

POSTGRESQL DATABASE HANDBOOK

Hot Recipes for the PostgreSQL DB



PostgreSQL

GABRIEL CANEPA

 **SYSTEM CODE GEEKS**
SYSADMINS RESOURCE CENTER

PostgreSQL Database Handbook

Contents

1	Introduction and installation	1
1.1	What's PostgreSQL?	1
1.2	Installing PostgreSQL	1
1.3	Populating the database with data	2
1.4	Configuring phppgadmin in Linux	3
1.5	PostgreSQL Windows client	6
2	Commands and datatypes	8
2.1	PostgreSQL commands	8
2.1.1	Getting help	8
2.1.2	Displaying databases and tables	9
2.2	PostgreSQL data types	10
2.3	Enumerated types	12
2.4	Summary	12
3	VACUUM Command Example	13
3.1	Updating and removing rows	13
3.2	Introducing VACUUM	15
3.3	Summary	17
4	PostgreSQL indexes example	18
4.1	Introducing indexes	18
4.2	Examples	18
4.3	Unique indexes	21
4.4	Multicolumn indexes	21
4.5	Summary	21
5	Database Creation and Data Population	22
5.1	Creating a new database	22
5.2	Populating the database	24
5.3	More queries	25
5.4	Summary	26

6	Common Table Expressions	27
6.1	Definition of Common Table Expressions (CTE)	27
6.2	Non-recursive Common Table Expressions	27
6.3	Summary	31
7	Hot-Standby Database Replication Tutorial	32
7.1	Step 0 - Change hostnames and IP addresses as needed	32
7.2	Step 1 - Configuring the master	33
7.3	Step 2 - Configuring the slave	33
7.4	Step 3 - Performing the replication	34
7.5	Step 4 - Testing the replication	34
7.6	Troubleshooting	35
8	Backup, Restore and Migration	36
8.1	Backup, restore, and migration strategies	36
8.2	Installing Barman	36
8.2.1	Step 1 - Create a dedicated PostgreSQL user in oldserver	36
8.2.2	Step 2 - Create the .pgpass file in newserver	37
8.2.3	Step 3 - Set up key-based authentication	38
8.2.4	Step 4 - Configure Barman	38
8.2.5	Step 5 - Configure PostgreSQL	39
8.2.6	Step 6 - Test the Barman configuration	39
8.2.7	Step 7 - Perform the backup	40
8.2.8	Step 8 - Restore the backup on newserver	41
8.3	Automating backups	43
9	Connect to PostgreSQL using PHP	44
9.1	Installing the software	44
9.2	Connecting to the database server	45
9.3	Writing the application	46
9.4	Creating a mobile-friendly web page	48
9.5	Summary	50

Copyright (c) Exelixis Media P.C., 2016

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

Preface

PostgreSQL, often simply Postgres, is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards-compliance. As a database server, its primary function is to store data securely, and to allow for retrieval at the request of other software applications. It can handle workloads ranging from small single-machine applications to large Internet-facing applications with many concurrent users.

PostgreSQL is developed by the PostgreSQL Global Development Group, a diverse group of many companies and individual contributors. It is free and open-source software, released under the terms of the PostgreSQL License, a permissive free-software license. (<https://en.wikipedia.org/wiki/PostgreSQL>)

In this ebook, we provide a compilation of PostgreSQL tutorials that will help you set up and run your own database management system. We cover a wide range of topics, from installation and configuration, to custom commands and datatypes. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

Gabriel Canepa is a Linux Foundation Certified System Administrator (LFCS-1500-0576-0100) and web developer from Villa Mercedes, San Luis, Argentina.

He works for a worldwide leading consumer product company and takes great pleasure in using FOSS tools to increase productivity in all areas of his daily work.

When he's not typing commands or writing code or articles, he enjoys telling bedtime stories with his wife to his two little daughters and playing with them, the great pleasure of his life.

Chapter 1

Introduction and installation

With all the Relational Database Management Systems (RDBMs) out there, it may be somewhat difficult to identify the best solution for your needs and to take an informed decision as to which one to choose. Thus, in this series we will provide an introduction to PostgreSQL and share some of the reasons why you may want to consider this solution when exploring the available technologies for a database implementation.

1.1 What's PostgreSQL?

PostgreSQL, also known by its alias Postgres, is a cross-platform object-relational database management system (ORDBMs for short). Its development started in the University of California at Berkeley in the mid '80s with a project they named simply **POSTGRES**, which did not feature SQL as query language at first. In the mid '90s, two students added SQL to the code inherited from the university, and PostgreSQL was born as an open-source project. Today, PostgreSQL has been long known (and has a strong reputation for) for being able to handle significant workloads with a large number of concurrent users. In addition, it provides bindings for many programming languages, making it an ideal solution for a client-server environment.

1.2 Installing PostgreSQL

In this article we will explain how to install a PostgreSQL server in Ubuntu Server 16.04 (IP address 192.168.0.54), how to load a sample database, and how to install a client application (which will serve as an administrative tool) for Linux and Windows.


Step 1 - Launch a terminal and install the server and the web-based administration tool:

```
sudo aptitude install postgresql phppgadmin
```

Step 2 - Verify that the database service is running and listening on port 5432:

```
systemctl is-active postgresql  
sudo netstat -npltu | grep postgres
```

The first command should indicate that unit postgresql is Active, and the second command should show that the service is listening on the right port, as shown in Fig. 1.1:



```
gacanepa@ubuntu:~$ systemctl is-active postgresql  
active  
gacanepa@ubuntu:~$ sudo netstat -npltu | grep postgres  
tcp        0      0 127.0.0.1:5432        0.0.0.0:*           LISTEN      6371/postgres  
tcp6       0      0 :::1:5432            :::*                LISTEN      6371/postgres  
gacanepa@ubuntu:~$
```

Figure 1.1: Verifying that PostgreSQL is running and listening on port 5432

Step 3 - Switch to the postgres Linux account and create a new role for queries:

The installation process created a new Linux account named postgres. By default, this is the only account with permissions to access the database prompt initially.

To switch to the postgres account, do

```
sudo -i -u postgres
```

And run the following command to create a new database role named gacanepa (enter the password twice when you're prompted to do so):

```
createuser gacanepa --no-createdb --no-superuser --no-createrole --pwprompt
```

Although the options in the above command are self-explanatory, let's just say that this particular role will not be allowed to create databases or roles, and will not have superuser privileges. Other options for the createuser command are available in its man page (which you can access from the Linux command prompt as `man createuser`).

Step 4 - Create a new database

While you're still logged on as postgres, create a database:

```
createdb World_db
```

1.3 Populating the database with data

Once we have created the database, it's time to populate it with actual data we can later query:

Step 5 - Download a sample database

The [wiki](#) links to several sample databases that we can download and use. For this example, we will download and install the world database, which contains countries, cities, and spoken languages, among other data.

```
wget https://pgfoundry.org/frs/download.php/527/world-1.0.tar.gz[https://pgfoundry.org/frs/ ↵  
download.php/527/world-1.0.tar.gz]  
tar xzf world-1.0.tar.gz
```

Step 6 - Restore the database dump:

The database dump file is located at dbsamples-01/world inside the current working directory, as shown in Fig. 1.2:

```
cd dbsamples-0.1/world  
psql World_db < world.sql
```

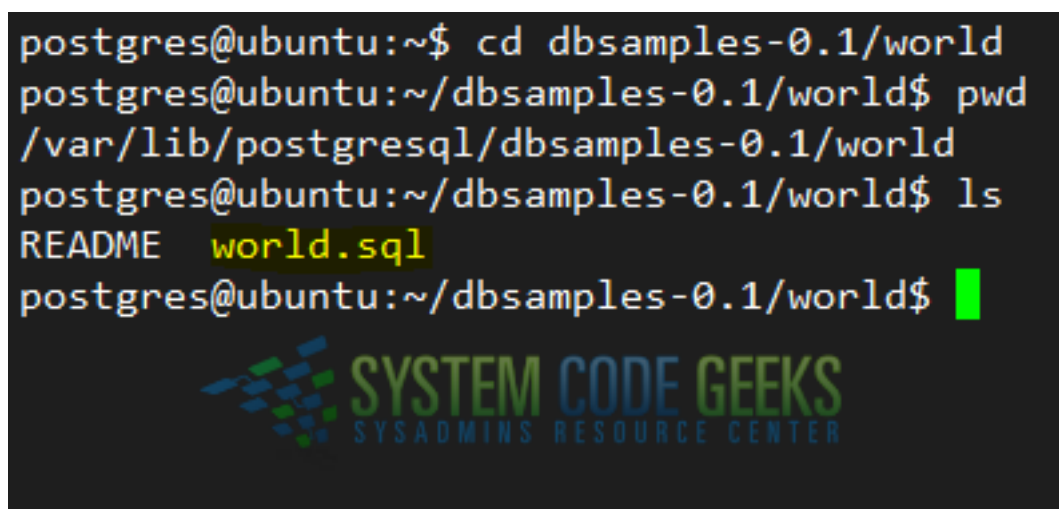
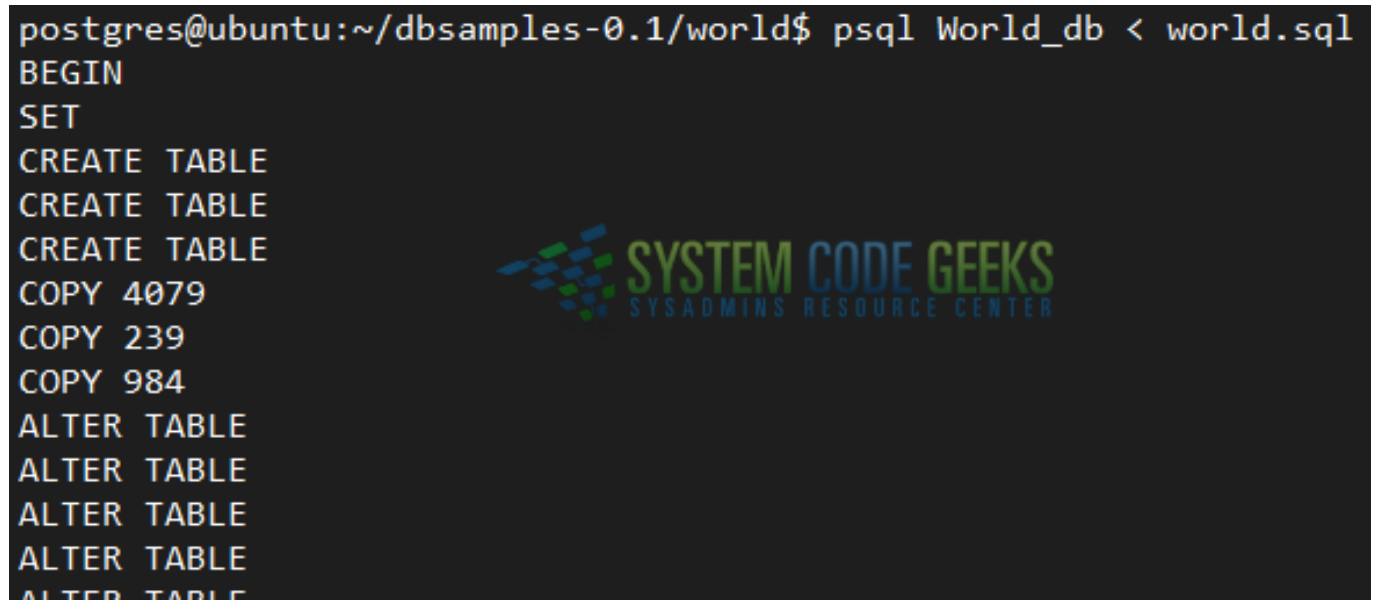


Figure 1.2: Locating the database dump file to restore

As the tables are created and populated with data, the output should be similar to Fig. 1.3:

A terminal window with a dark background. The prompt is 'postgres@ubuntu:~/dbsamples-0.1/world\$'. The command 'psql World_db < world.sql' has been entered. The output shows SQL commands being executed: 'BEGIN', 'SET', 'CREATE TABLE' (repeated three times), 'COPY 4079', 'COPY 239', 'COPY 984', and 'ALTER TABLE' (repeated four times). A watermark for 'SYSTEM CODE GEEKS' and 'SYSADMINS RESOURCE CENTER' is visible in the center of the terminal output.

```
postgres@ubuntu:~/dbsamples-0.1/world$ psql World_db < world.sql
BEGIN
SET
CREATE TABLE
CREATE TABLE
CREATE TABLE
COPY 4079
COPY 239
COPY 984
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
```

Figure 1.3: Restoring the database contents from the dump file

After completing the above 6 steps, we now have a fully-setup PostgreSQL database.

1.4 Configuring phppgadmin in Linux

In order to allow remote (LAN) access to the web-based administration tool, follow these steps:

Step 7 - Integrate phppgadmin with Apache

Open `/etc/apache2/conf-enabled/phppgadmin.conf`, and comment out the following line:

```
Require local
```

then add

```
Require all granted
```

just below (see Fig. 1.4 for details)

```
Alias /phppgadmin /usr/share/phppgadmin

<Directory /usr/share/phppgadmin>

<IfModule mod_dir.c>
DirectoryIndex index.php
</IfModule>
AllowOverride None

# Only allow connections from localhost:
#Require local ←
Require all granted
<IfModule mod_php.c>
php_flag magic_quotes_gpc Off
```

Figure 1.4: Configuring access permissions for phppgadmin

Step 8 - Grant SELECT permissions on World_db

Switch to the postgres Linux account and open the database prompt by typing

```
psql
```

Then connect to the World_db database:

```
\c World_db;
```

Finally, grant SELECT permissions to role gacanepa, and exit (q) the database prompt:

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO gacanepa;
\q
```

See Fig. 1.5 for details:

```
postgres=# \c World_db;
You are now connected to database "World_db" as user "postgres".
World_db=# GRANT SELECT ON ALL TABLES IN SCHEMA public to gacanepa;
GRANT
World_db=# \q
postgres@ubuntu:~$
```

Figure 1.5: Connecting to a database and granting SELECT permissions to a role

Step 9 - Restart Apache and PostgreSQL

We are almost there. Let's restart Apache and PostgreSQL:

```
systemctl restart {apache2,postgresql}
```

Step 10 - Login to phpPgadmin

Point your web browser to 192.168.0.54/phpPgadmin and click on PostgreSQL in the left hand section. Next, enter the role and password you created in Step 3 above, as shown in Fig. 1.6:



Figure 1.6: Logging on to phpPgadmin

Once there, click on the World_db database and then enter a SQL query of your choice (see Fig. 1.7):

