PostGIS – Users and Permissions – Part 2

In a previous Blog we started to explore the options for managing **Users/Roles** in your PostGIS Database. We looked at creating new **SCHEMAS** and new **ROLES**, and then options for granting **Privileges** to those Roles so that external users could manage our spatial assets.

In **Part 2** we will explore some additional functionality which will allow us to extend our Database Management capabilities, including:-

- Identifying Privileges
- Altering ROLES
- Revoking Privileges
- Listing Tables
- Changing Table Owners
- Viewing Database Connections
- Calculating Database Size
- Identifying Duplicate Records

1 – Identifying Privileges

Having created and managed database privileges in Part One of this blog, we may at some point wish to understand what are those Privileges? Generating a list of Users and their specific privileges will be a useful command to enable you to manage your database Users.

Running the following command:

```
SELECT grantee, privilege_type
FROM information_schema.role_table_grants
WHERE table_name='sssi'
```

Returns a list detailing the **grantee (User)** and their **privilege type** for a specific Table – in this case the **SSSI table** in the environmental Schema. Here we can see that the Postgres User has all privileges and the Environmental User has INSERT, SELECT, UPDATE and DELETE only.

Data Output | Explain | Messages | Query History

	grantee character varying	privilege_type character varying
1	postgres	INSERT
2	postgres	SELECT
3	postgres	UPDATE
4	postgres	DELETE
5	postgres	TRUNCATE
6	postgres	REFERENCES
7	postgres	TRIGGER
8	environmental	INSERT
9	environmental	SELECT
10	environmental	UPDATE
11	environmental	DELETE

This query can be flipped, so that instead of finding the privileges for a specific Table, you can instead list all of the privileges that a **specific User** has for any object in the database.

Running the following command:

```
SELECT table_catalog, table_schema, table_name, privilege_type
FROM   information_schema.table_privileges
WHERE  grantee = 'poi'
```

Will return a list detailing the **Tables** and **Privileges** for the User – **POI.**

Data Output | Explain | Messages | Query History

	table_catalog character varying	table_schema character varying	table_name character varying	privilege_type character varying
1	blog	poi	hospitals	INSERT
2	blog	poi	hospitals	SELECT
3	blog	poi	hospitals	UPDATE
4	blog	poi	hospitals	DELETE
5	blog	poi	schools	INSERT
6	blog	poi	schools	SELECT
7	blog	poi	schools	UPDATE
8	blog	poi	schools	DELETE
9	blog	poi	schools	TRUNCATE
10	blog	poi	schools	REFERENCES
11	blog	poi	schools	TRIGGER

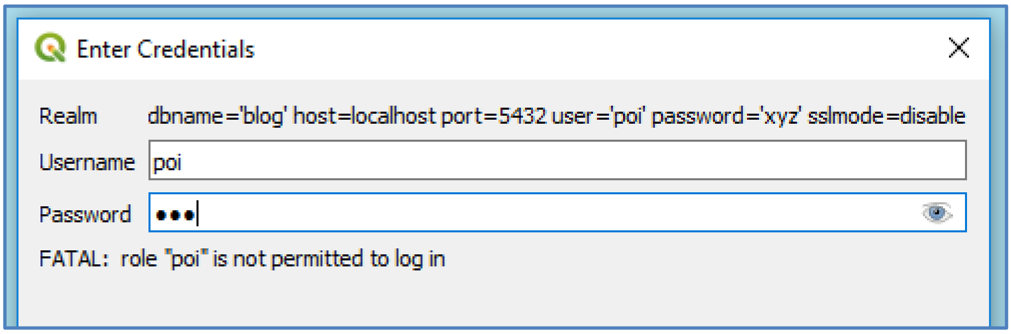
2 – Altering ROLES

Having now identified the privileges for a User/Role, you may need to alter those privileges. For example, a useful update may be to temporarily set the POI User/Role to **not be able to log in** and block their access to the database.

Running the following command:

```
ALTER ROLE poi WITH NOLOGIN;
```

Will mean that if the POI User tries to log into the PostGIS Database, for example via QGIS, it will say **access is denied:**



To then **re-instate** that access you can simply run the following command:

```
ALTER ROLE poi WITH LOGIN;
```

3 – Revoking Privileges

Instead of removing a User's Access rights, you may wish to simply update their privileges to remove an option. This could be to remove the Users rights to UPDATE a table, or even to remove the SELECT On privilege so that the User cannot open that Table.

