F21DF: Database and Information Systems

Introduction to Spatial Analysis using PostgreSQL & QGIS

1.0 Overview

1.1 What is PostgreSQL?

PostgreSQL is an open source object-relational database management system (ORDBMS). It allows you to store data in tables, which can be linked (relational) and can handle many concurrent users at a range of security levels (i.e. some can create and delete data while others can only view the data) with support for custom objects and table inheritance.

1.2 What is PostGIS?

By default PostgreSQL can handle many data types, such as text, integers, floating points, and dates. **PostGIS** is an open source extension which adds the functionality to handle spatial data, so that you can calculate distances, areas, spatial predicates (e.g. within) and so on. PostGIS is best at handling vector data (points, lines, polygons) although it can also handle raster data.

1.3 What is QGIS?

QGIS is a desktop Geographic Information System. This can be used to connect to a variety of spatial dataset including data held in a PostgreSQL database. QGIS includes tools to filter, analyse, and visualize spatial data.

1.4 Overview of this lab

This lab will take you through a number of spatial functions and exercises which you will need to use for the coursework. The lab is quite long and part of your coursework time will be spent completing the lab before undertaking the assignment.

2.0 SQL reminder

A quick reminder of the basic syntax for a SQL SELECT statement to retrieve records from a database server:

```
SELECT {field1,field2,…}
FROM {table1}
JOIN {table2} ON {table2.field} {comparator e.g. =} {table1.field}
WHERE {condition}
GROUP BY {field1, field2,…}
HAVING {condition based on group}
ORDER BY {field1,field2,…} {ASC or DESC} LIMIT {n};
```

- The first { } can be replaced with * which means return all of the fields in the table
- JOIN is optional – used when bring data from many tables together
- WHERE condition is optional – if not used all results will be returned
- GROUP BY is optional – aggregates data & therefore restricts the fields which can be returned
- HAVING is optional with GROUP BY and acts like a where condition on the results after grouping
- ORDER BY is optional and can be ASC or DESC for ascending or Descending order
- LIMIT is optional and can be used to returning a few records not the entire table
- Commands can be written on a single line or spread over many lines
- Field and Table names should be in lowercase for PostgreSQL
- Command should end with a semi-colon ;
- Comments are not parsed, but useful for keeping notes and are marked with double hyphen --

3.0 Accessing PostgreSQL

3.1 MACS Linux Server + Virtual Machine - access and setup

You can use PostgreSQL on the MACS Virtual Machine, your own local install, or via SSH to the MACS Linux servers. If you wish to use the MACS Linux servers you will need to request an account is set up first by emailing the MACS IT helpdesk.

Open a terminal prompt and try accessing your PostgreSQL server.
Note: Change abc123 for your username, and pgsql-server-1 for your PostgreSQL server address – repeat these substitutions throughout the lab.

```
If using the MACS Linux server:
psql -h pgsql-server-1 -U abc123 -d abc123 -W

OR if using the VM:
psql -h localhost -U hw -d hw -W
```

[enter the password for the PostgreSQL database]

You should now see a psql prompt where you can issue database commands and SQL statements: `abc123=>`

If you wish to change your MACS Linux server PostgreSQL password you can issue this statement from within psql, changing pass123 to your own unique password*:*

```
Changing your PostgreSQL password on the MACS Linux Server:

ALTER USER abc123 WITH PASSWORD 'pass123';
```

The first time you use a database on the VM you will need to add the POSTGIS Extension to that database. This has to be done as SUDO to gain the necessary privileges to add the extentsion. It is only needed once per database. The example below is to enable PostGIS on the default **hw** database on the VM.

From a terminal:

```
sudo -su postgres
password : ChMe210+4
psql
\c hw
password : ChMe210+4
CREATE EXTENSION postgis;
\q
exit
```

You can leave this console open as we'll use it later.

3.2 Using pgAdmin to access the database server

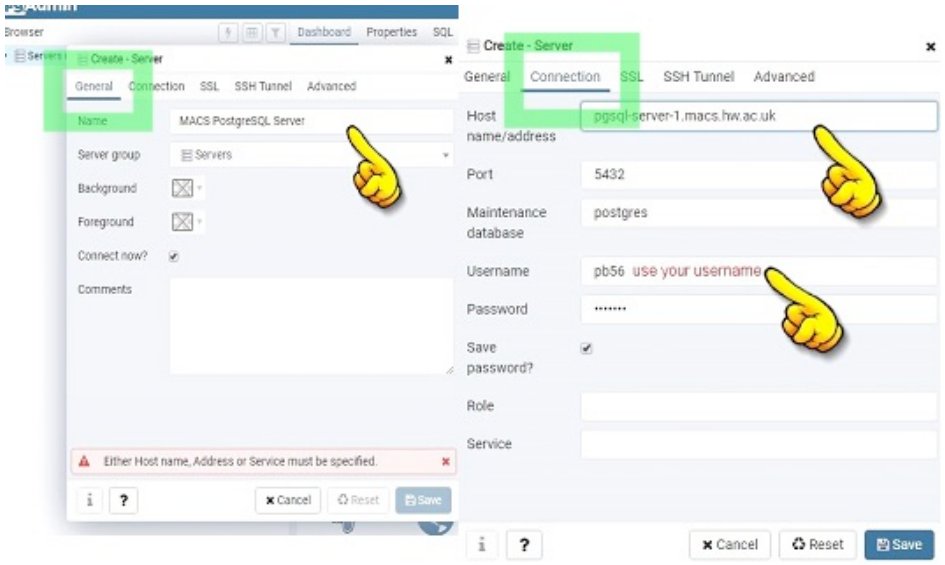
You can use the psql console to issue SQL commands, but a useful alternative is pgAdmin management console, which adds a graphical overview of tables, columns and the database schemas.