```
SELECT A.name "City", A.district "District",  
B.name "Country", C.language "Language"  
FROM city A JOIN country B ON A.countrycode=B.code  
JOIN countrylanguage C ON A.countrycode=C.countrycode  
WHERE A.name='Rosario' AND C.official='TRUE';
```

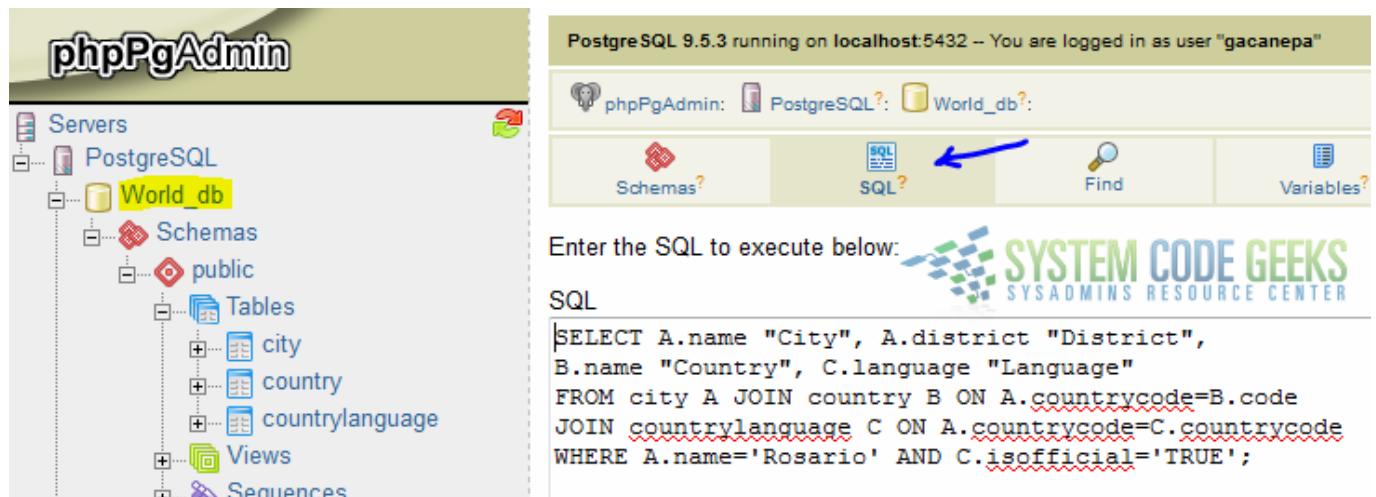


Figure 1.7: Our first query to the PostgreSQL database through phppgadmin

Click **Execute** at the bottom. The results should be as shown in Fig. 1.8:

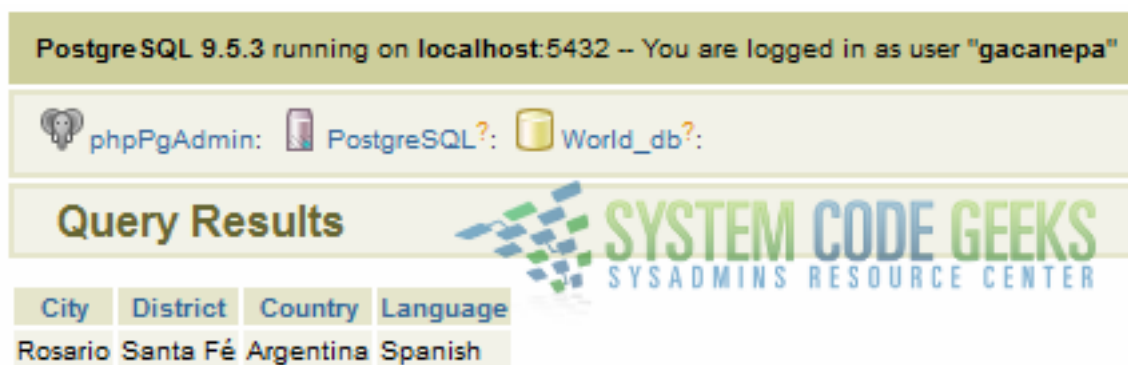


Figure 1.8: The results of our first query

1.5 PostgreSQL Windows client

If you are using Microsoft Windows, in addition to phppgadmin (which you can access through a web browser), you can also install a client application named pgAdmin in order to connect to the database server. You can download it from the pgAdmin PostgreSQL tools page at <https://www.pgadmin.org/download/windows.php>. The installation will only take a few clicks.

Although it is better known in Windows environments, pgAdmin is also available for Mac OS X as well.

When you're done with the installation, make sure the following lines are present in the configuration files. Otherwise, you will NOT be able to connect to the database server from a machine other than where you installed and running.

In `/etc/postgresql/9.5/main/postgresql.conf`:

```
listen_addresses = '*'
```

will ensure the database server is listening on all interfaces, and because of the following line in `/etc/postgresql/9.5/main/pg_hba.conf`:

```
host      all             all             192.168.0.0/24      md5
```

you can now connect to the database server from any machine in the 192.168.0.0/24 network.

Once you added the above lines, open pgAdmin from Start → All programs → pgAdmin III. Then click on File → Add server and fill the connection details (see Fig. 1.9). If you fill the password box as shown below, the credentials will be saved in plain text in your user profile. If you are using a shared computer that is probably not a good idea, so you may want to leave that field blank if that's the case:

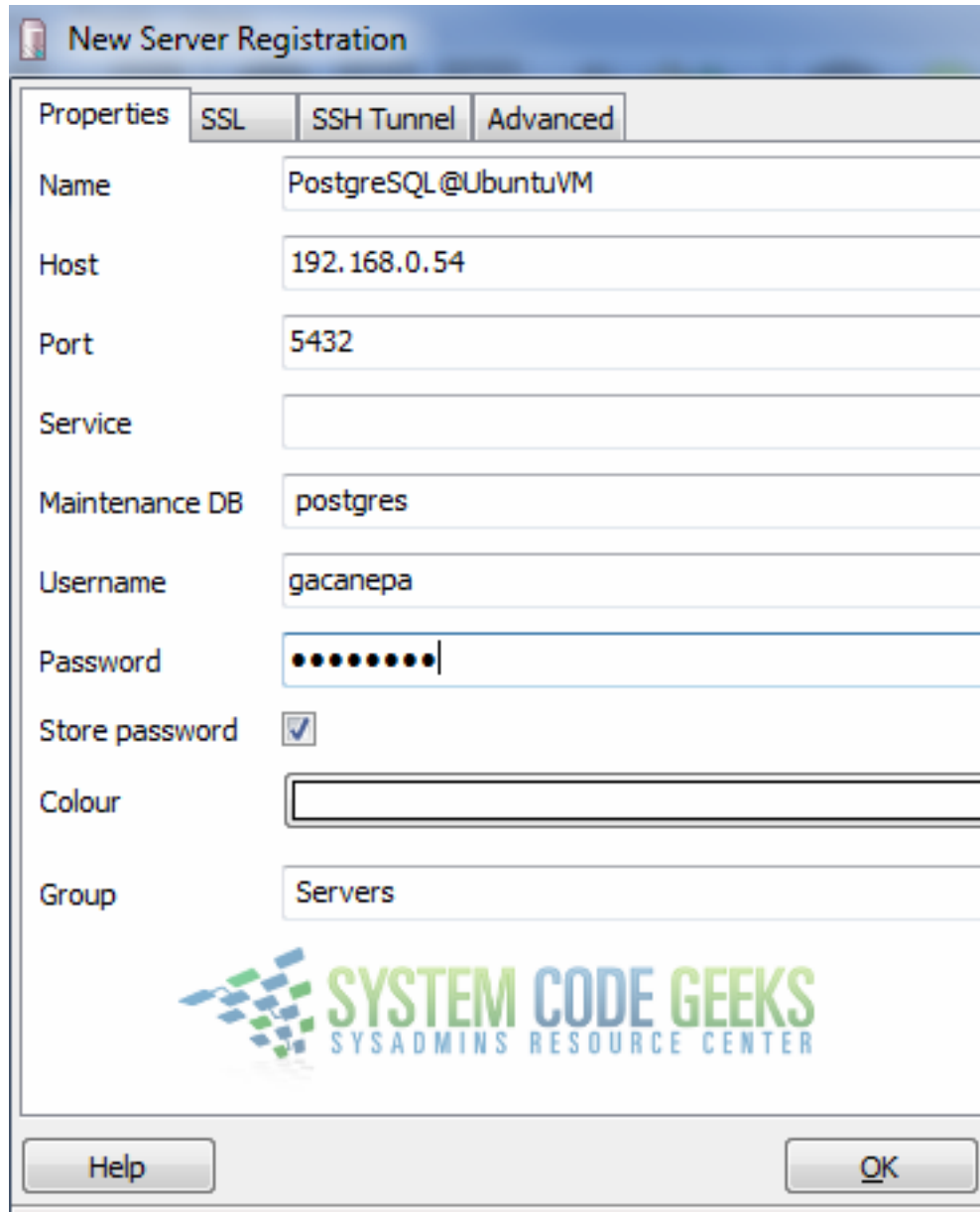


Figure 1.9: Configuring access to our database server through pgAdmin

Congratulations! You have successfully installed a PostgreSQL database server and are now able to access it both from a web interface and using a client application.

Chapter 2

Commands and datatypes

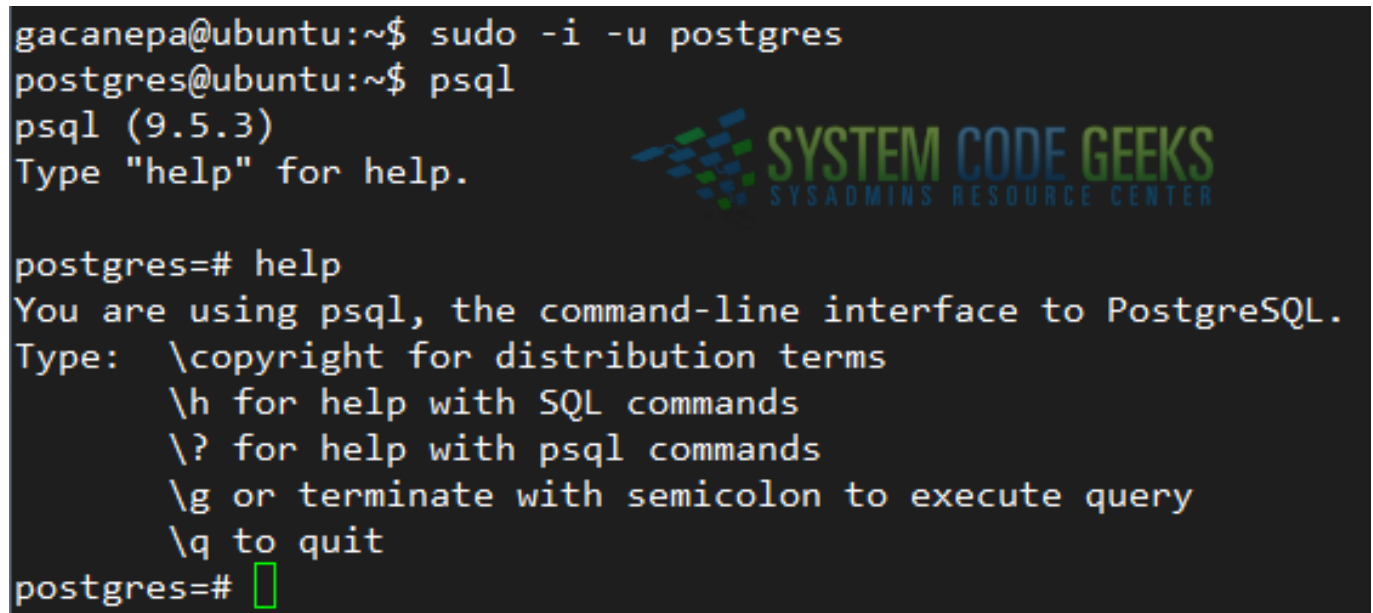
In our previous post ([PostgreSQL: Introduction and installation](#)) we explained how to install and use a desktop and a web-based client to query a sample database we created and populated. We also introduced two basic commands to connect to a database (`c` followed by the database name) and to quit (`q`) the PostgreSQL prompt.

2.1 PostgreSQL commands

In this tutorial we will introduce you to other useful PostgreSQL-specific (**psql** for short from now on) commands. To do so, let's open the psql prompt by switching to the postgres Linux account and typing `psql` in the command line.

2.1.1 Getting help

Once in the psql prompt, type `help` and press Enter. The output should be similar to Fig. 2.1:



```
gacanepa@ubuntu:~$ sudo -i -u postgres
postgres@ubuntu:~$ psql
psql (9.5.3)
Type "help" for help.

postgres=# help
You are using psql, the command-line interface to PostgreSQL.
Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit
postgres=#
```

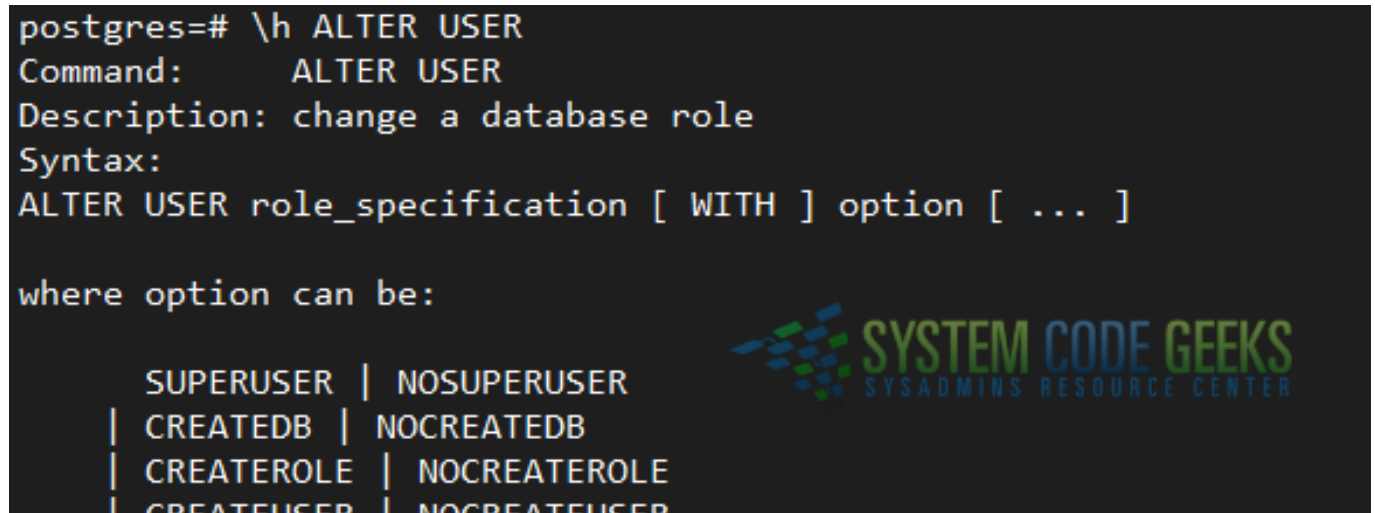
Figure 2.1: Accessing the built-in PostgreSQL help

The above figure shows the following tips - make sure you remember because they will come in handy more than once. If you need help with SQL commands, first off type `h` to view a list of the available options. Once you have identified the command

you need help with, use `q` to return to the `psql` prompt and then type `h` followed by the SQL command you have chosen. For example, let's say we chose `ALTER USER`. To see the help for that command, do

```
\h ALTER USER
```

as shown in Fig. 2.2:



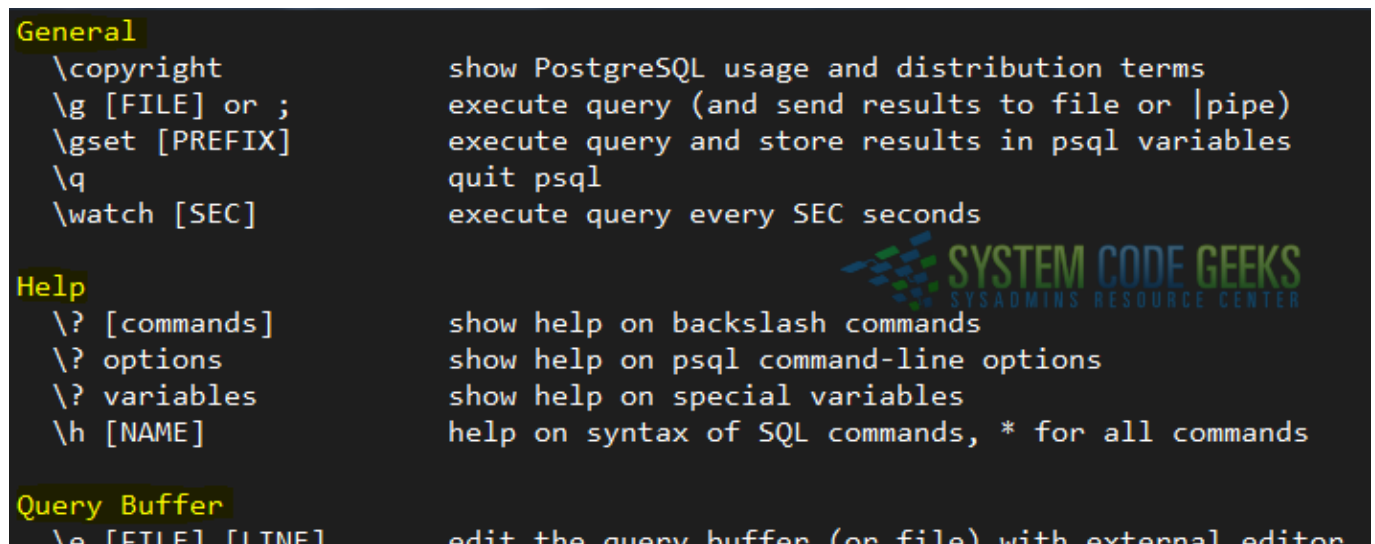
```
postgres=# \h ALTER USER
Command:      ALTER USER
Description:  change a database role
Syntax:
ALTER USER role_specification [ WITH ] option [ ... ]

where option can be:

          SUPERUSER | NOSUPERUSER
          | CREATEDB | NOCREATEDB
          | CREATEROLE | NOCREATEROLE
          | CREATEUSER | NOCREATEUSER
```

Figure 2.2: Getting help about SQL commands

On the other hand, if you get stuck with a database management task, do `?` and you will see the available `psql` commands grouped by categories, as seen in Fig. 2.3 (some of them are highlighted in yellow - the output is truncated for the sake of space):



```
General
\copyright      show PostgreSQL usage and distribution terms
\g [FILE] or ;  execute query (and send results to file or |pipe)
\gset [PREFIX]  execute query and store results in psql variables
\q             quit psql
\watch [SEC]    execute query every SEC seconds

Help
\? [commands]   show help on backslash commands
\? options      show help on psql command-line options
\? variables    show help on special variables
\h [NAME]       help on syntax of SQL commands, * for all commands

Query Buffer
\e [FILE] [LINE] edit the query buffer (or file) with external editor
```

Figure 2.3: Getting help with `psql` commands

As before, type `q` to exit the help and return to the `psql` prompt.

2.1.2 Displaying databases and tables

If you find yourself examining a database server you haven't previously worked with, or if you are not familiar with the structure of a given database, you may want to start off by listing the databases and their respective tables.

To list the databases, simply do `\l`

To view the tables in the **World_db** database (which is the one we imported in our previous tutorial), connect to it and type `\dt`

Keep in mind that you can switch from a given database to another (Alberdi in the following example) with `\c Alberdi`

The above commands are shown in Fig. 2.4 below:

```
postgres=# \l
```

Name	Owner	Encoding	Collate	Ctype	Access privileges
Alberdi	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
World_db	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres +
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	postgres=CTc/postgres +

```
(5 rows)
```

```
postgres=# \c World_db
You are now connected to database "World_db" as user "postgres".
World_db=# \dt
```

Schema	Name	Type	Owner
public	city	table	postgres
public	country	table	postgres
public	countrylanguage	table	postgres

```
(3 rows)
```

```
World_db=# \c Alberdi
You are now connected to database "Alberdi" as user "postgres".
Alberdi=#
```

Figure 2.4: Listing databases and tables

With the psql commands above we have learned to how to list databases and switch between one and another, how to list tables, and how to get help if we get stuck along the way.

2.2 PostgreSQL data types

As a preparation to creating our own databases and tables from scratch (which we will cover in an upcoming tutorial), we need to know how what are the allowed built-in, general-purpose data types for table fields. The [PostgreSQL 9.5 documentation](#) lists the following data types and more:

a) **Numeric types** (with corresponding storage sizes and ranges) are listed in Fig. 2.5:

Name	Storage Size	Range
smallint	2 bytes	-32768 to +32767
integer	4 bytes	-2147483648 to +2147483647
bigint	8 bytes	-9223372036854775808 to +9223372036854775807
decimal	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
smallserial	2 bytes	1 to 32767
serial	4 bytes	1 to 2147483647
bigserial	8 bytes	1 to 9223372036854775807




Figure 2.5: Numeric data types

You will often choose a numeric type for fields that will store amounts of items, grade results, etc.

b) **Character types** (see Fig. 2.6):

These types are used to store regular (English) text or character strings, typically resulting from user interaction.

Name	Description
varchar (n)	variable-length limited to n characters
char (n)	fixed-length (n), blank padded
text	variable unlimited length




Figure 2.6: Character data types

c) **Date/time types** (see Fig. 2.7):

These data types are used to indicate the date and or time when an event has been recorded in the database. If you require to store the time zone, there's a dedicated type for that as well.

Name	Storage Size	Description	Low Value	High Value
timestamp	8 bytes	both date and time (no time zone)	4713 BC	294276 AD
timestampz	8 bytes	both date and time, with time zone	4713 BC	294276 AD
date	4 bytes	date (no time of day)	4713 BC	5874897 AD
time	8 bytes	time of day (no date)	00:00:00	24:00:00
timetz	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459
interval	16 bytes	time interval	-178000000 years	178000000 years



Figure 2.7: Date time data types

d) **Binary type** (see Fig. 2.8):

You will often use this type to indicate true/false, active/inactive, and enabled/disabled statuses.

Name	Storage Size	Description
boolean	1 byte	state of true or false



Figure 2.8: Binary data type

Knowing the allowed ranges for each data type is essential to choosing the right type for fields. It is also critical as far as disk usage is concerned, as a 2-byte integer (**smallint**) will occupy less space than a 4-byte one (**integer**). As a rule of thumb, only use a "larger" data type if and only if a smaller type is not likely to scale well with the expected use and growth of the database in terms of record numbers.

Also, the length of character fields must be taken into account while planning -for example- a web application that will gather data through forms or other types of input. While in certain cases users should not be allowed to enter text of indefinite length, you should plan ahead so that they can still enter all that is necessary. Although form validation and sanitization are out of the scope of this tutorial, you definitely will want to make sure that your application does not present security holes and is not abuse-prone.

The use of data types, among other things, contributes to data consistency in a table by ensuring that a given field will only accept the type of data it is configured to store.

2.3 Enumerated types

Besides the general purpose data types, PostgreSQL allows us to create our own data types in the form of static, ordered set of values (for example, the months of the year, or the days of the week), similarly to the enum type supported in several programming languages. We will see the benefit of enumerated types when we create our first database and start inserting data into it.

2.4 Summary

Now that you have learned how to use basic psql commands and have reviewed the most used data types, we are better prepared to dive deeper into PostgreSQL database administration. Stay tuned for the next tutorial!

Chapter 3

VACUUM Command Example

In the previous tutorials of this series (“[PostgreSQL: Introduction and installation](#)” and “[PostgreSQL commands and datatypes](#)”) we explained how to load a sample database (World_db) into our PostgreSQL server and how to get help with both SQL and psql commands.

In real-world scenarios, you will need to perform **CRUD** (Create, Read, Update, and Delete) operations on database tables all the time. In this post we will learn how to do U (updates) and D (deletes), and show how to clean up the database by removing the left overs resulting from these operations. As we will see in a moment, PostgreSQL provides an effective garbage collector for this.

Without going into the nitty-gritty of what happens under the hood, we can mention briefly that the previous versions of updated records or deleted table entries are not actually removed from the database. Think about the need to rollback a given transaction and this will make sense. They are just “not visible” anymore by regular means, and they keep contributing to the amount of used hard disk space until a clean-up is performed using the **VACUUM** psql command. Let’s take a look at it in greater detail later.

3.1 Updating and removing rows

Using the World_db database, let’s update by 7% the population of all cities in the city table. Before we do that, let’s take a look at the impact this operation would have on the current data by using a basic SELECT statement.

Before a mass update or removal, using SELECT to print the records that will be impacted by that operation is a wise thing to do. Among other things, this can help you prevent undesired results (and the associated later regret), especially if you forget to add a WHERE clause to the operation.

We will print the city name, its current population, and the population after our proposed update. To round the population increase to the nearest integer, we will use the **ROUND** function as shown in Fig. 3.1:

Name	Current population	New population
Aachen	243825	260893
Aalborg	161161	172442
Aba	298900	319823
Abadan	206073	220498
Abaetetuba	111258	119046
Abakan	169200	181044
Abbotsford	105403	112781
Abeokuta	427400	457318
Aberdeen	213070	227985
Abha	112300	120161
Abidjan	2500000	2675000
Abiko	126670	135537
Abilene	115830	124045

Figure 3.1: Displaying the results of a preliminar SQL query before updating

```
SELECT name AS "Name", population AS "Current population", ROUND(population * 1.07) AS "↔  
New population" FROM city ORDER BY name;
```

With the AS keyword you can create an alias for the associated field so that the results of the query will use it as header. As you can see in Fig. 3.1, we renamed name and population to **Name** and **Current population**, respectively. In addition, we named the results of the mathematical operation as **New population**.

Now let's do the actual update. In this case we will not use a WHERE clause as we actually want to update all cities. This will result in the population update of all 4079 cities currently present in the city table, as we can see in Fig. 3.2:

```
World_db=# UPDATE city SET population = ROUND(population * 1.07);  
UPDATE 4079  
World_db=# SELECT COUNT(*) FROM city;  
count  
-----  
4079  
(1 row)  
  
World_db=#
```

Figure 3.2: Updating the city table

```
UPDATE city SET population = ROUND(population * 1.07);
```

Now let's delete all Australian cities where the Id is greater than 135 (this will exclude Canberra, the capital, which is referenced in the country table). As before, use a SELECT first to examine the records that will be deleted:

```
SELECT name FROM city WHERE countrycode='AUS';
```

If you're OK with it, then proceed with the DELETE operation:

```
DELETE FROM city WHERE Id BETWEEN 136 AND 143;
```

Refer to Fig. 3.3 for details:

```
World_db=# SELECT id, name FROM city WHERE countrycode='AUS';
 id |      name
-----+-----
 132 | Brisbane
 130 | Sydney
 131 | Melbourne
 133 | Perth
 134 | Adelaide
 135 | Canberra
 136 | Gold Coast
 137 | Newcastle
 138 | Central Coast
 139 | Wollongong
 140 | Hobart
 141 | Geelong
 142 | Townsville
 143 | Cairns
(14 rows)

World_db=# DELETE FROM city WHERE Id BETWEEN 136 AND 143;
DELETE 8
World_db=#
```

Figure 3.3: Selecting records before deleting them

On tables that are heavily updated or where removals are performed constantly, this will translate into a lot of wasted disk space. Keep in mind that when you perform an update on a table or remove a record, the original is kept in the database.

3.2 Introducing VACUUM

To formally introduce VACUUM, let's use what we learned in [PostgreSQL commands and data types](#) to display the help about this command (see Fig. 3.4):

```
World_db=# \h VACUUM
Command:      VACUUM
Description:  garbage-collect and optionally analyze a database
Syntax:
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ] [ table_name ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table_name ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table_name ] (column_name
World_db=#
```




Figure 3.4: Displaying help about VACUUM

```
\h VACUUM;
```

All of the below commands can be applied to the entire database (no arguments) or a single table (name the table at the end of the command).

To collect the garbage present in the database, just do

```
VACUUM;
```

However, that will not free up the space used by the old records back to the operating system - it will only clean up the old records and then make the space available to be reused by the same table. On the other hand,

```
VACUUM FULL;
```

will ensure that whatever space is freed up will be returned to the operating system.

Additionally,

```
VACUUM FULL VERBOSE;
```

will also display messages about what's going on.

That said, let's perform a full, verbose vacuum on the city table (refer to Fig. 3.5 for details):

```
World_db=# VACUUM FULL VERBOSE city;
INFO:  vacuuming "public.city"
INFO:  "city": found 8 removable, 4071 nonremovable row versions in 62 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU 0.01s/0.00u sec elapsed 0.03 sec.
VACUUM
World_db=#
```




Figure 3.5: Performing a FULL, VERBOSE VACUUM

```
VACUUM FULL VERBOSE city;
```

As you can see above, VACUUM located and removed the space left behind by the deletion of the 8 records from the city table earlier. On large scale updates and removals, this will translate into considerable space disk savings.

As good as the VACUUM command is, having to run it manually could become a tedious task. Thus, by default, there's an AUTOVACUUM daemon that is enabled and does the job for you automatically while the database server is running. You can

find more details about its operations in the AUTOVACUUM PARAMETERS section of the main configuration file `/etc/postgresql/9.5/main/postgresql.conf`.

You can verify that the AUTOVACUUM process is running with:

```
ps aux | grep autovacuum | grep -v grep
```

3.3 Summary

Freeing up space in tables that are constantly updated or where records are often deleted not only will help you save space, but also improve the performance of queries performed on the table. Following the instructions shared in this article you will be contributing to the health of your database and saving valuable storage space.

Chapter 4

PostgreSQL indexes example

In [our previous article](#) we discussed how to free up disk space by vacuuming tables with frequent updates and deletes. Under the hood, this procedure also helps to improve the performance of other CRUD operations performed on those tables. In this tutorial we will explain how to optimize SELECT queries with WHERE clauses using indexes in PostgreSQL tables.

4.1 Introducing indexes

The best way to introduce the concept and the use of indexes in a database is using a book analogy. If you buy a new book for a college class, you will most likely start by looking at the index at the end of the book for a particular topic. There is no doubt that this would be a much faster way to find the information that you need than thumbing through the book from the beginning.

Likewise, in the context of databases, an index is an actual structure that references the information found in a given table.

Particularly in PostgreSQL, an index consists of a copy of the indexed data along with the corresponding reference to its location. Thus, insert and update queries are expected to become slower on columns with indexes. That said, the first rule of thumb is: “Avoid at the extent possible creating indexes on columns with frequent bulk inserts or updates. Use indexes on columns that are mostly read-only or where the volume of insert / update operations is low.” Additionally, indexes can also improve the performance of update operations that use WHERE clauses.

4.2 Examples

Let’s return to the book analogy for a moment and use the World_db database to illustrate the need for indexes. Let’s modify a little the query that we used as an introductory example in [the first article of this series](#):

```
SELECT A.Id, A.name "City", A.district "District", B.name "Country", C.language "Language", ←  
       CASE WHEN C.official='TRUE' THEN 'Yes' WHEN C.official='FALSE' THEN 'No' END " ←  
Official language?" FROM city A JOIN country B ON A.countrycode=B.code JOIN ←  
countrylanguage C ON A.countrycode=C.countrycode WHERE A.Id=72;
```

The above query will return all records where the Id column in the city table is 72. Since we are performing a JOIN operation with other tables it is to be expected that we will get more than one result. In this case, we got 3 different records based on the different languages associated with this city, as you can see in Fig. 4.1:

```
World_db=# SELECT A.Id, A.name "City", A.district "District", B.name "Country", C.language
C.isofficial='FALSE' THEN 'No' END "Official language?" FROM city A JOIN country B ON A.c
countrycode WHERE A.Id=72;
```

id	City	District	Country	Language	Official language?
72	Rosario	Santa Fé	Argentina	Indian Languages	No
72	Rosario	Santa Fé	Argentina	Italian	No
72	Rosario	Santa Fé	Argentina	Spanish	Yes

(3 rows)

Figure 4.1: Our initial query

If SELECT operations like the above query are performed frequently searching by city.Id, it makes sense to create an index on that column in order to improve the overall performance. Before we do that, Let's do an EXPLAIN ANALYZE on this query by prepending this operation to the query itself. This will perform the query and indicate the execution time (see details highlighted in yellow in Fig. 4.2):