In this example we will update the rights of the **POI User** so that they can't **SELECT** from the **SSSI** table anymore – because it is in another Schema and they shouldn't have access to that data.

Running this command will identify the privileges that the **POI** currently has:

```
SELECT table_catalog, table_schema, table_name, privilege_type
FROM   information_schema.table_privileges
WHERE  grantee = 'poi'
```

As we can see, they currently have SELECT privileges to the **SSSI** table.

Data Output | Explain | Messages | Query History

	table_catalog character varying	table_schema character varying	table_name character varying	privilege_type character varying
1	blog	poi	hospitals	INSERT
2	blog	poi	hospitals	SELECT
3	blog	poi	hospitals	UPDATE
4	blog	poi	hospitals	DELETE
5	blog	poi	schools	INSERT
6	blog	poi	schools	SELECT
7	blog	poi	schools	UPDATE
8	blog	poi	schools	DELETE
9	blog	environmental	sssi	SELECT

To **REVOKE** the Select On privilege, we can run this command:

```
REVOKE SELECT ON environmental.sssi FROM poi;
```

Now if we view the privileges for the POI User, the list will be shorter and they no longer have SELECT rights on the SSSI table.

Data Output | Explain | Messages | Query History

	table_catalog character varying	table_schema character varying	table_name character varying	privilege_type character varying
1	blog	poi	hospitals	INSERT
2	blog	poi	hospitals	SELECT
3	blog	poi	hospitals	UPDATE
4	blog	poi	hospitals	DELETE
5	blog	poi	schools	INSERT
6	blog	poi	schools	SELECT
7	blog	poi	schools	UPDATE
8	blog	poi	schools	DELETE

4 – Listing Tables

As your PostGIS database grows you will need to ensure that you can monitor this growth, by identifying the number of tables within your database, as well as a **count** within individual Schemas. In addition, you may wish to reveal who **owns** those tables.

Running the following command:

```
SELECT *
FROM pg_tables t
```

Will provide a **full list** of all the tables within your database.

Data Output | Explain | Messages | Query History

	schemaname name	tablename name	tableowner name	tablespace name	hasindexes boolean	hasrules boolean	hastriggers boolean	rowsecurity boolean
1	poi	hospitals	postgres	[null]	true	false	false	false
2	roads	mways	postgres	[null]	true	false	false	false
3	pg_catalog	pg_statistic	postgres	[null]	true	false	false	false
4	pg_catalog	pg_type	postgres	[null]	true	false	false	false
5	pg_catalog	pg_authid	postgres	pg_global	true	false	false	false
6	public	spatial_ref_sys	postgres	[null]	true	false	false	false
7	pg_catalog	pg_user_map...	postgres	[null]	true	false	false	false
8	poi	schools	poi	[null]	true	false	false	false

You may wish to edit this to only list the tables in the database, where the **Current User** is the Table Owner.

```
SELECT *
FROM pg_tables t
WHERE t.tableowner = current_user;
```

Here the results only show the tables owned by the **Postgres User.**

Data Output | Explain | Messages | Query History

	schemaname name	tablename name	tableowner name	tablespace name	hasindexes boolean	hasrules boolean	hastriggers boolean	rowsecurity boolean
1	poi	hospitals	postgres	[null]	true	false	false	false
2	roads	mways	postgres	[null]	true	false	false	false
3	pg_catalog	pg_statistic	postgres	[null]	true	false	false	false
4	pg_catalog	pg_type	postgres	[null]	true	false	false	false
5	pg_catalog	pg_authid	postgres	pg_global	true	false	false	false

Many of the tables being listed are default database tables such as - pg_statistic, pg_constraints etc... If you wish to refine the list, you can choose to select the tables which are not within the Information_Schema or the pg_catalog.

```
SELECT table_name FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema','pg_catalog');
```

This time the results show a filtered list of tables.. mainly the spatial tables.

Data Output | Explain | Messages | Query History

	table_name character varying
1	hospitals
2	mways
3	geography_columns
4	geometry_columns
5	spatial_ref_sys
6	raster_columns
7	raster_overviews
8	schools
9	my_tables_list
10	aonb
11	A_ROADS
12	my_tables
13	sssi

If we add the **table_schema** into the SQL:

```
SELECT table_name,table_schema FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema', 'pg_catalog')
```

We then reveal which Schema each table is within.