This has been installed on the Windows PCs in the GRID Lab (Edinburgh), and you will find it under P in the start menu. There is an icon on the desktop on the VM. The first thing to do is configure the management console to connect to the database server you plan to use. The screenshots below show the connection details for the MACS Linux server – for the VM you would use localhost as the server address.

Run pgAdmin4 - it sets up a local background webserver **then** opens a web page **in** your browser.

Click on the OBJECT menu > **and then** CREATE SERVER

Enter the details as follows - there are 2 tabs to complete (General, Connection)

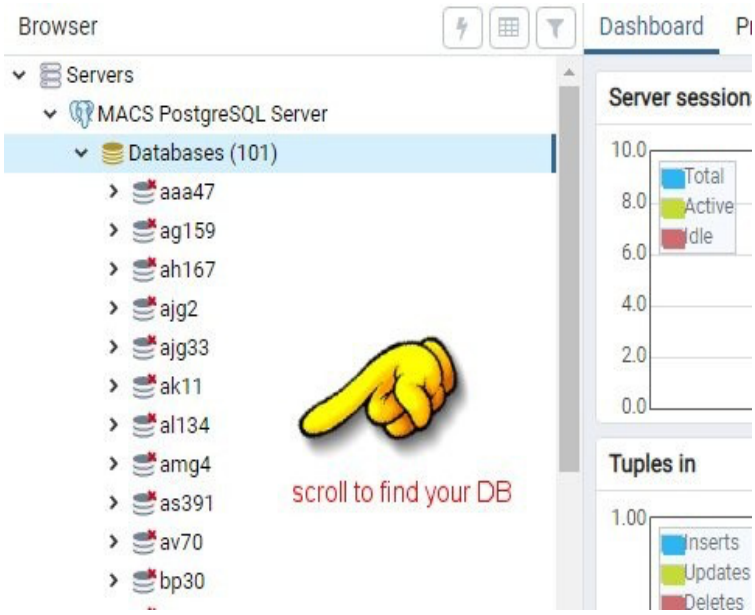


- The MACS Linux server host name is: pgsql-server-1.macs.hw.ac.uk (all lowercase) – substitute your own server IP address if running locally or in the MACS CS VM (e.g. localhost)

- **Set** your username (e.g. ab12) **and** your PostgreSQL **database password and** tick the SAVE **Password option**

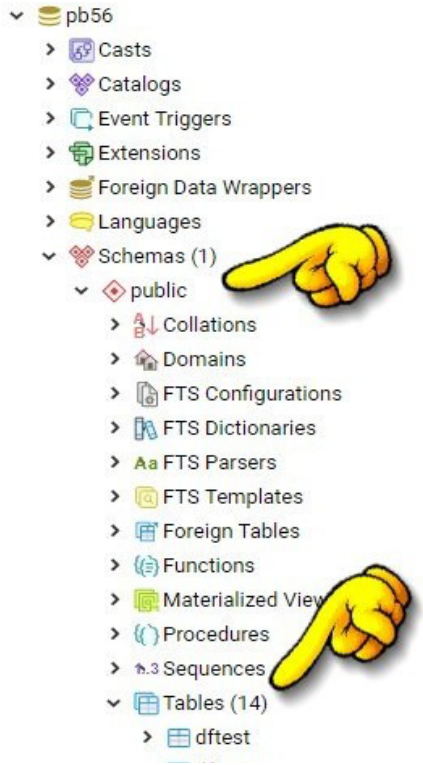
- Click **SAVE** to store these details in pgAdmin

Now click on the entry under SERVERS to expand the view. You should see something like this if you have connected to the MACS Linux Server version – and if running locally there will be fewer (or no) databases listed at this time.



Scroll down until your find your database (e.g. ab12) if on the MACS Server. If running locally for the first time you will need to issue a CREATE DATABASE command first and then enable the PostGIS extension as outlined in section 3.1 above.