```
World_db=# EXPLAIN ANALYZE SELECT A.Id, A.name "City", A.district "District", B.name "Country", C.language
C.isofficial='FALSE' THEN 'No' END "Official language?" FROM city A JOIN country B ON A.countrycode=C.countrycode
WHERE A.Id=72;
```

QUERY PLAN

```
Nested Loop (cost=0.70..17.07 rows=7 width=43) (actual time=0.085..0.130 rows=3)
  Join Filter: (a.countrycode = c.countrycode)
    -> Nested Loop (cost=0.43..16.47 rows=1 width=42) (actual time=0.036..0.049 rows=1)
      -> Index Scan using city_pkey on city a (cost=0.28..8.30 rows=1 width=2)
        Index Cond: (id = 72)
      -> Index Scan using country_pkey on country b (cost=0.14..8.16 rows=1 width=1)
        Index Cond: (code = a.countrycode)
    -> Index Scan using countrylanguage_pkey on countrylanguage c (cost=0.28..0.51 rows=2 width=1)
      Index Cond: (countrycode = b.code)
Planning time: 0.796 ms
Execution time: 0.191 ms
```

Figure 4.2: Running EXPLAIN ANALYZE against the SQL query

As you can see, EXPLAIN ANALYZE says 3 rows were returned and gives us information about each step of our query. The execution time was 0.191 ms.

Let's now create the index on the city.Id column as follows. Please note that your indexes must ideally have a descriptive name (cityId_idx in this case, which fairly indicates that it is associated with the city.Id column):

```
CREATE INDEX cityId_idx ON city(Id);
```

Then repeat the EXPLAIN ANALYZE plus the query. Results are shown in Fig. 4.3:

```

World_db=# EXPLAIN ANALYZE SELECT A.Id, A.name "City", A.district "District", B.name "Official language?" FROM city A
WHERE A.countrycode=C.countrycode WHERE A.Id=72;

```

QUERY PLAN

```

Nested Loop  (cost=0.70..17.07 rows=7 width=43) (actual time=0.053..0.098 rows=3)
  Join Filter: (a.countrycode = c.countrycode)
    -> Nested Loop  (cost=0.43..16.47 rows=1 width=42) (actual time=0.032..0.045 rows=1)
      -> Index Scan using cityid_idx on city a  (cost=0.28..8.30 rows=1 width=42)
          Index Cond: (id = 72)
      -> Index Scan using country_pkey on country b  (cost=0.14..8.16 rows=1 width=4)
          Index Cond: (code = a.countrycode)
    -> Index Scan using countrylanguage_pkey on countrylanguage c  (cost=0.28..0.50 rows=1 width=4)
          Index Cond: (countrycode = b.code)
Planning time: 0.828 ms
Execution time: 0.159 ms

```




Figure 4.3: Running EXPLAIN ANALYZE against the SQL query AFTER creating an index

We can see that the use of the newly-created index was able to reduce the execution time by ~17% (0.159 ms compared to 0.191 ms).

On top of that, please refer to the figures in Fig. 4.4 that correspond to each query:

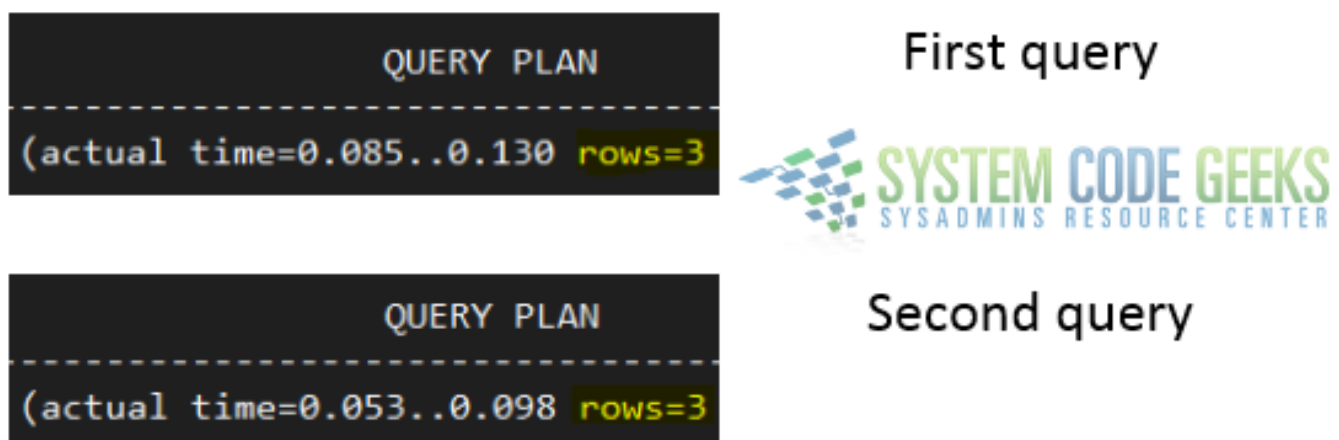


Figure 4.4: Estimated and actual startup and completion times before and after using an index

While the number of rows returned by each query was the same, the numbers inside parentheses show a performance increase. The first number (0.085 in the first case and 0.053 in the second) represents the estimated start-up time of the associated query step whereas the second number (0.130 and 0.098) indicates the actual execution time of such step.

The PostgreSQL documentation specifically states that learning how to use and interpret the **EXPLAIN** command is an art, and as such, it takes time to understand and master. We used it here to analyze our query and demonstrate the bounties of having an index in a table, but there is much more to EXPLAIN than that.

4.3 Unique indexes

There is a special type of index called unique. When it is used, it guarantees that the associated table will not have more than one row with the same value and thus will help us maintain data integrity and improve performance. Instead of a regular index, we could have created a unique index in the city.Id column above as follows:

```
CREATE UNIQUE INDEX cityId_idx ON city(Id);
```

You can also delete existing indexes in PostgreSQL as follows:

```
DROP INDEX cityId_idx;
```

Fairly easy, isn't it?

In this sense, a unique index will prevent a record with the same Id to be inserted into the table if no previous constraint (such as a primary key) exists on that column.

4.4 Multicolumn indexes

If you are likely to use more than one column in a SELECT query with a WHERE clause frequently, you may consider using a multicolumn index on them. The syntax is similar to the case of a single index:

```
CREATE INDEX index ON table (column1, column2);
```

where column1 and column2 are the columns where the index will be created. Feel free to add more columns if needed.

4.5 Summary

In this article we have discussed the need for indexes to improve performance on SELECT queries that use WHERE clauses. If you keep in mind the book analogy presented at the beginning, you will remember the fundamental concept behind using indexes.

Chapter 5

Database Creation and Data Population

In the first tutorial of this series ([PostgreSQL: Introduction and installation tutorial](#)) we explained how to download a sample database and import it into our PostgreSQL server. This approach, although appropriate at the time, did not show how to create a database of our own from scratch - which is an essential skill that every database administrator (DBA) or developer who intends to use PostgreSQL must have. In addition, under regular circumstances there are multiple database user accounts with different access permissions based on their respective assigned tasks. For that reason, we will also cover the topic of user creation here.

One important principle in database administration consists in creating as many users as needed depending on the required access privileges, but restricting such at a minimum. In other words, it is not wise to use the same database account for all applications since as that represents a serious security issue: if different accounts with different access permissions are used for separate applications, a compromised account will not necessarily have a negative impact on the others.

5.1 Creating a new database

To begin, let's switch to the postgresql Linux account and enter the psql prompt.

```
sudo -i -u postgres
psql
```

As we have explained previously, at this point we are not connected to any database.

Inside the database we were about to create, we will add two tables where we will store the actual information in an organized manner. Our database will be called **BookstoreDB** and the two tables will be **AuthorsTBL** and **BooksTBL** with the following fields in them (if you feel you need to brush up your memory about data types, feel free to refer to [PostgreSQL commands and datatypes](#)):

AuthorsTBL	
Field name	Data type
AuthorID	serial, primary key
AuthorName	varchar(100)
LastPublishedDate	date

Figure 5.1: Table AuthorsTBL

BooksTBL	
Field name	Data type
BookID	serial, primary key
BookName	varchar(100)
AuthorID	int, foreign key

Figure 5.2: Table BooksTBL

A primary key is a field in a table that is unique for each record, whereas a foreign key is a field that points to a primary key in another table. When performing operations on the table where the foreign key exists, it is required that the value used for such field exists as the primary key in the other table.

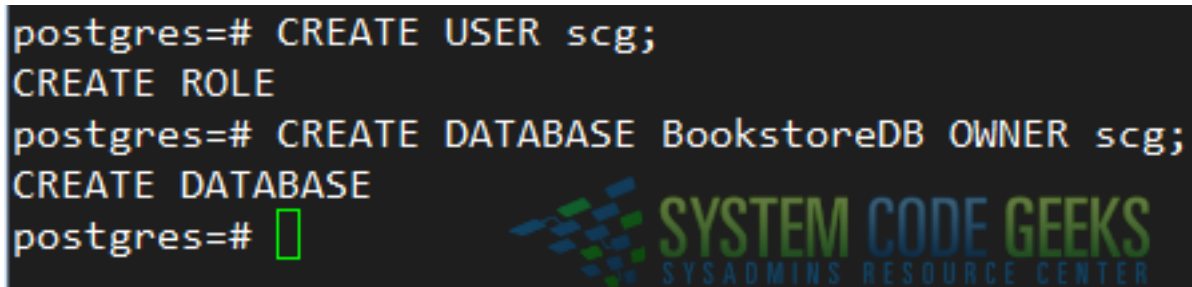
However, we will begin by creating a database role (also known as username) so that in the next step we will use it as owner of the database:

```
CREATE ROLE scg;
```

That said, let's create the new database as follows:

```
CREATE DATABASE BookstoreDB OWNER scg;
```

The expected output is shown in Fig. 5.3:



```
postgres=# CREATE USER scg;
CREATE ROLE
postgres=# CREATE DATABASE BookstoreDB OWNER scg;
CREATE DATABASE
postgres=#
```

The screenshot shows a PostgreSQL terminal session. The user 'postgres' enters the command 'CREATE USER scg;', which results in 'CREATE ROLE'. Then, the user enters 'CREATE DATABASE BookstoreDB OWNER scg;', which results in 'CREATE DATABASE'. The prompt 'postgres=#' is shown at the end of the session. A logo for 'SYSTEM CODE GEEKS SYSADMINS RESOURCE CENTER' is visible in the background of the terminal window.

Figure 5.3: Creating a role and use it as owner of a new database

Next, we will connect to our newly-created database and create the tables:

```
\c bookstoredb;

CREATE TABLE AuthorsTBL (
  AuthorID      SERIAL PRIMARY KEY,
  AuthorName     VARCHAR(100),
  LastPublishedDate  DATE
);

CREATE TABLE BooksTBL (
  BookID        SERIAL PRIMARY KEY,
  BookName       VARCHAR(100),
  AuthorID       SERIAL,
  FOREIGN KEY (AuthorID) REFERENCES AuthorsTBL (AuthorID)
);
```

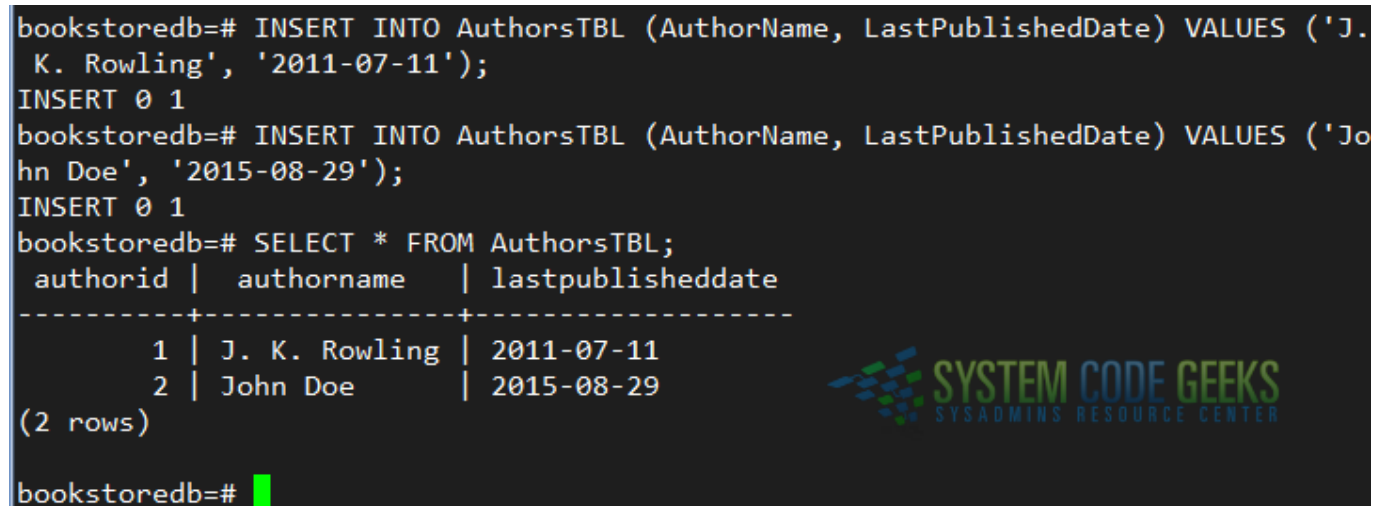
Now the next step consists of populating the database with actual data.

5.2 Populating the database

Since BooksTBL contains a foreign key that points to AuthorsID in AuthorsTBL, we will need to create a few records in that table first using the INSERT statement as follows. Note that each value must match the right field and data type:

```
INSERT INTO AuthorsTBL (AuthorName, LastPublishedDate) VALUES ('J. K. Rowling', '2011-07-11' ←
');
INSERT INTO AuthorsTBL (AuthorName, LastPublishedDate) VALUES ('John Doe', '2015-08-29');
```

Afterwards, we can use the SELECT statement to query the AuthorsTBL. Note how the AuthorID field was populated automatically since its data type was set to serial and primary key (see Fig. 5.4):



```
bookstoredb=# INSERT INTO AuthorsTBL (AuthorName, LastPublishedDate) VALUES ('J.
K. Rowling', '2011-07-11');
INSERT 0 1
bookstoredb=# INSERT INTO AuthorsTBL (AuthorName, LastPublishedDate) VALUES ('Jo
hn Doe', '2015-08-29');
INSERT 0 1
bookstoredb=# SELECT * FROM AuthorsTBL;
 authorid |  authorname  | lastpublisheddate
-----+-----+-----
       1 | J. K. Rowling | 2011-07-11
       2 | John Doe     | 2015-08-29
(2 rows)
```

Figure 5.4: Inserting records and querying the database

Next, let's insert some records into the BooksTBL table. If we try to insert a record with an AuthorID that does not exist in AuthorsTBL we will get an error as you can see in Fig. 5.5:

```
INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Harry Potter', 3);
INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Harry Potter and the philosophers stone' ←
, 1);
INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Harry Potter and the half-blood prince', ←
1);
INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Whatever', 2);
INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Whatever returns', 2);
```

```
bookstoredb=# INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Harry Potter', 3);
ERROR: insert or update on table "bookstbl" violates foreign key constraint "bookstbl_authorid_fkey"
DETAIL: Key (authorid)=(3) is not present in table "authorstbl".
bookstoredb=# INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Harry Potter and the philosophers st
one', 1);
INSERT 0 1
bookstoredb=# INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Harry Potter and the half-blood prin
ce', 1);
INSERT 0 1
bookstoredb=# INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Whatever', 2);
INSERT 0 1
bookstoredb=# INSERT INTO BooksTBL (BookName, AuthorID) VALUES ('Whatever returns', 2);
INSERT 0 1
bookstoredb=#
```

Figure 5.5: Inserting data with a non-existent foreign key causes an error

As you can see, an insert with a foreign key referencing a non-existent primary key in AuthorsTBL fails.

5.3 More queries

The classic `SELECT` statement as used earlier will return all the fields in a given table (that is what the star sign `*` stands for). We can also restrict the number of fields by listing them after the `SELECT`. For example, we can do