Data Output | Explain | Messages | Query History

	table_name character varying	table_schema character varying
1	hospitals	poi
2	mways	roads
3	geography_columns	public
4	geometry_columns	public

So we can use the table_schema as part of a **WHERE** statement to only identify the tables that are within the **POI Schema**.

```
SELECT table_name,table_schema FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema', 'pg_catalog')
AND table_schema IN('poi');
```

Now we have a list of the tables within a specific Schema (POI).

Data Output | Explain | Messages | Query History

	table_name character varying	table_schema character varying
1	hospitals	poi
2	schools	poi

Here is a more advanced query that will also return extra fields of information for each Table as stored in the pg_catalog and information_schema table.

```
select t.table_name, t.table_type, c.relname, c.relowner, u.usename
from information_schema.tables t
join pg_catalog.pg_class c on (t.table_name = c.relname)
join pg_catalog.pg_user u on (c.relowner = u.usesysid)
where t.table_schema='poi';
```

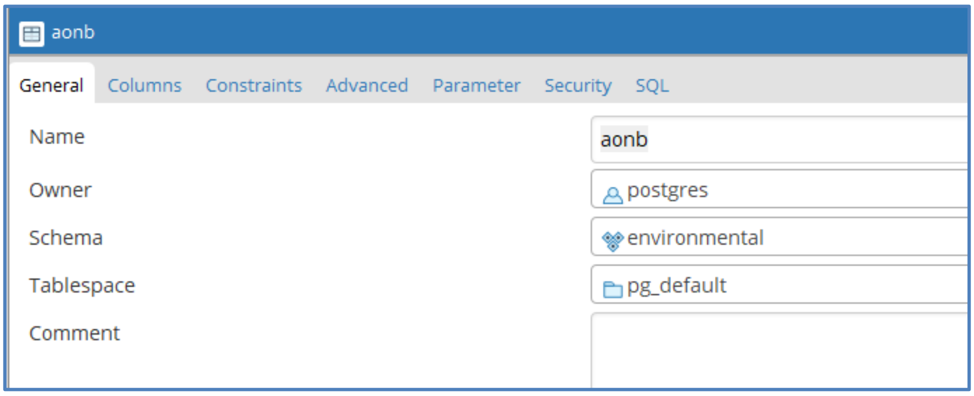
Now we list the 2 Tables but have some extra information, such as the **Table Owner.**

Data Output | Explain | Messages | Query History

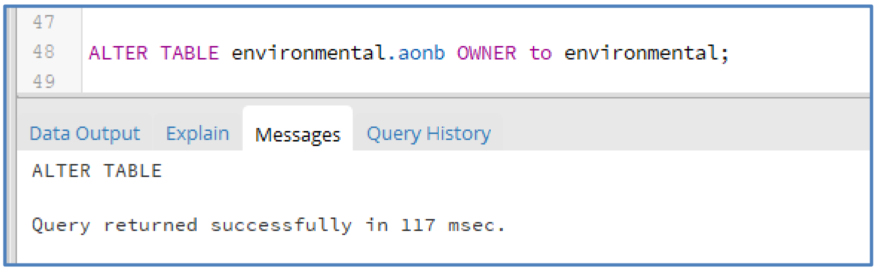
	table_name character varying	table_type character varying	relname name	relowner oid	usename name
1	hospitals	BASE TABLE	hospitals	10	postgres
2	schools	BASE TABLE	schools	10	postgres