For example: `CREATE DATABASE ab12;`

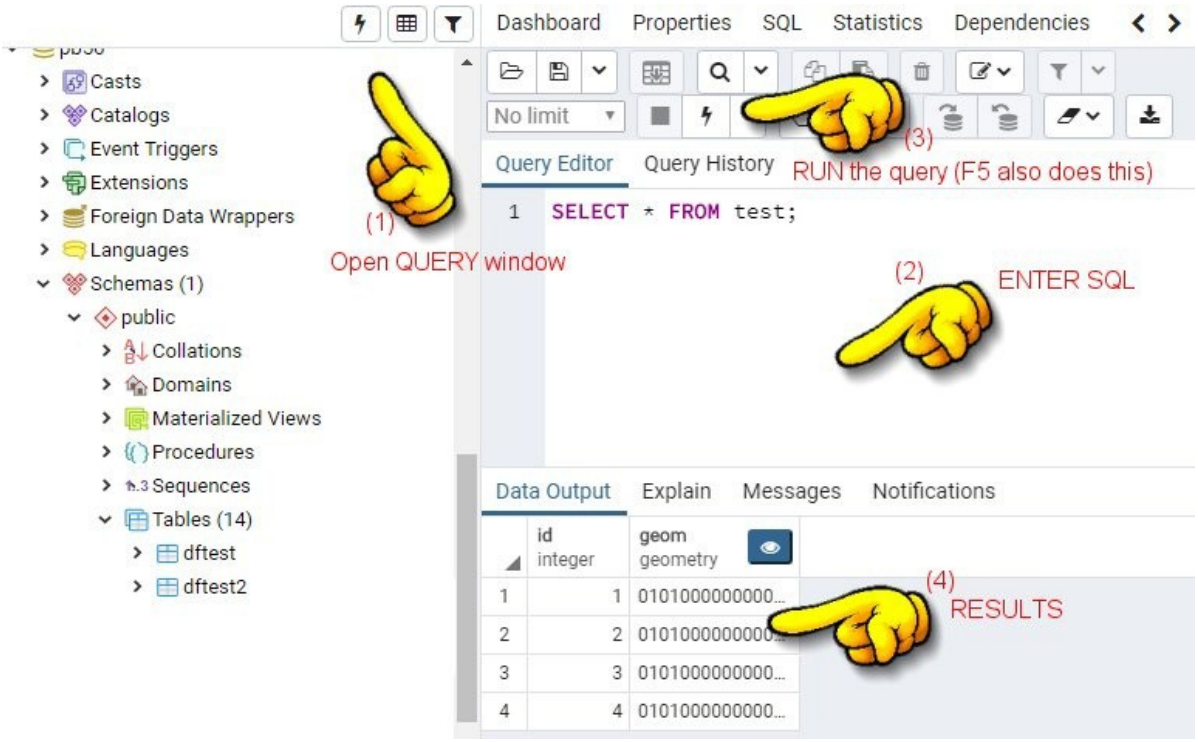


Click on your database item to expand it - it should reveal a large number of items. The only part you need to use for this lab is the **TABLES** section where the datasets are stored.

Click on **Schemas > Public > Tables** to show the tables available. You should now be seeing something like this - but without any tables for now.

To run SQL against this database (ensure your database is highlighted) then use the QUERY EDITOR window (lightning bolt icon). From here you can CREATE tables, SELECT data, INSERT data and so on by writing SQL and would then click on the query lightning bolt (see screenshot) or push F5.

Currently you don't have data to query but in Section 4 you will create a new table and populate it with some records.



4.0 An Introduction to Spatial Analysis

4.1 Overview of the data used in the lab

In this lab you will cover the following:

- creating a point, line, polygon from SQL
- calculating lengths and areas of geometries
- load spatial data into the database
- spatial buffering
- spatial joins (i.e. linking data based on their location)
- calculate distances between features
- visualising spatial data using a desktop GIS (QGIS)

4.2 Create a new table to add points, lines, polygons

To begin we'll create a new empty table in your database, which has 2 fields. Run this SQL QUERY from either psql console, or a pgAdmin4 query window.

```
CREATE TABLE dftest (id serial,geom geometry);
```

If you get an error about not recognising the geometry data type that means you did not install PostGIS for this database.

Notes:

- the serial datatype definition creates an integer field which increments by 1 with each new record added
- geometry is a spatial datatype which can handle points, polylines, polygons

4.3 Insert geometries into a table using SQL

There are a number of ways to create geometries using SQL - to start we'll use the Well-Known Text representation as defined by the Open Geospatial Consortium (OGC). You can read more about the OGC here: http://www.opengeospatial.org/

You can add a single record for a **POINT** at coordinate (x=10, y=0) as follows:

```
INSERT INTO dftest (geom) values (ST_GeomFromText('POINT (10 0)'));
```

To check that this was entered correctly retrieve the information from the table:

```
SELECT * FROM dftest
```

Notice that the id has been automatically given an integer value, and that the geometry column has been converted to binary. To see this as text use ST_ASTEXT(geometry) as follows:

```
SELECT id,ST_ASTEXT(geom) from dftest;
```