```
SELECT AuthorName FROM AuthorsTBL;
```

to retrieve only the `AuthorName`. Of course, that's going to be of little use, but it is worth mentioning.

We can also choose to combine records from both tables using a **JOIN**. This operation allow us to return a set of records from two or more tables as if they were stored in a single one. To illustrate, we will list all book titles along with the author name and perform the JOIN on the field that both tables have in common (`AuthorID`):


```
SELECT BooksTBL.BookName, AuthorsTBL.AuthorName FROM BooksTBL JOIN AuthorsTBL ON BooksTBL. ←
    AuthorID=AuthorsTBL.AuthorID;
```

If we only want to return those books where J. K. Rowling is the author, we can add a `WHERE` clause and either use `AuthorID=1` or `AuthorName=J. K. Rowling` in the filter. Usually, integers are preferred in `WHERE` clauses, so we will go with

```
SELECT BooksTBL.BookName, AuthorsTBL.AuthorName FROM BooksTBL JOIN AuthorsTBL ON BooksTBL. ←
    AuthorID=AuthorsTBL.AuthorID WHERE AuthorsTBL.AuthorID=1;
```

You can view the result of the above queries in Fig. 5.6:


```
bookstoredb=# SELECT BooksTBL.BookName, AuthorsTBL.AuthorName FROM BooksTBL JOIN AuthorsTBL ON BooksTBL.AuthorID=AuthorsTBL.AuthorID;
 bookname | authorname
-----+-----
 Harry Potter and the philosophers stone | J. K. Rowling
 Harry Potter and the half-blood prince | J. K. Rowling
 Whatever | John Doe
 Whatever returns | John Doe
(4 rows)
```



```
bookstoredb=# SELECT BooksTBL.BookName, AuthorsTBL.AuthorName FROM BooksTBL JOIN AuthorsTBL ON BooksTBL.AuthorID=AuthorsTBL.AuthorID WHERE AuthorsTBL.AuthorID=1;
 bookname | authorname
-----+-----
 Harry Potter and the philosophers stone | J. K. Rowling
 Harry Potter and the half-blood prince | J. K. Rowling
(2 rows)
```

```
bookstoredb=#
```

Figure 5.6: Using JOINS and WHERE clauses to refine searches

You can view other examples of querying in each of the previous articles in this series.

5.4 Summary

In this article we have explained how to create a database role and make it the owner of a database during creation. In addition, we showed how to create tables -taking into consideration the available datatypes- and how to populate and query them. By using JOINS and WHERE clauses you will be able to retrieve the necessary information as if it was all in the same table.

Chapter 6

Common Table Expressions

In our [previous article](#) we explained how to use **JOINS** to create more advanced **SELECT** queries. However, there are instances when using this technique to retrieve data from two or more tables does not satisfy our requirements or makes the query difficult to read - for example, if we need several **JOINS** or a subquery to return the desired information.

To solve this, standard SQL (note that this is not something exclusive to PostgreSQL), introduced the concept of Common Table Expressions (best known as CTE for short) in order to simplify this type of queries. In this article we will explain what CTEs are and how to use them.

6.1 Definition of Common Table Expressions (CTE)

Formally speaking, a CTE is a temporary result set that is created through the use of a **WITH** clause and is valid only during the execution of a given query. Another distinguishing feature of a CTE is that it can either reference itself (recursive CTE) or not (non-recursive CTE), providing the flexibility that common queries do not provide. A recursive CTE is often used when a calculation needs to be reported as part of the final result set, whereas a non-recursive one is usually utilized for a regular query. Additionally, its definition -meaning the fields it returns- is not stored as a separate database object.

Although Common Table Expressions can be used in **SELECT**, **INSERT**, **UPDATE**, or **DELETE** operations, we will only use the first type as it is the easiest to understand. Once you feel comfortable with using CTEs that involve **SELECT**s only, refer to [the official PostgreSQL 9.5 documentation](#) to learn how to use them with the other operation types.

All of these new concepts will better sink in as we illustrate them through examples, so let's begin.

6.2 Non-recursive Common Table Expressions

As usual, we will use the **World_db** database we installed in [the first article of this series](#). To begin, let's consider the following query:

```
SELECT A.name "City", A.district "District",  
       B.name "Country", C.language "Language"  
FROM city A JOIN country B ON A.countrycode=B.code  
JOIN countrylanguage C ON A.countrycode=C.countrycode  
WHERE A.name='Rosario' AND C.official='TRUE';
```

As you can probably guess by now, it will return the city name, the district, the country, and the official language where the city name is Rosario. If you look carefully, this query uses 2 **JOINS** - not a bad thing in itself, but the readability certainly could use some improvements.

Our first example of a Common Table Expression will be rather basic but does the job of introducing the concept:

```
WITH t AS (  
  SELECT A.name City, A.district District,  
         A.countrycode CountryCode, B.name Country  
  FROM city A JOIN country B ON A.countrycode=B.code)  
SELECT t.City, t.District, t.Country, C.language  
FROM t JOIN countrylanguage C on t.CountryCode = C.countrycode  
WHERE t.City='Rosario' AND C.official='TRUE';
```

Before we go into PostgreSQL and run the above query, let's split it into two parts to explain what is happening.

Step 1 - Define the CTE using the WITH clause. For simplicity, we will name the CTE as t, but you can use other name if you want.

```
WITH t AS (  
  SELECT A.name City, A.district District,  
         A.countrycode CountryCode, B.name Country  
  FROM city A JOIN country B ON A.countrycode=B.code)
```

If we were to do a `SELECT * FROM t;` at this point, we would get all the cities with their corresponding district and country. You may well be saying to yourself, "Then I don't see what's the point in using CTEs" - but wait, Step 2 will shed some light on the why.

Step 2 - Select the fields from the CTE and perform a JOIN with another table. As the CTE can be considered a temporary result set, we can perform JOINS on other tables. However, in this case we can use the more descriptive names given by the CTE instead of the original table names (are you seeing the readability improvements already?). Since both the city and country tables contain a field called *name*, the CTE allows us to refer to the city and country names as City and Country instead.


```
SELECT t.City, t.District, t.Country, C.language  
FROM t JOIN countrylanguage C on t.CountryCode = C.countrycode  
WHERE t.City='Rosario' AND C.official='TRUE';
```

As you can see in Fig. 6.1, the result is identical to the original query:

```

postgres=# \c World_db;
You are now connected to database "World_db" as user "postgres".
World_db=# SELECT A.name "City", A.district "District",
World_db=# B.name "Country", C.language "Language"
World_db=# FROM city A JOIN country B ON A.countrycode=B.code
World_db=# JOIN countrylanguage C ON A.countrycode=C.countrycode
World_db=# WHERE A.name='Rosario' AND C.official='TRUE';
  City   | District | Country | Language
-----+-----+-----+-----
Rosario | Santa Fé | Argentina | Spanish
(1 row)

```



```

World_db=# WITH t AS (
World_db=# SELECT A.name City, A.district District,
World_db=# A.countrycode CountryCode, B.name Country
World_db=# FROM city A JOIN country B ON A.countrycode=B.code)
World_db=# SELECT t.City, t.District, t.Country, C.language
World_db=# FROM t JOIN countrylanguage C on t.CountryCode = C.countrycode
World_db=# WHERE t.City='Rosario' AND C.official='TRUE';
  city   | district | country | language
-----+-----+-----+-----
Rosario | Santa Fé | Argentina | Spanish
(1 row)

```

Figure 6.1: A non-recursive Common Table Expression

Recursive Common Table Expressions

A recursive CTE references itself usually via a WITH clause referring to its own output. To better illustrate through an example, we are going to create a new database and table named College and CollegeClasses, respectively, and populate the former with dummy data as follows:

```

CREATE TABLE CollegeClasses (
  ClassID serial PRIMARY KEY,
  ClassDescription VARCHAR NOT NULL,
  ClassParentID INT
);

INSERT INTO CollegeClasses (
  ClassDescription,
  ClassParentID
)
VALUES
('Calculus 1', NULL),
('Algebra 1', 1),
('Analytic Geometry', 1),
('Physics 1', 1),
('Statistics', 1),
('Algebra 2', 2),
('Discrete Math', 2),
('Programming 1', 2),

```

```
('Programming 2', 2),
('Advanced Geometry', 3),
('Control systems', 3),
('English as a Second Language 1', 3),
('Literature', 3),
('Physics 2', 4),
('Calculus 2', 4),
('Graphs and Math', 7),
('English as a Second Language 2', 7),
('Basic algorithms', 8),
('Advanced algorithms', 8),
('Programming with C', 8);
```

In this case we're interested in retrieving a list of classes and their children down to a given level. For example, we will start with Algebra 1 (ClassID=2) and descend down to the last class that depends on it:

```
WITH RECURSIVE classes AS (
SELECT
ClassID,
ClassParentID,
ClassDescription
FROM
CollegeClasses
WHERE
ClassID = 2
UNION
SELECT
e.ClassID,
e.ClassParentID,
e.ClassDescription
FROM
CollegeClasses e
INNER JOIN classes s ON s.ClassID = e.ClassParentID
) SELECT * FROM classes;
```

This query, as in the previous section, deserves a detailed explanation. Let's begin by saying a recursive CTE consists of 4 components:

#1 - A non-recursive query. In this case, it is a query to retrieve the CollegeClass information where ClassID=2:

```
SELECT ClassID, ClassParentID, ClassDescription
FROM CollegeClasses WHERE ClassID = 2
```

#2 - The UNION or UNION ALL operator. Any of these operators allows us to combine one or more result sets into a single one. The choice of one above the other will depend on whether you want to avoid duplicates (if any) or return them, respectively.

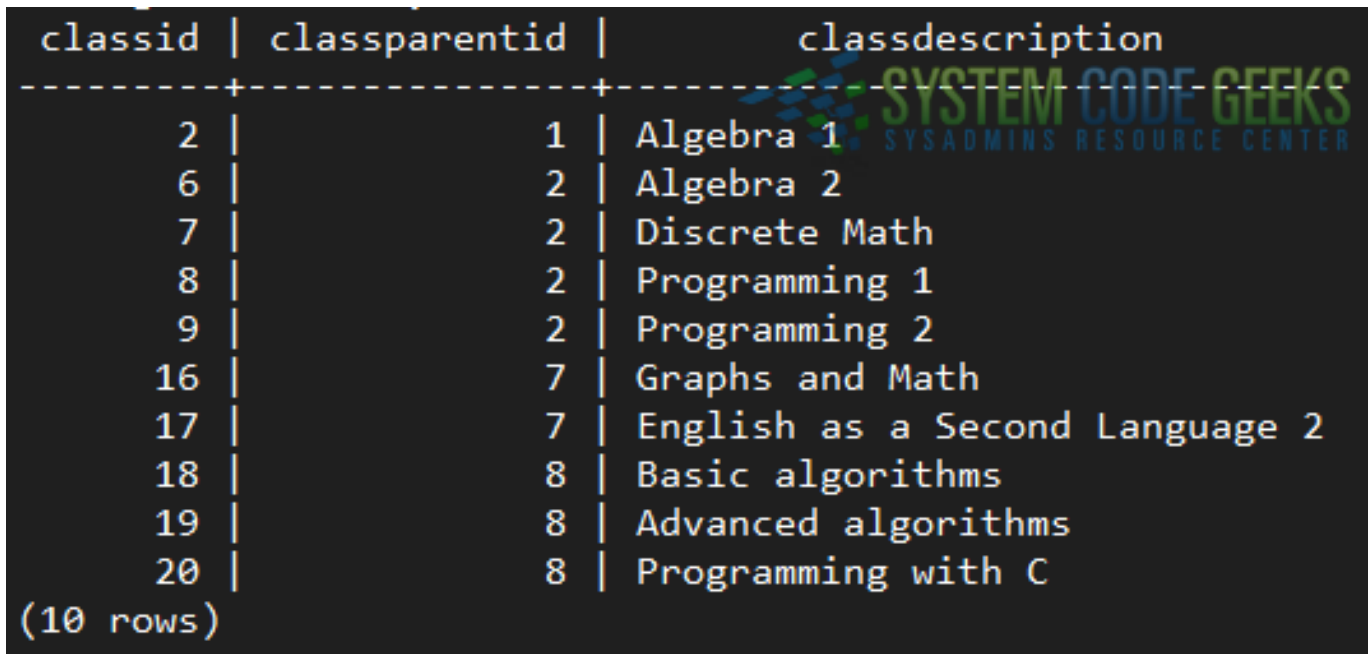
#3 - The recursive term. Note that the classes temporary table references itself in this part of the CTE:

```
SELECT e.ClassID, e.ClassParentID, e.ClassDescription
FROM CollegeClasses e INNER JOIN classes s ON s.ClassID = e.ClassParentID
```

#4 - The final statement, which is executed once the iterations in Part 3 have finished. In this case,

```
SELECT * FROM classes;
```

That said, let's take a look at the result of the query (see Fig. 6.2) and examine it to see if it meets our expectations:



classid	classparentid	classdescription
2	1	Algebra 1
6	2	Algebra 2
7	2	Discrete Math
8	2	Programming 1
9	2	Programming 2
16	7	Graphs and Math
17	7	English as a Second Language 2
18	8	Basic algorithms
19	8	Advanced algorithms
20	8	Programming with C

(10 rows)

Figure 6.2: A recursive CTE

As we can see, the result set begins with ClassID=2, and shows all its children (ClassID=6, 7, 8, and 9). Then it shows all the children of this last set as well.

You will want to use a recursive CTE if you need to retrieve information in the form of a hierarchical tree. It is precisely the keyword `RECURSIVE` at the top of the query which allows the CTE to reference itself.

6.3 Summary

In this article we have explained how to create recursive and non-recursive Common Table Expressions in PostgreSQL. As you pursue the study of this topic, keep in mind that using CTEs is not a matter of improving performance, but readability and maintainability.

Chapter 7

Hot-Standby Database Replication Tutorial

In [the previous articles](#) of this series, we have learned several PostgreSQL database management skills. So far, those skills have only included working with one database on one machine only. Today we will explain how to set up database replication, with a master machine and a slave one, in order to provide redundancy.

In particular, we will use an approach known as *hot standby*, that allows to run read-only queries on a replicated database (residing on a separate machine) while the main one is under maintenance. Hot Standby is available in PostgreSQL starting with version 9.0.

Given the nature of the topic at hand, we will need an extra Ubuntu server. We will call it “slave” and will change its hostname to `ubuntu-slave` and set its IP address to `192.168.0.55`. The master server (`192.168.0.54` - the one we have been using until now) will be renamed to `ubuntu-master`. We will begin this article by outlining step by step the prerequisites for the setup, so you should not run into any significant issues.

If you’re using a VirtualBox-based VM to follow along with this series, you can easily clone it as we explained in [Cloning, exporting, importing, and removing virtual machines in VirtualBox](#) (don’t forget to check the **Reinitialize the MAC address of all network cards** box). Otherwise, you may need to install an Ubuntu 16.04 server instance from scratch.

7.1 Step 0 - Change hostnames and IP addresses as needed

In the master machine, do:

```
hostnamectl set-hostname ubuntu-master
```

Next, edit `/etc/hosts` as follows:

```
127.0.0.1    ubuntu-master
192.168.0.55 ubuntu-slave
```

In the slave machine:

```
hostnamectl set-hostname ubuntu-slave
```

Then edit `/etc/network/interfaces` and make sure the configuration for `enp0s3` (the main NIC) looks as follows:

```
iface enp0s3 inet static
address 192.168.0.55
netmask 255.255.255.0
gateway 192.168.0.1
dns-nameservers 8.8.8.8 8.8.4.4
```

Next, edit `/etc/hosts` as follows:

```
127.0.0.1      ubuntu-slave
192.168.0.54   ubuntu-master
```

Finally, restart the network service on both machines

```
systemctl restart networking
```

and logout, then log back in to apply changes.

Now we're ready to talk.

7.2 Step 1 - Configuring the master

To begin, we will create a dedicated user (repuser in this case) and we will limit the number of simultaneous connections to 1. Enter the `psql` command prompt and do:

```
CREATE USER repuser REPLICATION LOGIN CONNECTION LIMIT 1 ENCRYPTED PASSWORD 'rep4scg';
```

In `/etc/postgresql/9.5/main/postgresql.conf`, make sure the following settings and values are included:

```
listen_addresses = 'localhost,192.168.0.54'
wal_level = 'hot_standby'
max_wal_senders = 1
hot_standby = on
```

and in `/etc/postgresql/9.5/main/pg_hba.conf`:

```
hostssl      replication repuser  192.168.0.55  md5
```

Next, switch to user `postgres`, generate a public key and copy it to the slave. This will allow the master to replicate automatically to the slave:

```
ssh-keygen -t rsa
ssh-copy-id 192.168.0.55
```

When prompted to enter the password for user `postgres` in the slave machine, do so before proceeding.

Now restart the database service:

```
sudo systemctl restart postgresql
```

7.3 Step 2 - Configuring the slave

Make sure the database service is stopped before proceeding. Otherwise, you're in for a nasty database corruption in a few moments.