5 – Changing Table Owners

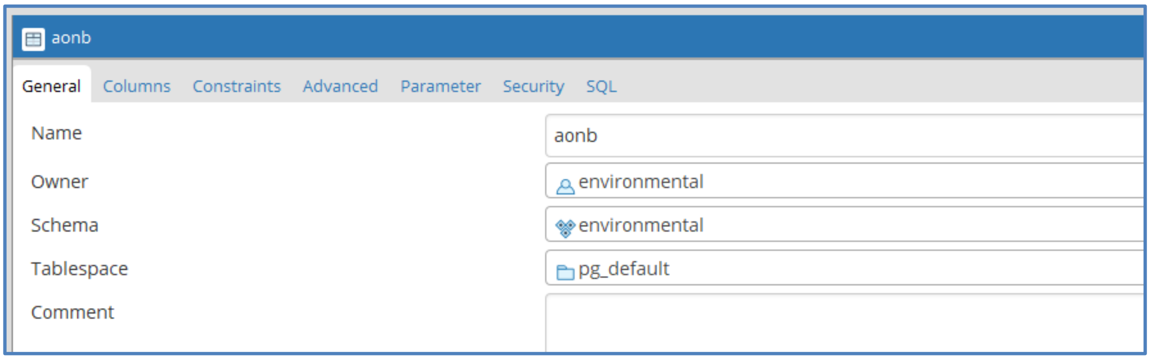
Having learnt how to identify tables within your database and who owns them, you may wish to explore options for changing Table Owners. For example, each of the spatial tables in the POI and Environmental Schemas in our database were imported by the Postgres Super User, which means that the User Postgres is the current Table Owner for all tables.



Using an **Alter Table** statement, we can now change the Owner for one of our tables e.g. (aonb) to now be Owned by the **Environmental User**.



The **Tables Owner** has now changed to be the Environmental User.



If you wish to re-assign the Table back to the Postgres Role, then you can run the **REASSIGN** query. This will not just re-assign tables, but other things such as **Sequences, Triggers, View**s etc..

```
REASSIGN OWNED BY environmental TO postgres;
```

6 – Viewing Database Connections

The next database management process we will explore is how to identify connectivity with your database. This is great for understanding not just Who, but When, How and Why Users are connecting to tables in your database.

To see the full **database activity** you can run this command.

```
SELECT * FROM pg_stat_activity ;
```

A list of **current connections** to your database will be shown in the results panel.

Data Output | Explain | Messages | Query History

	datid oid	datname name	pid integer	usesysid oid	usename name	application_name text	client_addr inet	client_hostname text	client_port integer	backend_start timestamp with time zone
1	12373	postgres	11672	10	postgres	pgAdmin 4 - DB:postgres	::1	[null]	55450	2019-02-18 13:13:06.013195+00
2	445961	blog	7448	10	postgres	pgAdmin 4 - DB:blog	::1	[null]	55620	2019-02-18 13:13:41.747898+00
3	445961	blog	8040	10	postgres	pgAdmin 4 - CONN:8975878	::1	[null]	56272	2019-02-18 13:15:54.718553+00
4	445961	blog	1188	10	postgres	pgAdmin 4 - CONN:7193485	::1	[null]	56552	2019-02-18 13:16:52.231921+00

Let's connect to our PostGIS database by opening a Table via the QGIS interface. If we re-run the above query we can now see that there are some extra connections where the application name is **QGIS.**

```
SELECT * FROM pg_stat_activity ;
```

Data Output | Explain | Messages | Query History

	datid oid	datname name	pid integer	usesysid oid	usename name	application_name text	client_addr inet	client_hostname text	client_port integer	backend_start timestamp with time zone
1	12373	postgres	11672	10	postgres	pgAdmin 4 - DB:postgres	::1	[null]	55450	2019-02-18 13:13:06.013195+00
2	445961	blog	7448	10	postgres	pgAdmin 4 - DB:blog	::1	[null]	55620	2019-02-18 13:13:41.747898+00
3	445961	blog	8040	10	postgres	pgAdmin 4 - CONN:8975878	::1	[null]	56272	2019-02-18 13:15:54.718553+00
4	445961	blog	1188	10	postgres	pgAdmin 4 - CONN:7193485	::1	[null]	56552	2019-02-18 13:16:52.231921+00
5	445961	blog	16596	10	postgres	QGIS	::1	[null]	58514	2019-02-18 15:20:56.707546+00
6	445961	blog	3612	10	postgres	QGIS	::1	[null]	58409	2019-02-18 15:20:09.525891+00

By running the query below we can then filter the activity table to only show the connections via **QGIS.**

```
SELECT usename,application_name,query_start,query FROM pg_stat_activity
WHERE application_name = 'QGIS';
```

The results not only list the Users who have connected, but the **time** they connected and the **function** they performed e.g. the **POI** connected and selected (opened) the **poi** and **hospitals** tables.

```
SELECT usename,application_name,query_start,query FROM pg_stat_activity
WHERE application_name = 'QGIS';
```

Data Output | Explain | Messages | Query History

	usename name	application_name text	query_start timestamp with time zone	query text
1	postgres	QGIS	2019-02-18 15:20:56.791624+00	COMMIT
2	postgres	QGIS	2019-02-18 15:34:25.098169+00	SELECT st_estimatedextent('environmental','sssi','geom')
3	poi	QGIS	2019-02-18 15:34:25.092806+00	SELECT st_extent("geom") FROM "poi"."hospitals"
4	postgres	QGIS	2019-02-18 15:34:25.684976+00	COMMIT

7 – Calculating Database Size

As the database Owner you should also be wary of the database size and act accordingly should the size become too large. Here are a few simple queries to help you identify the size of your database and the individual tables within it.