You should see this result:

```
1 | POINT(10 0)
```

To add multiple records in a single INSERT statement then do the following:

```
INSERT INTO dftest (geom) VALUES (ST_GeomFromText('POINT (3 3)')),
(ST_GeomFromText('POINT(12 15)')), (ST_GeomFromText('POINT(15 20)'));
```

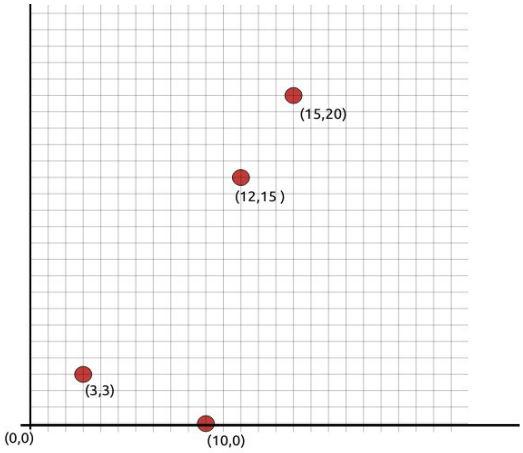
Now try SELECTING the records in the table again to check all looks correct.

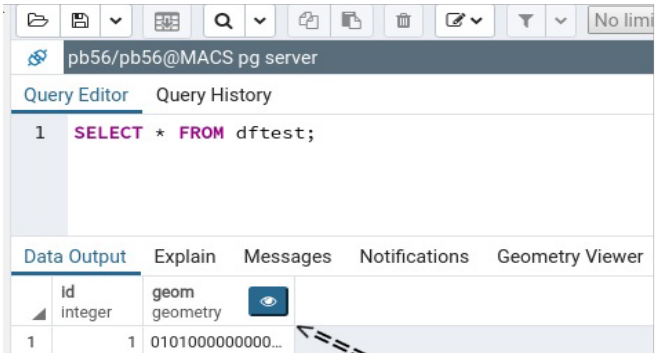
Note:

In pgAdmin4 you can have many query windows open, or have many SQL statements in a single query window - you can run a single statement by highlighting it before pushing **lightning bolt or F5**. Take care if you have many SQL commands in a window, as if you don't select a single statement then all of the queries in that windows will run sequentially. This is by design and can be very useful as it allows you to build models from complex sequences of SQL statements. You can add comments prefixed with – (double hyphen).

In the latest version of **pgAdmin** you can visualize geometry columns by clicking on the **geometry viewer.**

Below is an adapted version of the map output, labelled to show the coordinates of points in the table so far.





As well as **POINTS** we can also add **LINES** (or **POLYLINES**), and **POLYGONS** to our spatial database.

LINES are entered as a list of ordered points separated by a comma (x1 y1, x2 y2). A simple line consists of 2 points (start point and end point), where as a **POLYLINE** is the name given to a line with many segments. They are both defined the same way, just the number of coordinates differentiates them. Here you will insert a polyline with 2 segments (i.e. 3 coordinates).

```
INSERT INTO dftest (geom)
    values (ST_GeomFromText('LINESTRING(10 10, 15 10, 15 20)'));
```

A **POLYGON** is a closed region (an area), therefore the first and last points must be identical. Here we will make a simple polygon, but they can be more complex defined with holes (e.g. a donut shape) - therefore take care with the brackets!
There are more complex versions of geometries including MULTI-geometries, 3D and GEOMCOLLECTIONS. We will not go into that during this lab but you can read more here: https://postgis.net/docs/ST_GeomFromText.html

You can also use ST_MAKEPOINT (), ST_MAKELINE(), and ST_MAKEPOLYGON() as alternative ways to create geometries, which we'll try in a later section.

```
INSERT INTO dftest (geom)
    values (ST_GeomFromText('POLYGON( (0 0,10 0, 10 10, 0 10, 0 0) )'));
```

4.4 Calculate the length, area, centroid of features

Now you have created some geometries in a table let's do a few simple calculations based on those geometries. Here we'll output the geometry type as well as it's length. Note that the spatial functions are prefixed with **ST_** before the function name (e.g. ST_LENGTH).

```
SELECT id,ST_LENGTH(geom),ST_GEOMETRYTYPE(geom) FROM dftest;
```

Notice that only the LINE has a length. Now let's try that again but adding columns for the area and perimeter (these are PostGIS spatial functions that take a geometry input).