```
systemctl stop postgresql
```

Then edit `/etc/postgresql/9.5/main/postgresql.conf` and make sure the following settings / values are included:

```
listen_addresses = 'localhost,192.168.0.55'
wal_level = 'hot_standby'
max_wal_senders = 1
hot_standby = on
```

Finally, add the following line to `/etc/postgresql/9.5/main/pg_hba.conf`:

```
hostssl      replication repuser  192.168.0.54  md5
```


7.4 Step 3 - Performing the replication

This step consists in two sub-steps:

3a- In the master, run the following commands to start an initial backup of all databases currently residing in our server, excluding the logs (you can choose a different backup identification instead of Initial backup). If the destination files exist, they will be updated in place; additionally, the data will be compressed during the transfer - this may come in handy if you have several large databases (otherwise, feel free to ignore the `-z` option of `rsync`).

```
psql -c "select pg_start_backup('Initial backup');"
rsync -cvaz --inplace --exclude=*pg_xlog* /var/lib/postgresql/9.5/main/ 192.168.0.55:/var/ ↵
lib/postgresql/9.5/main/
psql -c "select pg_stop_backup();"

```

3b- In the slave, create a `.conf` file with the connection info to the master server. We will name it `recovery.conf` and save it in `/var/lib/postgresql/9.5/main/`:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.0.54 port=5432 user=repuser password=rep4scg'

```

where **user** and **password** need to match the credentials created at the beginning of Step 1.

Now we can proceed to start the database server in the slave:

```
sudo systemctl start postgresql

```

and test the replication in the next step.

7.5 Step 4 - Testing the replication

In the master, we will switch to user `postgres` and execute a simple query to `SELECT` and then update a record from the `city` table in the `World_db` database. At the same time, we will query that same record in the slave before and after performing the `UPDATE` in the master. Refer to Fig. 7.1 for more details:

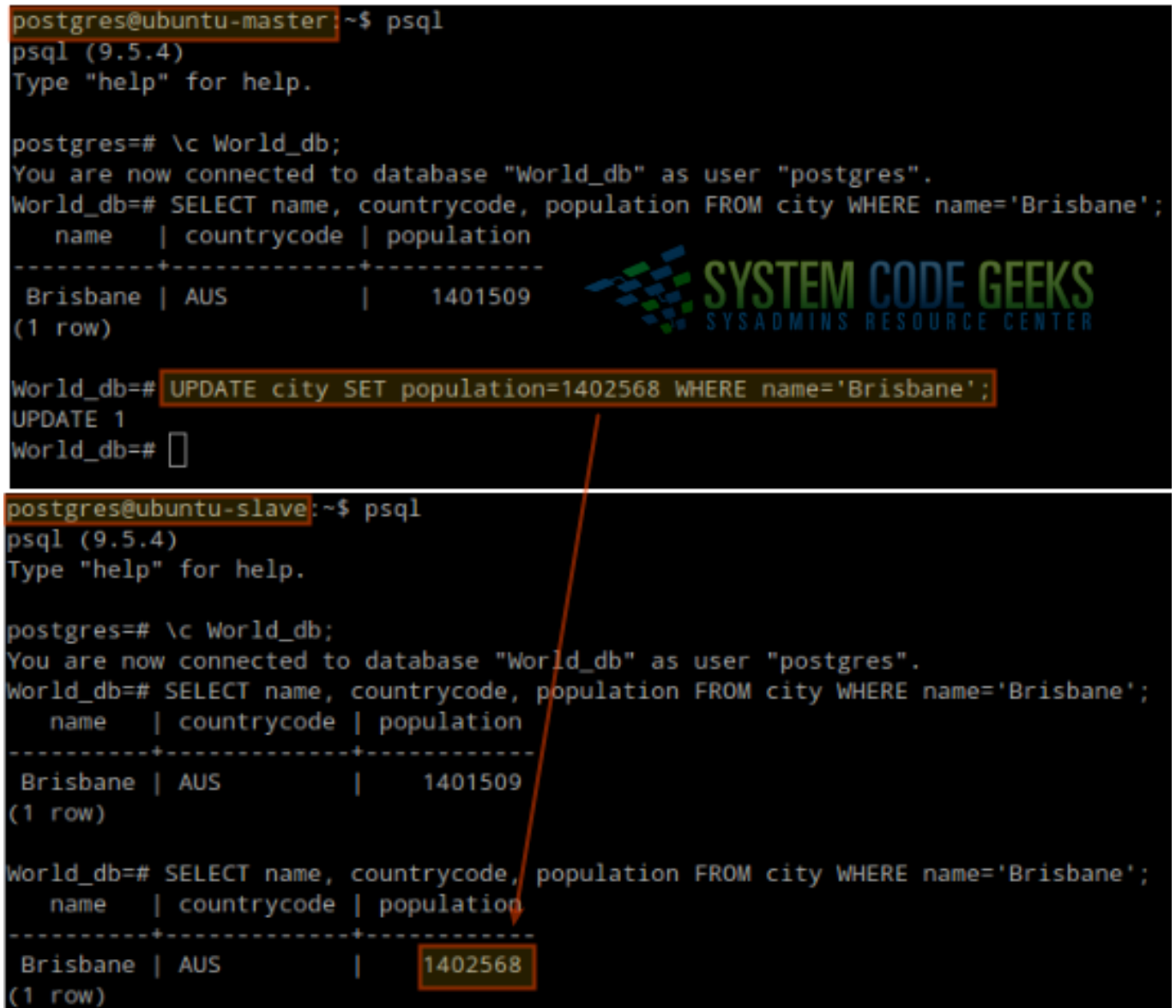
```
sudo -i -u postgres
psql
\c World_db;

```

then

```
SELECT name, countrycode, population FROM city WHERE name='Brisbane';
UPDATE city SET population=1402568 WHERE name='Brisbane';

```



The screenshot shows two terminal windows. The top window is on the master node (postgres@ubuntu-master) and the bottom window is on the slave node (postgres@ubuntu-slave). Both are connected to the 'World_db' database. The master window shows a SELECT query for 'Brisbane' with a population of 1401509, followed by an UPDATE query to change the population to 1402568. The slave window shows the same SELECT query, but the population is now 1402568, indicating successful replication. An orange arrow points from the UPDATE statement in the master window to the updated population value in the slave window's output.

```
postgres@ubuntu-master:~$ psql
psql (9.5.4)
Type "help" for help.

postgres=# \c World_db;
You are now connected to database "World_db" as user "postgres".
World_db=# SELECT name, countrycode, population FROM city WHERE name='Brisbane';
   name   | countrycode | population 
-----+-----+-----
Brisbane | AUS         |    1401509 
(1 row)

World_db=# UPDATE city SET population=1402568 WHERE name='Brisbane';
UPDATE 1
World_db=#
```

```
postgres@ubuntu-slave:~$ psql
psql (9.5.4)
Type "help" for help.

postgres=# \c World_db;
You are now connected to database "World_db" as user "postgres".
World_db=# SELECT name, countrycode, population FROM city WHERE name='Brisbane';
   name   | countrycode | population 
-----+-----+-----
Brisbane | AUS         |    1401509 
(1 row)

World_db=# SELECT name, countrycode, population FROM city WHERE name='Brisbane';
   name   | countrycode | population 
-----+-----+-----
Brisbane | AUS         |    1402568 
(1 row)
```

Figure 7.1: Database replication in action

As you can see, the slave was updated automatically the population for Brisbane was changed in the master. If you attempt to perform an UPDATE from the slave, it will fail with the following error: **“ERROR: cannot execute UPDATE in a read-only transaction.”**

7.6 Troubleshooting

If the database service refuses to start successfully, you will not be able to run psql in the Linux command line. In that case, you will have to troubleshoot using the following resources:

```
systemctl -l status postgresql@9.5-main
journalctl -xe
tail -f /var/log/postgresql/postgresql-9.5-main.log
```

That’s all folks! You should have a PostgreSQL hot standby replication in place.

Chapter 8

Backup, Restore and Migration

Since hardware can fail and human error may occur, having frequent backups and knowing how to restore them are important skills that every system administrator should have. Additionally, you will need to know how to migrate PostgreSQL databases from one machine to another in case you purchase new, more powerful servers. Thus, in this tutorial we will discuss how to perform these critical operations using a test environment with two Ubuntu 16.04 (server edition) virtual machines. We will name these VMs **newserver** (192.168.0.54) and **oldserver** (192.168.0.55), where the same PostgreSQL version (9.5) has been installed on both. We assume we will migrate the `World_db` database on **oldserver** over to **newserver**.

8.1 Backup, restore, and migration strategies

Traditionally, PostgreSQL database administrators used shell scripts and cron jobs to back up their databases. Although this approach was considered efficient a decade (or so) ago, today there are tools that make this process hassle-free and easier to maintain. Among these tools, Barman (Backup and Recovery Manager), a Python-based open source solution developed and maintained by [2ndQuadrant](#) (a firm that specializes in PostgreSQL services) stands out.

8.2 Installing Barman

More accurately, Barman is a backup, restore, and disaster recovery tool for PostgreSQL. We will install it on the virtual machine that we called **newserver** (192.168.0.54) to migrate the databases from **oldserver** (192.168.0.55).

That said, let's install Barman:

```
sudo aptitude update && sudo aptitude install barman
```

Once the installation has completed successfully, proceed with the following steps.

8.2.1 Step 1 - Create a dedicated PostgreSQL user in oldserver

In order for barman (which has been installed in **newserver**) to communicate with the PostgreSQL instance running on **oldserver**, we need to create a dedicated database user. To do so, run the following command as **postgres** on **oldserver** and enter the desired password for the new database user. Also, when you're prompted to confirm if the account should have superuser privileges, enter `y` and press Enter

```
createuser --interactive -P barman
```

Then test the connection from **newserver**. We will check the connection against the `postgres` database, but you can use other database (in that case, you'll have to modify the SQL query inside single quotes):

```
psql -c 'SELECT version()' -U barman -h 192.168.0.55 postgres
```

Refer to Fig. 8.1 for details:



```
gacanepa@oldserver:~$ sudo -i -u postgres
[sudo] password for gacanepa:
postgres@oldserver:~$ createuser -s -W barman
Password:
postgres@oldserver:~$ psql -U barman
psql: FATAL: Peer authentication failed for user "barman"
postgres@oldserver:~$ dropuser barman
postgres@oldserver:~$ createuser --interactive -P barman
Enter password for new role:
Enter it again:
Shall the new role be a superuser? (y/n) y
postgres@oldserver:~$ psql -U barman -h localhost World_db
Password for user barman:
psql (9.5.4)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256
compression: off)
Type "help" for help.
World_db=# \q
postgres@oldserver:~$
```

```
gacanepa@newserver:~$ sudo -i -u postgres
postgres@newserver:~$ psql -c 'SELECT version()' -U barman -h 192.168.0.55 postgres
Password for user barman:
postgres@newserver:~$
```

```
-----
PostgreSQL 9.5.4 on i686-pc-linux-gnu, compiled by gcc
4.0 20160609, 32-bit
(1 row)
```

Figure 8.1: Creating a dedicated user account and testing the connection

Throughout this article, we will use the word Barman to refer to the program itself, whereas the all-lowercase barman will represent either the command associated with the program or an account.

8.2.2 Step 2 - Create the .pgpass file in newserver

As part of the installation of Barman on **newserver**, a Linux account called barman was created. To set its password, do

```
sudo passwd barman
```

and enter the desired password. Then, switch user to barman:

```
sudo -i -u barman
```

and create the .pgpass file for user barman:

```
echo "192.168.0.55:5432:*:barman:password" >> ~/.pgpass
```

The actual format for the .pgpass file is hostname:port:database:username:password. If an asterisk is used in any of the first four fields, it will match everything. Please note that username here represents the PostgreSQL user we created in Step 1, not the Linux account we just referred to. The official documentation for this file can be found [here](#).

This file can contain passwords to be used if a connection requires one (in this case, barman will use it to talk to the PostgreSQL instance on **oldserver**).

8.2.3 Step 3 - Set up key-based authentication

In order to perform backups without user intervention we will need to set up and copy SSH keys for passwordless authentication. Barman will make use of this method to copy data through rsync.

On **newserver**, switch to user barman and generate the keys

```
ssh-keygen -t rsa
```

(choose the default destination file for the public key and an empty passphrase).

Next, copy the public key to the authorized keys of user postgres on **oldserver**:

```
ssh-copy-id postgres@192.168.0.55
```

This will allow barman on **newserver** to connect to **oldserver** as user postgres. To test if the connection can be made without password, as expected, you can run the following command (on success, it will not return anything):

```
ssh postgres@192.168.0.55 -C true
```

You'll also need to allow barman to SSH into localhost as the local user postgres:

```
ssh-copy-id postgres@localhost  
ssh postgres@localhost -C true
```

Finally, on **oldserver** log in as postgres and do

```
ssh-keygen -t rsa
```

and copy the resulting key to the list of authorized keys for user barman on newserver:

```
ssh-copy-id barman@192.168.0.54
```

Again, test the connection before proceeding:

```
ssh barman@192.168.0.54 -C true
```

8.2.4 Step 4 - Configure Barman

On **newserver**, open the Barman main configuration file (/etc/barman.conf) and uncomment this line by removing the leading semicolon:

```
;configuration_files_directory = /etc/barman.d
```

should read

```
configuration_files_directory = /etc/barman.d
```

(if /etc/barman.d does not exist, you'll have to create it with `mkdir /etc/barman.d`)

And create a file named oldserver.conf with the following contents (the word inside square brackets represents the name that barman will use to identify the connection details):

```
[oldserver]  
description = "Our old PostgreSQL server"  
conninfo = host=192.168.0.55 user=barman dbname=World_db  
ssh_command = ssh postgres@192.168.0.55  
retention_policy = RECOVERY WINDOW OF 2 WEEKS
```

where most variables are self-explanatory with the exception of **retention_policy**. This variable is used to determine for how long backups should be kept (2 weeks in this case). This should be modified based on the expected activity and growth of the database, and the available space on the filesystem where backups will be kept.