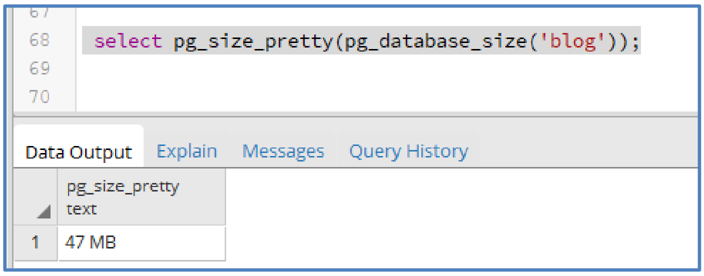
Firstly, the command below will generate a **Total Size** value for the current database that you are querying.

```
SELECT pg_database_size(current_database());
```

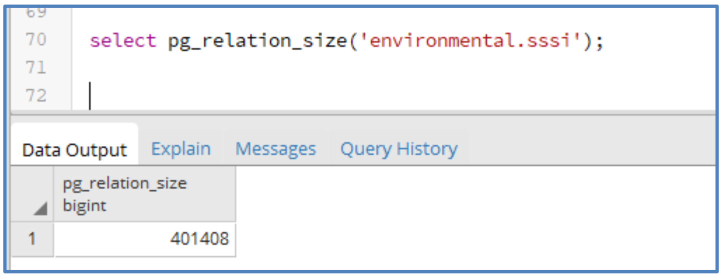
Data Output | Explain | Messages | Query History

	pg_database_size bigint
1	48840876

With the updated statement below, now specifying the **database name** and showing the size in **MB**.



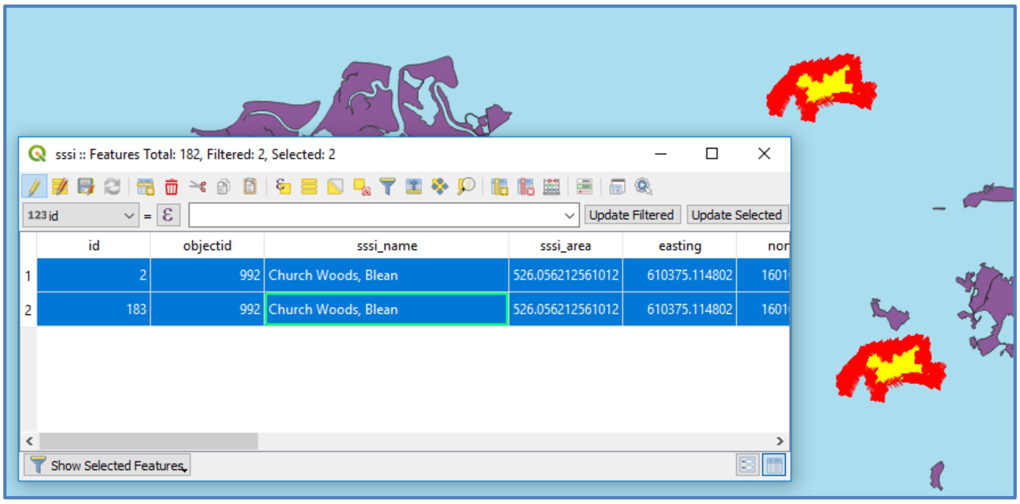
If you then need to understand which tables are the largest, you can identify the size of individual tables using the command below.



8 – Identifying Duplicate Records

The final example we will explore is how to identify duplicate records in your database tables. These can be problematic as they are often records that you don't need and are simply making your database too large.