You should see the polygon has an area of 100 map units, and a perimeter of 40 map units. The map units are defined by the coordinate system for each record, which in these simple examples are a non-geographic space - so you can think of these coordinates as being on a planar surface. The values returned from the length and area calculations are in the same units as the coordinate system, so if the coordinates are defined as metres then the output distances are also in metres and areas in metres squared. Here we will assume metres.

```
SELECT id,ST_LENGTH(geom),ST_AREA(geom),ST_PERIMETER(geom),
  ST_GEOMETRYTYPE(geom)
  FROM dftest;
```

Each feature also has a centroid, this is the central point of the feature. Here we'll nest the ST_CENTROID() function within the ST_ASTEXT() function.

```
SELECT ST_ASTEXT(geom),ST_ASTEXT(ST_CENTROID(geom)) FROM dftest;
```

4.5 Calculate the distance between features

Let's now find the Euclidean (straight line) distance between 2 features. We'll use ST_MAKEPOINT(x,y) as an alternative way to create a **POINT**. You could also use this or the **WKT** approach we used in Section 4.3 - whichever way you prefer.

ST_DISTANCE (geom1,geom2) returns the minimum distance between those features - here we populate geom1 with values from the table, and specify a static point at (3,3) for geom2.

```
SELECT id,ST_GEOMETRYTYPE(geom),ST_DISTANCE(geom,ST_MAKEPOINT(3,3))
  FROM dftest;
```

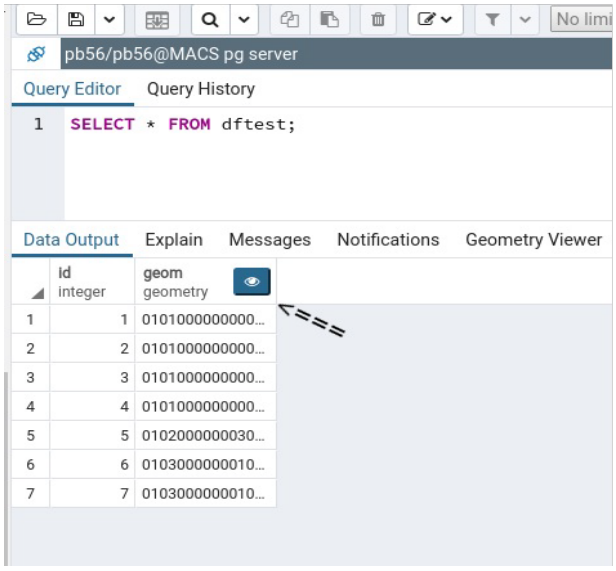
The output should look like this, showing the distance in map units (e.g. metres) from a point at coordinate (3,3) to each of the records in the table.

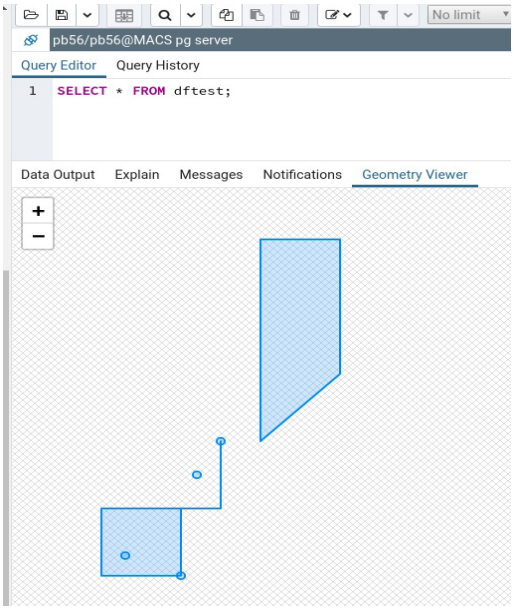
```
1 | ST_Point           |     7.61577310586391
2 | ST_Point           |     0
3 | ST_Point           |     15
4 | ST_Point           |     20.8086520466848
5 | ST_LineString      |     9.89949493661167
6 | ST_Polygon         |     0
```

Notice that it returns a distance of 0 for the id=2 which was a point also at (3,3). It also returns a distance of 0 for id=6 which is the polygon. This is because the point (3,3) is located within (i.e. inside) of the region defined by the polygon. Also notice that it returns the shortest distance from the point(3,3) to the line.

TASK: Try adding a new polygon with coordinates (20 20, 30 30, 30 50, 20 50, 20 20) to the **dftest** table , and then calculate the distance to a point at (5,5). As this new polygon (id=7) does not contain the point (5,5) you should get a distance other than 0 (e.g. 21.2).

Try the pgAdmin geometry visualisation tool again to see if your database table with 4 points, 1 line, and 2 polygons look like this:





4.6 Using a Spatial Join to count the number of Points in a Polygon

You will have used database JOINS before to link records, perhaps between an EMPLOYEE table and a PROJECT table based on primary and foreign keys. We can do something similar using the spatial column to join tables based on geographic locations. For example linking all rivers (polylines) within a national park (polygon). This is known as a **SPATIAL JOIN** and a powerful feature as it allows us to join otherwise unrelated data together.