8.2.5 Step 5 - Configure PostgreSQL

On **oldserver**:

Add this line to `/etc/postgresql/9.5/main/pg_hba.conf`:

```
host      all             all             192.168.0.54/24      trust
```

Then make sure the following variables on `/etc/postgresql/9.5/main/postgresql.conf` have the indicated values:

```
wal_level = archive
archive_mode = on
archive_command = 'rsync -a %p barman@192.168.0.54:/var/lib/barman/oldserver/incoming/%f'
```

As you will probably guess, the directory in the `rsync` connection string represents the directory where the backup files for **oldserver** will be kept on **newserver**.

On **newserver**, make sure the following variable on `/etc/postgresql/9.5/main/postgresql.conf` has the indicated value:

```
data_directory = '/var/lib/postgresql/9.5/data'
```

If the directory called **data** does not exist under `/var/lib/postgresql/9.5`, create it before proceeding (that is where the data files will be stored on **newserver**)

Then restart the `postgresql` service to activate the latest changes:

```
sudo systemctl restart postgresql
```

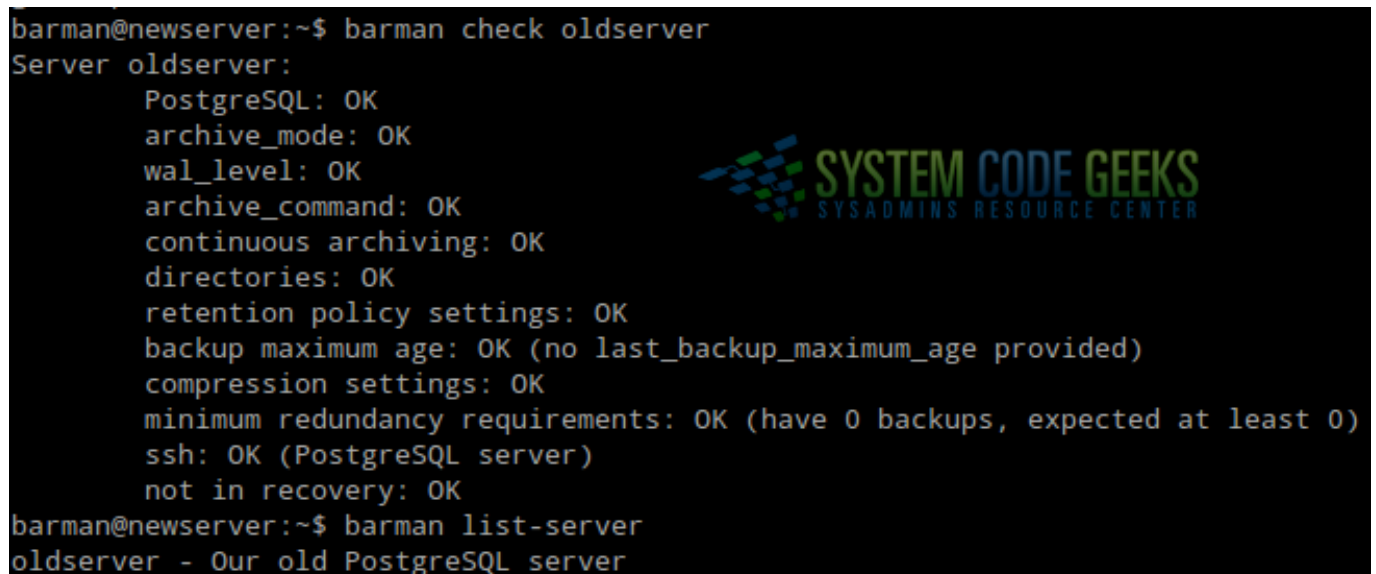
8.2.6 Step 6 - Test the Barman configuration

Once PostgreSQL has been configured on **oldserver** to allow connections from **newserver**, we are ready to test the configuration. To do so, switch to user `barman` on **newserver** and do

```
barman check oldserver
barman list-server
```

The first command will check the SSH and PostgreSQL connections, whereas the second one will show the list of configured PostgreSQL servers we wish to back up.

The output should be as follows (see Fig. 8.2):



```
barman@newserver:~$ barman check oldserver
Server oldserver:
  PostgreSQL: OK
  archive_mode: OK
  wal_level: OK
  archive_command: OK
  continuous archiving: OK
  directories: OK
  retention policy settings: OK
  backup maximum age: OK (no last_backup_maximum_age provided)
  compression settings: OK
  minimum redundancy requirements: OK (have 0 backups, expected at least 0)
  ssh: OK (PostgreSQL server)
  not in recovery: OK
barman@newserver:~$ barman list-server
oldserver - Our old PostgreSQL server
```

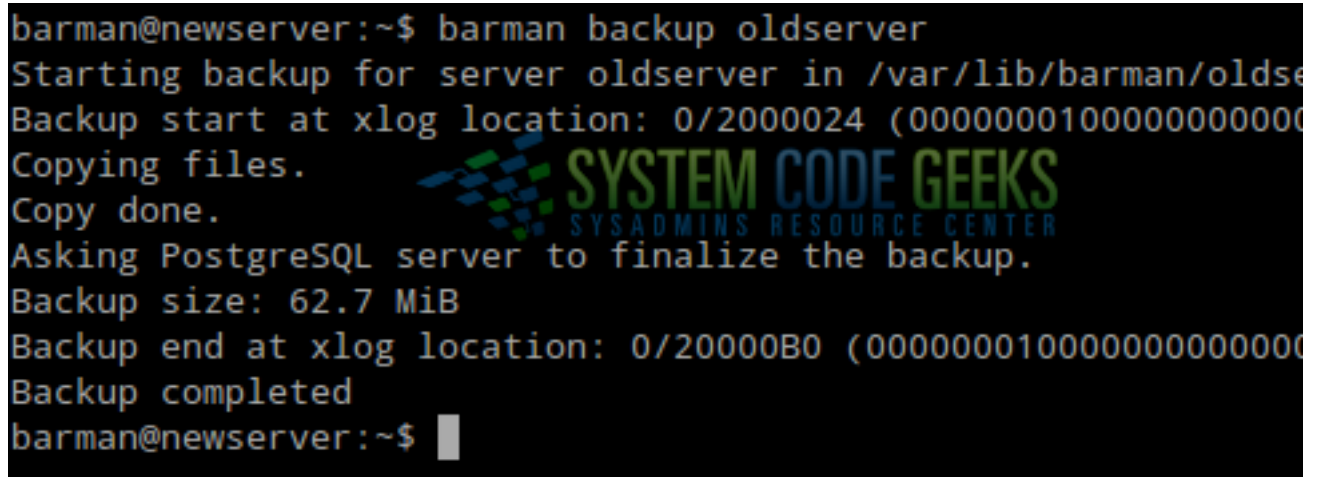
Figure 8.2: Checking the barman connection from newserver to oldserver

8.2.7 Step 7 - Perform the backup

Once all of the items in the output of `barman check oldserver` return OK, we are ready to perform our first backup with the following command (see Fig. 8.3):

```
barman backup oldserver
```

The output should be similar to Fig. 8.3:

A terminal window screenshot showing the execution of the 'barman backup oldserver' command. The output includes: 'Starting backup for server oldserver in /var/lib/barman/oldse', 'Backup start at xlog location: 0/2000024 (00000001000000000000)', 'Copying files.', 'Copy done.', 'Asking PostgreSQL server to finalize the backup.', 'Backup size: 62.7 MiB', 'Backup end at xlog location: 0/20000B0 (00000001000000000000)', and 'Backup completed'. A watermark for 'SYSTEM CODE GEEKS' is visible in the center. The prompt 'barman@newserver:~\$' is shown at the end.

```
barman@newserver:~$ barman backup oldserver
Starting backup for server oldserver in /var/lib/barman/oldse
Backup start at xlog location: 0/2000024 (00000001000000000000
Copying files.
Copy done.
Asking PostgreSQL server to finalize the backup.
Backup size: 62.7 MiB
Backup end at xlog location: 0/20000B0 (00000001000000000000
Backup completed
barman@newserver:~$
```

Figure 8.3: Creating our first backup with barman

Once the backup has completed we can identify it with the help of

```
barman list-backup oldserver
```

which will list all the backups we have performed for oldserver. To view details about a specific backup, we'll use

```
barman show-backup oldserver backup_id
```

where `backup_id` is the backup identification (**20161015T120420** in Fig. 8.4).

```
barman@newserver:~$ barman list-backup oldserver
oldserver 20161015T120420 - Sat Oct 15 12:04:32 2016 - Size: 62.7 MiB -
barman@newserver:~$ barman show-backup oldserver 20161015T120420
Backup 20161015T120420:
  Server Name      : oldserver
  Status           : DONE
  PostgreSQL Version : 90504
  PGDATA directory : /var/lib/postgresql/9.5/main

Base backup information:
  Disk usage      : 62.7 MiB (62.7 MiB with WALs)
  Incremental size : 62.7 MiB (-0.00%)
  Timeline        : 1
  Begin WAL       : 00000001000000000000000002
  End WAL         : 00000001000000000000000002
  WAL number      : 0
  Begin time      : 2016-10-15 12:04:19.575591-03:00
  End time        : 2016-10-15 12:04:32.751555-03:00
  Begin Offset    : 36
  End Offset      : 176
  Begin XLOG      : 0/2000024
  End XLOG        : 0/20000B0

WAL information:
  No of files     : 0
```




Figure 8.4: Checking backups

8.2.8 Step 8 - Restore the backup on newserver

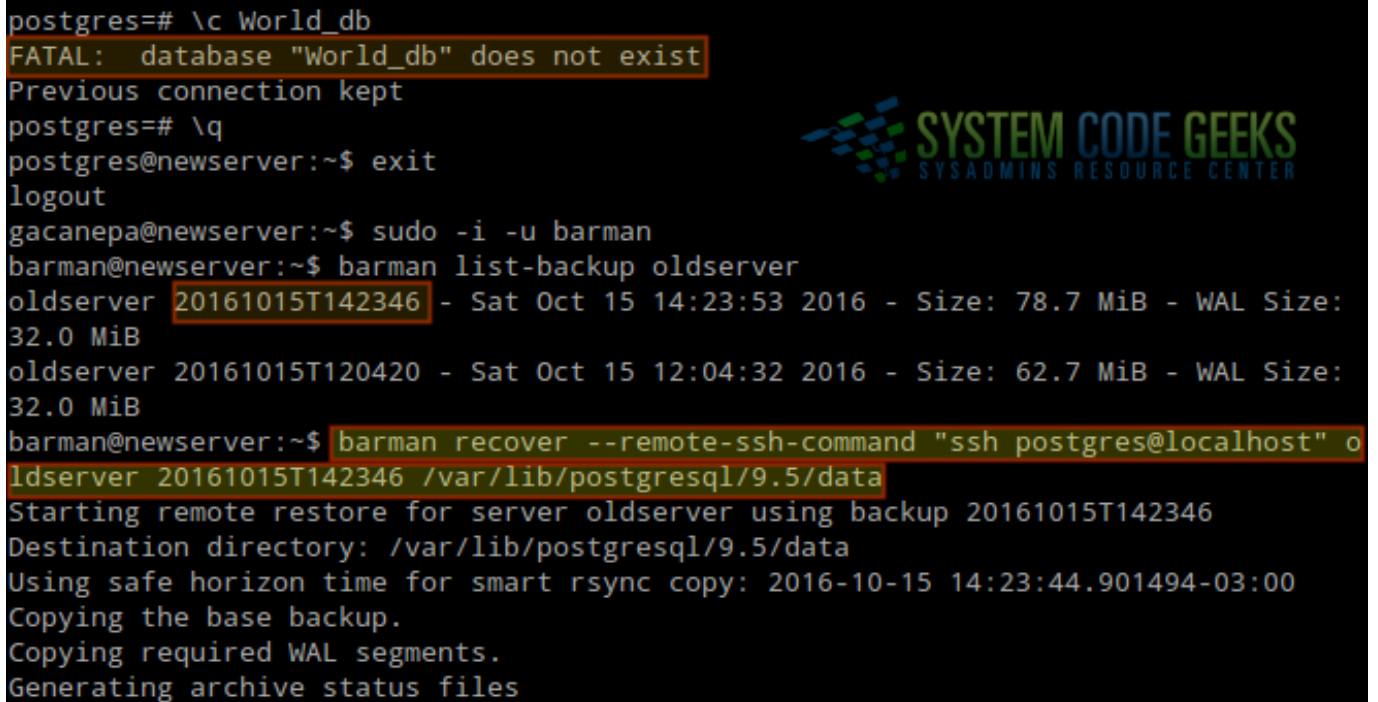
As we can see in Fig. 8.5, the World_db database can't be found on **newserver**. To migrate a backup, we will stop the postgresql service

```
sudo systemctl stop postgresql
```

and run the following command as user barman:

```
barman recover --remote-ssh-command "ssh postgres@localhost" oldserver 20161015T142346 /var ↵
/lib/postgresql/9.5/data
```

Note how barman makes use of the SSH keys to connect as user postgres to localhost in order to load the backup with id **20161015T142346** to the data directory. The result is shown in Fig. 8.5:



```
postgres=# \c World_db
FATAL: database "World_db" does not exist
Previous connection kept
postgres=# \q
postgres@newserver:~$ exit
logout
gacanepa@newserver:~$ sudo -i -u barman
barman@newserver:~$ barman list-backup oldserver
oldserver 20161015T142346 - Sat Oct 15 14:23:53 2016 - Size: 78.7 MiB - WAL Size: 32.0 MiB
oldserver 20161015T120420 - Sat Oct 15 12:04:32 2016 - Size: 62.7 MiB - WAL Size: 32.0 MiB
barman@newserver:~$ barman recover --remote-ssh-command "ssh postgres@localhost" oldserver 20161015T142346 /var/lib/postgresql/9.5/data
Starting remote restore for server oldserver using backup 20161015T142346
Destination directory: /var/lib/postgresql/9.5/data
Using safe horizon time for smart rsync copy: 2016-10-15 14:23:44.901494-03:00
Copying the base backup.
Copying required WAL segments.
Generating archive status files
```

Figure 8.5: Restoring a backup with barman

After the recovery or migration is complete, start the postgresql service with a user with sudo access

```
sudo systemctl start postgresql
```

and check the World_db database as user postgres:

```
sudo -i -u postgres
psql
\c World_db
```

Now let's run queries against the database, as shown in Fig. 8.6:

```
gacanepa@newserver:~$ sudo systemctl restart postgresql
gacanepa@newserver:~$ sudo -i -u postgres
postgres@newserver:~$ psql
psql (9.5.4)
Type "help" for help.

postgres=# \c World_db
You are now connected to database "World_db" as user "postgres".
World_db=# SELECT * FROM city WHERE name='Rosario';
 id | name    | countrycode | district | population
-----+-----+-----+-----+-----
 72 | Rosario | ARG         | Santa Fé |    971258
(1 row)

World_db=#
```




Figure 8.6: Querying the database we migrated

Congratulations! You have successfully set up a very effective method to back up, restore, and migrate PostgreSQL databases.

8.3 Automating backups

In order to automate the backup process, switch to user barman and open the crontab file:

```
sudo -i -u barman
crontab -e
```

Then add the following two lines in it in order to execute a backup of oldserver each day at 12:45 pm

```
45 12 * * * /usr/bin/barman backup oldserver
```

Please note that this is a basic Barman / PostgreSQL setup, so I strongly suggest to check the official Barman docs [here](#).

Chapter 9

Connect to PostgreSQL using PHP

After we have learned how to set up and configure PostgreSQL for a variety of scenarios, having a database and populating it with data will not be of any use until we can retrieve it and use it in some way. Today, using a mobile-friendly web application is the most common way to accomplish this goal.