In this example we have **2 SSSI polygons** with the same **object ID – 992.**



You may not know that there are duplicate records in your tables, so using the following command will allow you to firstly **identify** if there are any duplicates.

```
SELECT * FROM environmental.sssi WHERE ctid NOT IN
(SELECT max(ctid) FROM environmental.sssi GROUP BY objectid)
```

blog on postgres@PostgreSQL 9.5

```
SELECT * FROM environmental.sssi WHERE ctid NOT IN
(SELECT max(ctid) FROM environmental.sssi GROUP BY objectid)
```

Data Output   Explain   Messages   Query History

	id integer	geom geometry	objectid bigint	sssi_name character varying (120)	sssi_area numeric	easting numeric	northing numeric	latitude character varying (12)	longitude character varying (12)	reference character varying (8)
1	2	...	992	Church Woods, Blean	56212561012	0375.114802	0102.510759	51:18:04N	1:01:00E	TR103601

Note – *CTID is an invisible field that acts as a unique value for each record in your table.*

Once you have identified that a duplicate record exists, you may wish to make the edits via QGIS, or you can simply run the command below to **delete** one of the duplicate records.

```
DELETE FROM environmental.sssi WHERE ctid NOT IN
(SELECT max(ctid) FROM environmental.sssi GROUP BY objectid);
```

One of the duplicate records will then be deleted.

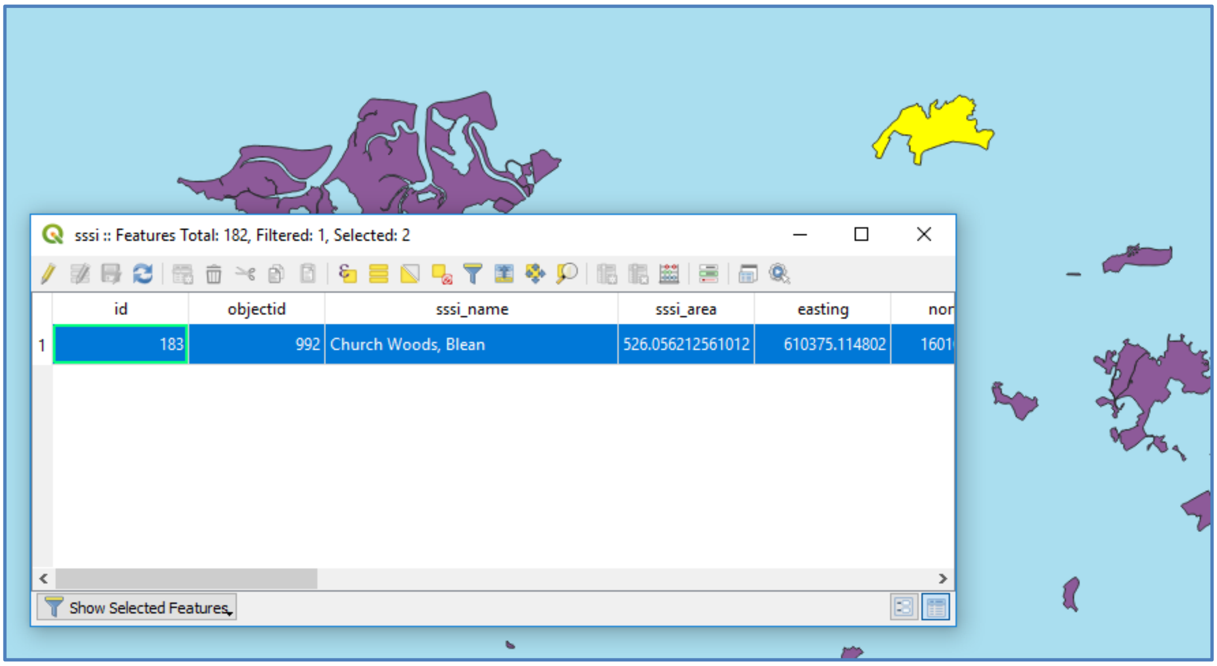
Data Output   Explain   Messages   Query History

```
DELETE 1

Query returned successfully in 164 msec.
```

Note – *This will not allow you to specify which of the 2 records is deleted.*

But if we now check within QGIS we can see that one of the duplicate records has now been removed.



You have now learnt a number of ways to manage the Users, Schemas, Tables and Records within your PostGIS database and hopefully you now feel more in control.

So there is definitely no need to provide all Users with just the default **postgres** loigin details anymore!