Let's try this with a simple example - first of all let's create another table and insert a single polygon. We'll then create a SPATIAL JOIN between the tables to count how many features from the **dftest** table are within the new polygon.

```
CREATE TABLE dftest2 (id serial,geom geometry);
```

Now insert a polygon:

```
INSERT INTO dftest2 (geom)
   VALUES  (ST_GeomFromText('POLYGON((0 0,13 0, 15 20, 3 10, 0 0))'));
```

To count the number of features from **dftest within** this polygon we will use a **SPATIAL JOIN**:

```
SELECT ST_ASTEXT(dftest.geom) as df1geom, ST_ASTEXT (dftest2.geom) as df2geom
FROM dftest
JOIN dftest2 ON ST_WITHIN(dftest.geom,dftest2.geom);
```

Notice that this time we've added column name *aliases* (df1geom, df2geom) to make the output more readable.

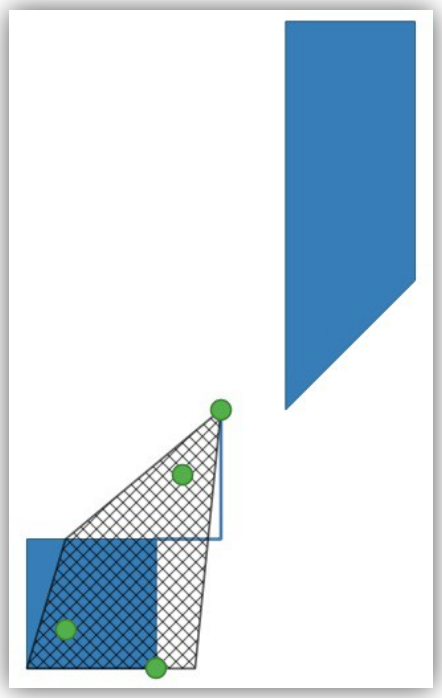
You should find that two points from **dftest** are contained within the polygon defined in **dftest2**.

```
df1geom       |  df2geom
--------------+-----------------------------------
POINT(3 3)    |  POLYGON((0 0,13 0,15 20,3 10,0 0))`
POINT(12 15)  |  POLYGON((0 0,13 0,15 20,3 10,0 0))
```

Aside:
As we only added a single record to dftest2 we could have done this on the fly, for example the following SQL shows us which features are within the polygon defined in the SQL statement.

```
SELECT ST_ASTEXT(dftest.geom) as df1geom,
ST_WITHIN(dftest.geom,(ST_GeomFromText('POLYGON( (0 0,13 0,15 20,3 10, 0 0))')))
FROM dftest;
```

```
df1geom                                    |  st_within
-------------------------------------------+----------
POINT(10 0)                                | f

POINT(3 3)                                 | t

POINT(12 15)                               | t

POINT(15 20)                               | f

LINESTRING(10 10,15 10,15 20)              | f

POLYGON((0 0,10 0,10 10,0 10,0 0))         | f

POLYGON((20 20,30 30,30 50,20 50,20 20)) | f

(7 rows)
```



The map confirms our findings that the polygon in **dftest2** (cross-hatched) contains just 2 points completely. The other features are on the boundary, not completely contained, or outside of the polygon. We'll look at other types of spatial predicate (e.g. WITHIN, TOUCHES, INTERSECTS) in the next section.

Here using the geometry has allowed you to link tables together in a way that otherwise would not have been possible. Often geography is the only way to link information from different sources (e.g. population density, crime rates, house prices).

4.7 Using a Self Join

Now we'll try this with the **dftest** table to find out how many points are within the first polygon (id=6). As we are trying to find out how many geometries are within a polygon in the same table (**dftest**) we need to use a **SELF JOIN**. This is where we reference a table twice in the SQL statement, each time with a different alias. We can then treat the aliases as if they were different tables, and therefore find the spatial relationships between features. This is used quite often – for example in a table which holds vehicle crash locations a self join is necessary to find out how many vehicle crashes occurred nearby (e.g. within 100m).

Here we'll load **dftest** and give it the alias of '**a**', then load the table again and alias it as '**b**' - these could be any alias name you wish as long as they are unique.

```
SELECT a.id, ST_ASTEXT(a.geom) FROM dftest a
JOIN dftest b ON ST_WITHIN(a.geom,b.geom)
WHERE a.id != b.id AND b.id = 6;
```

You should have a single result as follows:

```
id | ST_ASTEXT

----+------------

2 | POINT(3 3)
```

This means that only the record with id=2 (point (3,3)) is within the polygon (id=6).

Notice from the figure above (Section 4.6) that there is also a point on the boundary, and a line which goes along the boundary. These are not considered to be WITHIN the polygon.

