $USER
 public | enrolment | table | $USER
 public | staff     | table | $USER
 public | students  | table | $USER
(4 rows)
```

You can ignore the `Schema` column for the time being. The `Name` column tells you the names of all tables (relations) in the current database instance. The `Type` column is obvious, and, you may think, unnecessary. It's there because `\d` will list all objects in the database, not just tables; it just happens that there are only tables in this simple database. The `Owner` should be your username, for all tables.

One thing to notice is that the table names are all in lower-case, whereas in the `mydb.sql` file, they had an initial upper-case letter. The SQL standard says that case does not matter in unquoted identifiers and so `Staff` and `staff` and `STAFF` and even `StAfF` are all equivalent. To deal with this, PostgreSQL simply maps *identifiers* into all lower case internally. You can still use `Staff` when you're typing in SQL commands; it will be mapped automatically before use.

> There are, however, advantages to using all lower case whenever you're dealing with `psql` . For one thing, it means that you don't have to keep looking for the shift-key. More importantly, `psql` provides table name and field name completion (you type an initial part of a table name, then type the TAB key, and `psql` completes the name for you if it has sufficient context to determine this unambiguously), but it only works when you type everything in lower case. The `psql` interface has a number of other features (e.g. history, command line editing) that make it very nice to use.

If you want to find out more details about an individual table, you can use:

```
mydb=# \d Staff
             Table "public.staff"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 userid   | character varying(10) | not null
 name     | character varying(30) |
 position | character varying(20) |
 phone    | integer               |
Indexes:
    "staff_pkey" PRIMARY KEY, btree (userid)
Referenced by:
    TABLE "courses" CONSTRAINT "courses_lecturer_fkey" FOREIGN KEY (lecturer) REFERENCES staff(userid)
```

As you can see, the complete name of the table is `public.staff` , which includes the schema name. PostgreSQL has the notion of a "current schema" (which is the schema called `public` , by default), and you can abbreviate table names by omitting the current schema name, which is what we normally do. The types of each column look slightly different to what's in the `mydb.sql` file; these are just PostgreSQL's internal names for the standard SQL types in the schema file. You can also see that the `userid` field is not allowed to be null; this is because it's the primary key (as you can see from the index description) and primary keys may not contain null values. The index description also tells you that PostgreSQL has built a B-tree index on the `userid` field.

The final line in the output tells you that one of the other tables in the database (`Courses`) has a foreign key that refers to the primary key of the `Staff` table, which you can easily see by looking at the `mydb.sql` file. This is slightly useful for a small database, but becomes extremely useful for larger databases with many tables.

The next thing we want to find out is what data is actually contained in the tables. This requires us to use the SQL query language, which you may not know yet, so we'll briefly explain the SQL statements that we're using, as we do them.

We could find out all the details of staff members as follows:

```
mydb=# select * from Staff;
  userid  |     name      |    position     | phone
----------+---------------+-----------------+-------
 jingling | Jingling Xue  | Professor       | 54889
 jas      | John Shepherd | Senior Lecturer | 56494
 andrewt  | Andrew Taylor | Senior Lecturer | 55525
(3 rows)
```

The SQL statement says, more or less, "tell me everything (*) about the contents of the `Staff` table". Each row in the output below the heading represents a tuple in the table.

Note that the SQL statement ends with a semi-colon. The meta-commands that we've seen previously didn't require this, but SQL statements can be quite large, and so, to allow you to type them over several lines, the system requires you to type a semi-colon to mark the end of the SQL statement.

If you forget to put a semi-colon, the prompt changes subtly:

```
mydb=# select * from Staff
```

```
mydb-#
```

This is PostgreSQL's way of telling you that you're in the middle of an SQL statement and that you'll eventually need to type a semi-colon. If you then simply type a semi-colon to the second prompt, the SQL statement will execute as above.

Mini-Exercise: find out the contents of the other tables.

Here are some other SQL statements for you to try out. You don't need to understand their structure yet, but they'll give you an idea of the kind of capabilities that the SQL language offers.

- Which students are studying for a CS degree (3778)?