In this tutorial we will explain how to connect to our PostgreSQL database server using PHP, a popular server-side scripting language, how to retrieve data, and how to display it in a web page. Using this foundation, you will be able to go on to create more robust applications that make use of PostgreSQL and PHP.

9.1 Installing the software

As we just mentioned, we will use PHP to connect to the database server and to display the results of a query in a web page. Before we even start writing the application, we will need to install PHP and some additional packages - including the Apache web server. To do this in an Ubuntu 16.04 server with IP 192.168.0.54, use the following command:

```
sudo aptitude update && sudo aptitude install apache2 postgresql-contrib php7.0-pgsql
```

After the installation is complete, create a php file named `info.php` under `/var/www/html` with the following three lines. This will help us to verify that PHP has been installed along with the PostgreSQL dependencies:

```
<?php
phpinfo();
?>
```

Then browse to `192.168.0.54/info.php` and look for the section with the PostgreSQL details. You should find that the PDO driver is enabled and that PHP is supporting our RDBMS, as shown in Fig. 9.1:

PDO

PDO support	enabled
PDO drivers	pgsql

pdo_pgsql

PDO Driver for PostgreSQL	enabled
PostgreSQL(libpq) Version	9.5.4
Module version	7.0.8-0ubuntu0.16.04.3
Revision	\$Id: f9b0c62eba234361d62f16fcbaaa120353ab5175 \$

**pgsql**

PostgreSQL Support	enabled
PostgreSQL(libpq) Version	9.5.4
PostgreSQL(libpq)	PostgreSQL 9.5.4 on i686-pc-linux-gnu, compiled by gcc (Ubuntu 5.4.0-6ubuntu16.0609, 32-bit
Multibyte character support	enabled
SSL support	enabled

Figure 9.1: Checking the status of PostgreSQL-related PHP components

Now we're ready to start writing our simple, yet functional PHP-based application.

9.2 Connecting to the database server

The first thing that we must do is ensure PHP can connect to the database server. Create a file named `con.php` under `/var/www/html` with the following contents:

```
<?php

// Connection details
$conn_string = "host=localhost port=5432 dbname=World_db user=scg password=MyPassword ↔
options='--client_encoding=UTF8'";

// Establish a connection with MySQL server
$dbconn = pg_connect($conn_string);

// Check connection status. Exit in case of errors
if(!$dbconn) {
    echo "Error: Unable to open database\n";
} else {
    echo "Opened database successfully\n";
}

// Close connection
pg_close($dbconn);

?>
```

For security purposes, set the appropriate ownership to the Linux account `postgres` (the user the database service runs as) and add the `www-data` account to the `postgres` group. This will allow Apache to read this file:

```
sudo chown postgres:postgres /var/www/html/con.php  
  
sudo chmod 660 /var/www/html/con.php  
sudo usermod -a -G postgres www-data
```

Now go to 192.168.0.54/con.php and make sure the connection to the database is successful before proceeding:

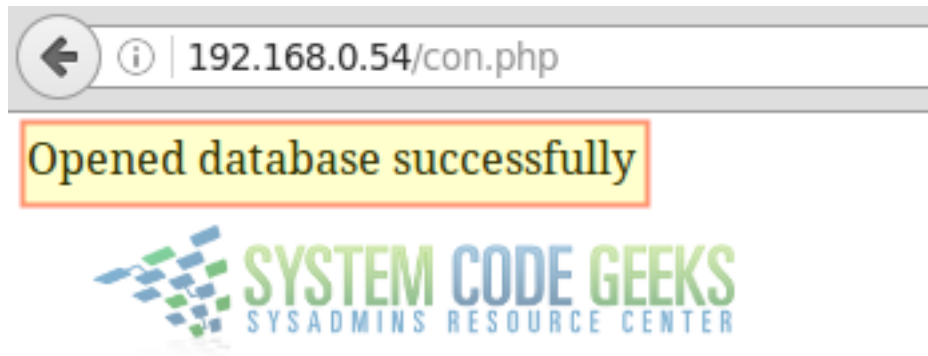


Figure 9.2: Verifying database connection via PHP

If you get a blank page instead of the confirmation message shown in Fig. 2, inspect the Apache logs to troubleshoot. A missing semicolon or a misplaced quote can cause the connection to fail.

9.3 Writing the application

To begin, we will comment out the following line in con.php:

```
echo "Opened database successfully\n";
```

and insert the following lines below it. Please note that we will use a very simple query that will retrieve city names and the district it belongs to in Argentina (you will later be able to change it to a more complicated query using Common Table Expressions, for example):

```
$query = "SELECT name, district FROM city WHERE countrycode='ARG'";  
$cities = pg_query($query) or die('Query failed: ' . pg_last_error());  
$myarray = array();  
while ($row = pg_fetch_assoc($cities)) {  
    $myarray[] = $row;  
}  
  
// Encode response into JSON array  
echo json_encode($myarray);
```

The con.php file should now look as seen in Fig. 9.3:

```
<?php
// Connection details
$conn_string = "host=localhost port=5432 dbname=World_db user=scg password=MyPa

// Establish a connection with MySQL server
$dbconn = pg_connect($conn_string);
// Check connection status. Exit in case of errors
if(!$dbconn) {
    echo "Error: Unable to open database\n";
} else {
    // echo "Opened database successfully\n";
    $query = "SELECT name, district FROM city WHERE countrycode='ARG'";
    $cities = pg_query($query) or die('Query failed: ' . pg_last_error());
    $myarray = array();
    while ($row = pg_fetch_assoc($cities)) {
        $myarray[] = $row;
    }
    // Encode response into JSON array
    echo json_encode($myarray);
}

// Close connection
pg_close($dbconn);
?>
```




Figure 9.3: The connection file

Save the changes and grant privileges on the city table for the scg user. Note that you'll have to do this from the PostgreSQL prompt:

```
GRANT ALL PRIVILEGES ON TABLE city TO scg;
```

Next, go to 192.168.0.54/con.php. You should see the results of the query in JSON format (see Fig. 9.4):

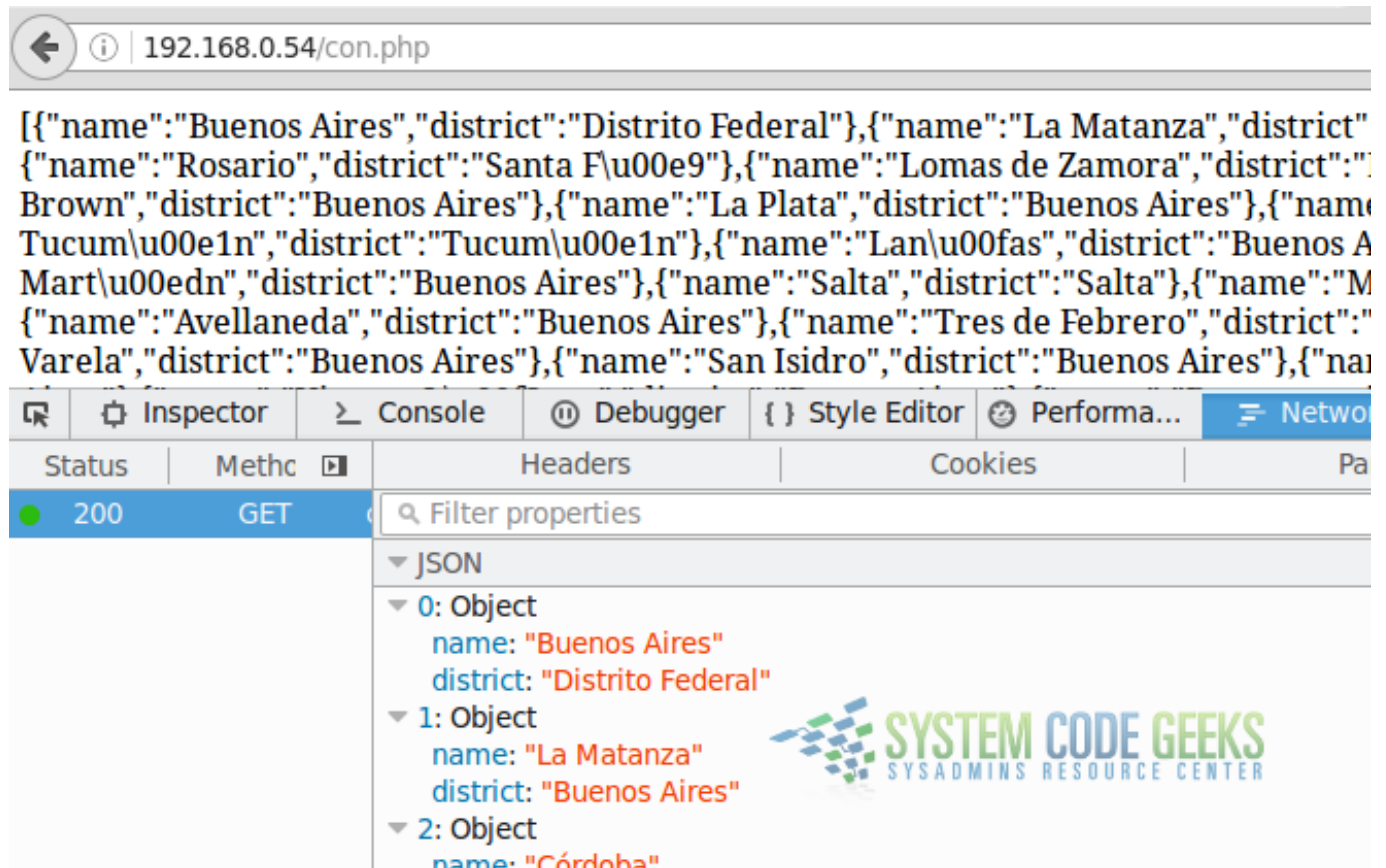


Figure 9.4: Query results in JSON format

JSON stands for *JavaScript Object Notation*. It is a lightweight data-interchange format that is easy for humans to read and write and for machines to parse and generate.

Now that we have successfully 1) connected to the database server, and 2) retrieved records into a JSON array, we will explain how to display this information into a mobile-friendly web page.

9.4 Creating a mobile-friendly web page

Most web developers nowadays use a robust HTML5/CSS/Javascript framework called Bootstrap to write mobile-friendly applications very easily. Though a full discussion about Bootstrap (and the HTML5-related technologies) is out of the scope of this article, it is sufficient to say that one of its distinguishing characteristics is that it divides the viewport in a 12-column grid.

It is up to the developer to decide how many columns will be assigned to a particular piece of content for xs (extra small, i.e. cell phones), sm (small, i.e. tablets and ipads), md (medium, i.e. laptops), and lg or large devices (high resolution monitors). In this tutorial we will assume that we desire to show the city and district fields using 6 columns each in small devices (sm) and up. For extra small screens, city will stack on top of district, as we will see later.

To do this, create a file named `index.php` in the same location as `con.php` and insert the following lines into it:

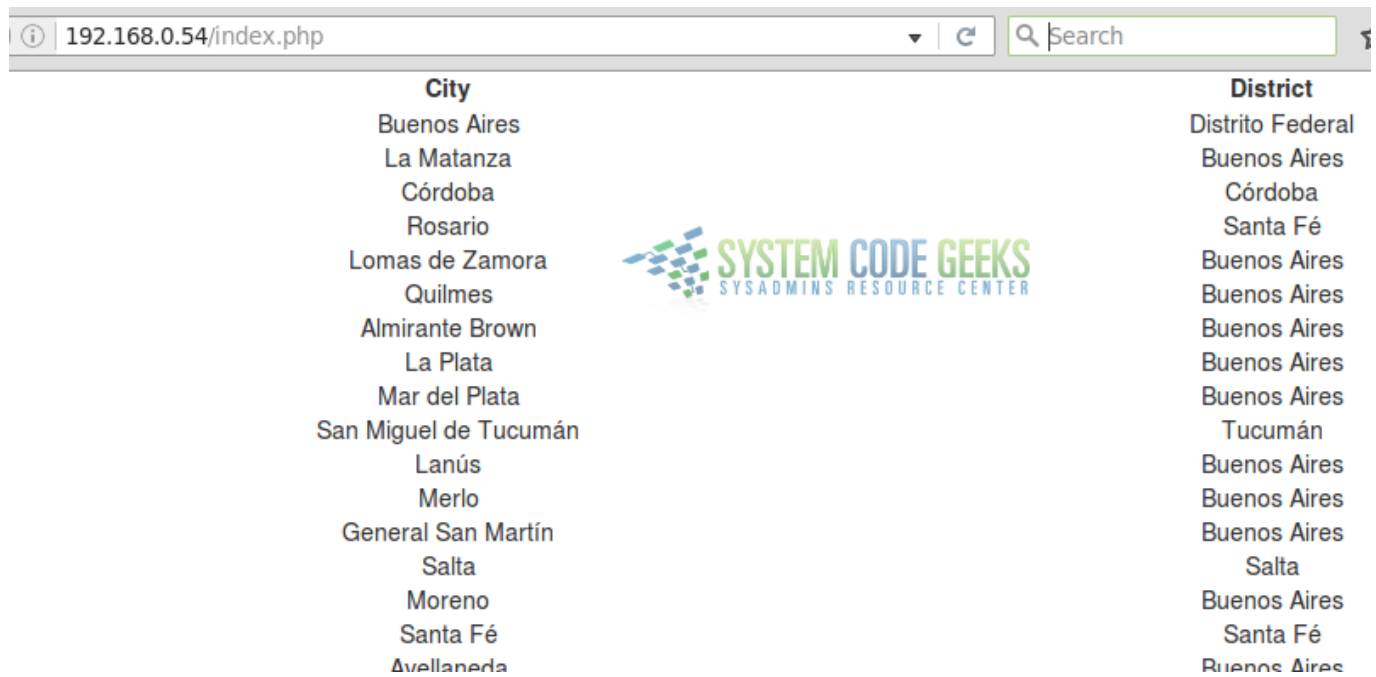
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Mobile friendly page with PostgreSQL and PHP</title>
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
```

```
</head>
<body>

<div class="row">
<div class="col-md-6" id="city" style="text-align: center"><strong>City</strong>
<strong>District</strong>
</div>
</div>
</body>
<script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</html>
<script>
$(document).ready(function() {
$.ajax({
url: 'con.php',
datatype: 'json',
type: 'POST',
success: function(data){
var output = $.parseJSON(data);
for(var i =0;i < output.length;i++)
{
var item = output[i];
$("#city").append("<br>" +item.name);
$("#district").append("<br>" +item.district);
}
}}
);
});
</script>
```

As you can see, this simple page uses a well-known Javascript library called jQuery to make an Ajax call to con.php and retrieve the results. Again, an adequate discussion about jQuery, Ajax, and Javascript is out of the scope of this article, but you can find some very valuable information on [W3schools](#).

When you browse to 192.168.0.54/index.php, the result should be similar to Fig. 9.5:



City	District
Buenos Aires	Distrito Federal
La Matanza	Buenos Aires
Córdoba	Córdoba
Rosario	Santa Fé
Lomas de Zamora	Buenos Aires
Quilmes	Buenos Aires
Almirante Brown	Buenos Aires
La Plata	Buenos Aires
Mar del Plata	Buenos Aires
San Miguel de Tucumán	Tucumán
Lanús	Buenos Aires
Merlo	Buenos Aires
General San Martín	Buenos Aires
Salta	Salta
Moreno	Buenos Aires
Santa Fé	Santa Fé
Avellaneda	Buenos Aires

Figure 9.5: Displaying the web page with the results of the query

Feel free to resize your browser's window to see the visualization changes as the viewport changes.

9.5 Summary

If you followed this tutorial carefully, congratulations! You have set connected to your PostgreSQL server using PHP and displayed data from your database in a mobile-friendly web page. Hopefully this will give you the foundation to create more sophisticated applications.