Try running the SQL above again but this time substitute **ST_WITHIN** with **ST_TOUCHES** instead. Notice a different point ID (id=1) is returned with a line (id=5) now - these are touching the boundary, but not within the polygon. Try once more but this time use **ST_INTERSECTS** rather than **ST_TOUCHES.** This means that any interaction (touches, within) will be returned in the results. You should see the following result:

```
1 | POINT(10 0)
2 | POINT(3 3)
5 | LINESTRING(10 10,15 10,15 20)
```

Aside:
You can determine the spatial relationship between features according to the Dimensionally Extended 9 Intersection Model (DE-9IM) using ST_RELATE(geom1,geom2) https://postgis.net/docs/ST_Relate.html

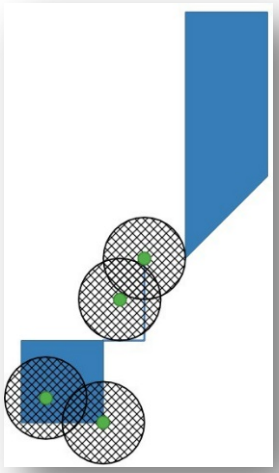
4.8 Spatial Buffer

We can nest functions, for example modifying a geometry during a join by adding a spatial buffer around it. Here we'll do that and add a buffer of **3m** around the line (**id=5**) before doing the spatial join. Here we'll use a **SUBQUERY** to select the line from the **dftest** table based on its **id** (5) and then **buffer** it, and then **count** the features that intersect that buffer.

```
SELECT dftest.id,ST_ASTEXT (dftest.geom) `
FROM  dftest
JOIN (SELECT * FROM dftest WHERE id=5) line ON
ST_INTERSECTS(dftest.geom,ST_BUFFER(line.geom,3));
```

We could also use this approach to make a new table with modified geometries. For example we'll make **dftest3** based on buffering all the points in the table **dftest** by 5 metres, using the GEOMETRYTYPE to filter the input table for just the points.

```
CREATE TABLE dftest3 AS
(SELECT id, ST_BUFFER(geom,5) as geom FROM dftest
WHERE ST_GEOMETRYTYPE(geom) = 'ST_Point' );
```



Now you have a **dftest3** table which has 4 polygons - as you've turned the points into polygons using ST_BUFFER(). In the map on the right, the crosshatched circles are from **dftest3**, and the blue features from **dftest**.

4.9 Distance Matrix using a CROSS JOIN

Earlier on you found the distance from a point (3,3) to all other features - you could do something similar to calculate the distance from each feature to every other feature using a CROSS JOIN. This requires a self join between the table but here you specify the join type to be a CROSS JOIN.

```
SELECT a.id as origin,b.id as dest,ST_DISTANCE(a.geom,b.geom)as dst
FROM dftest a CROSS JOIN dftest b;
```

The results look like this - notice that it is comparing each record against all other features (check the id columns).

```
   origin   |   dest     |     dst
     1      |      1     |     0
     1      |      2     |     7.61577310586391
     1      |      3     |     15.1327459504216
     1      |      4     |     20.6155281280883
     1      |      5     |     10
     1      |      6     |     0
     1      |      7     |     22.3606797749979
     2      |      1     |     7.61577310586391
     2      |      2     |     0
     2      |      3     |     15
 … etc
```

TASK: Try modifying the previous query to exclude all results where origin and destination are not identical.

5.0 Working with GIS Data in PostgreSQL + PostGIS

5.1 Using the command line to load data

Now that you have learnt the basics of handling spatial data within a database, we will load some larger datasets for a section of Edinburgh (Scotland). The features are defined in the British National Grid (BNG) coordinate system, which uses metres. You don't need to be concerned about the coordinate system other than to know that for this lab (and coursework) all the data will be loaded using the same coordinate system (BNG).

Aside:
There are many hundreds of coordinate systems, with projections defined from local regions. For example:

- British National Grid (EPSG: 27700) https://epsg.io/27700
- Dubai Local TM (EPSG:3997) https://epsg.io/3997

Use the Linux terminal window you opened earlier and download some GIS data using `wget`.
(if you shut the terminal then in Edinburgh GRID Lab open Virtual Box then run the terminal prompt)

```
wget www.macs.hw.ac.uk/~pb56/df_flickredin.sql
```

Note:
If you don't have access to `wget` you can download the data required for the lab and coursework from a web browser from: www.macs.hw.ac.uk/~pb56/f21df.html

Now load the data from the command line into the PostgreSQL server using psql as follows:
(substitute abc123 for your username + change the server host address as appropriate)

```
via ssh to MACS server:
psql -h pgsql-server-1 -U abc123 -d abc123 -W -q -f df_flickredin.sql

OR on the VM:
psql -h localhost -U hw -d hw -W -q -f df_flickredin.sql

where:
 -h is the host server; -U is the user account to use; -d is the database;
 -W prompt for password; -f file to load; -q work quietly / avoid verbose msgs
```

enter your Postgresql database password when prompted - it may take a few minutes to load the data