  ```
  select * from Students where degree=3778;
  ```

- How many students are studying for a CS degree?

  ```
  select count(*) from Students where degree=3778;
  ```

- Who are the professors?

  ```
  select * from Staff where position ilike '%professor%';
  ```

- How many students are enrolled in each course?

  ```
  select course,count(*) from Enrolment group by course;
  ```

- Which courses is Andrew Taylor teaching?

  ```
  select c.code, c.title
  from   Courses c, Staff s
  where  s.name='Andrew Taylor' and c.lecturer=s.userid;
  ```

  or

  ```
  select c.code, c.title
  from   Courses c join Staff s on (c.lecturer=s.userid)
  where  s.name='Andrew Taylor';
  ```

The last query is laid out as we normally lay out more complex SQL statements: with a keyword starting each line, and each clause of the SQL statement starting on a separate line.

Try experimenting with variations of the above queries.

Sorting out Problems

It is very difficult to diagnose problems with software over email, unless you give sufficient details about the problem. An email that's as vague as “My PostgreSQL server isn't working. What should I do?”, is basically useless. Any email about problems with software should contain details of

- what you were attempting to do
- precisely what commands you used
- exactly what output you got

One way to achieve this is to copy-and-paste the last few commands and responses into your email.

Alternatively, you should come to a consultation where we can work through the problem via screen sharing (which is usually very quick).

Can't shut server down?

When you use `p0` to shut down your PostgreSQL server, you'll observe something like:

```
$ p0
waiting for server to shut down....
```

Dots will keep coming until the server is finally shut down, at which point you will see:

```
$ p0
waiting for server to shut down........ done
server stopped
```

Sometimes, you'll end up waiting for a long time and the server still doesn't shut down. This is typically because you have an `psql` session running in some other window (the PostgreSQL server won't shut down until all clients have disconnected from the server). The way to fix this is to find the `psql` session and end it. If you can find the window where it's running, simply use `\q` to quit from `psql`. If you can't find the window, or it's running from a different machine (e.g. you're in the lab and find that you left a `psql` running at home), then use `ps` to find the process id of the `psql` session and stop it using the Linux `kill` command.

Can't restart server?

Occasionally, you'll find that your PostgreSQL server was not shut down cleanly the last time you used it and you cannot re-start it next time you try to use it. We'll discuss how to solve that here ...

The typical symptoms of this problem are that you log in to `d` , set up your environment, try to start your PostgreSQL server and you get the message:

```
pg_ctl: another server may be running; trying to start server anyway
waiting for server to start.... stopped waiting
pg_ctl: could not start server
Examine the log output.
```

When you go and check the log file, you'll probably find, right at the end, something like:

```
$ tail -2 $PGDATA/log
FATAL:  lock file "postmaster.pid" already exists
HINT:  Is another postmaster (PID NNNN) running in data directory "/localstorage/$USER/pgsql"?
```

where *`NNNN`* is a number.

There are two possible causes for this: the server is already running, or the server did not terminate properly after the last time you used it. You can check whether the server is currently running by the command `psql -l` . If that gives you a list of your databases, then you simply forgot to shut the server down last time you used it and it's ready for you to use again. If `psql -l` tells you that there's no server running, then you'll need to do some cleaning up before you can restart the server ...

When the PostgreSQL server is run, it keeps a record of the Unix process that it's running as in a file called:

```
$PGDATA/postmaster.pid
```

Normally when your PostgreSQL server process terminates (e.g. via `p0`), this file will be removed. If your PostgreSQL server stops, and this file persists, then `p1` becomes confused and thinks that there is still a PostgreSQL server running even though there isn't.

The first step in cleaning up is to remove this file:

```
$ rm $PGDATA/postmaster.pid
```

You should also clean up the socket files used by the PostgreSQL server. You can do this via the command:

```
$ rm $PGDATA/.s*
```

Once you've cleaned all of this up, then the `p1` command ought to allow you to start your PostgreSQL server ok.

Happy PostgreSQL'ing!