If you are using the VM the password will be: **ChMe210+4**

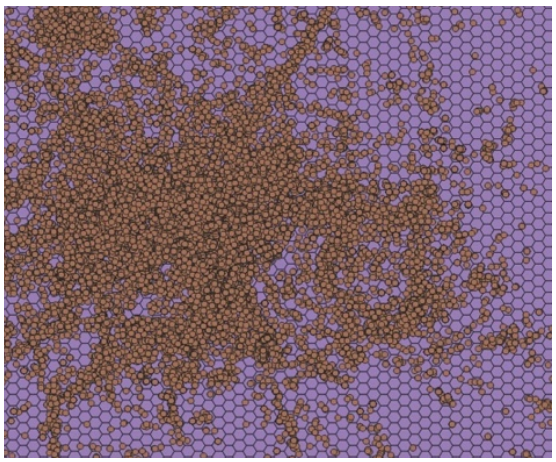
Repeat this procedure for **df_grid100m.sql** which can also be downloaded from the same URL.

You should have just loaded two tables:

- one contains the photo locations (as points) around Edinburgh based on what has been uploaded to Flickr. Flickr is a photo sharing website - if you've not head of it take a look here: https://www.flickr.com/

-the other contains a hexagon (polygon) grid, with cells of height 100m.

Together they look something like this ===>



We'll get to how you can visualise these layers later but for now just check they loaded into your database and check the number of records in each table.

In pgAdmin refresh (right click menu – refresh option) the list of tables to see the new table names added. Then run the COUNT() function for reach table to find out how many records were loaded into each.

Then this SQL will return the number of records for the **flickr_edin** table would be as follows:

```
SELECT COUNT(*) from flickr_edin;
```

You should have 80237 records

TASK

Repeat this for the other table that was just loaded (the polygon grid) - how many records are there? _____

5.2 Spatial Indexes

We want to make a new map which shows the number of photos taken in each of the hexagons - this involves a point-in-polygon spatial operation as you did before but with a larger dataset. To speed up this operation we'll add a spatial index to each table so that the join is faster. This index enables the database to subset the data based on spatial location, thereby improving performance for larger datasets.

For geometries we use a GIST index as follows:

```
CREATE INDEX sidx_flickr_edin_geom ON flickr_edin USING gist (geom);

CREATE INDEX sidx_grid100m_geom ON grid100m USING gist (geom);
```

An index is an extra data structure stored on disk that helps the database to search records based on the specified index field - in this case the geometries stored in each table.

5.3 CREATE a NEW TABLE from a SQL Command

We will now create a new table which shows the number of flickr photos in each of the hexagon polygons by using a spatial join based on points inside of polygons ST_WITHIN(geomA,geomB). In this example the output table is going to be called **flickrgrid** and will consist of the point count and the grid geometry.

```
CREATE TABLE flickrgrid AS SELECT COUNT(*) as count,b.geom FROM flickr_edin a
JOIN grid100m b ON ST_WITHIN(a.geom,b.geom)
GROUP BY b.geom;
```

(This may take a little while to run)

TASK Check what happened by summing the total count column in the new table and checking it matches the total number of **flickr** photos. Does it match - if not why do you think this might be? The answer to this should be clearer when you get to visualising the data in QGIS in step 6 of the lab.

How about the number of grid cells - do they match?

TASK: Try to fix the query so that it returns 0 counts for cells without any flickr photos.

5.4 Using SQL to handle Dates and Times

As well as working with spatial data you have a time series of photo locations (i.e. spatio-temporal data). PostgreSQL has a range of tools for processing dates and timestamps.

Check here for details: https://www.postgresql.org/docs/11/functions-datetime.html

Let's take a look at the Flickr data and extract the **Day of the Week** (e.g. Mon, Tue) from the **date_taken** timestamp. We can use this **DOW** to count the number of photos taken each day to see which are the most popular days, using GROUP BY.

```
SELECT EXTRACT('dow' FROM date_taken) as dow,COUNT(*)
FROM flickr_edin
GROUP BY dow
ORDER BY dow;
```

Should give you this:

```
dow | count
0   | 14630
1   | 8318
2   | 8769
3   | 8199
4   | 10152
5   | 10919
6   | 19250
```

TASK: Try doing the same thing but group by the **week** of the year (e.g. week 1 to week 52).

5.5 Filtering based on the content of a field

So far we've been using all of the data in a table, but often you will need to filter the table using an attribute. This is done in SQL with the WHERE condition. The example below uses the ILIKE match function which ignores case and permits the use of wildcards(%). A wildcard means that any other combination of characters can be substituted (e.g. %h% would match '*h*ere' and 't*h*ere' while 'h%' would only match '*h*ere').

First take a look at the usertags field for a record which has some usertags:

```
SELECT usertags
FROM flickr_edin
WHERE LENGTH(usertags)>1
LIMIT 1;
```

Let's count how many flickr photo locations mention the Edinburgh Fringe.

```
SELECT COUNT(*)
 FROM flickr_edin
 WHERE usertags ILIKE '%edinburgh fringe%';
```

6.0 Viewing the map

For the final part of this lab we'll use a desktop GIS (geographic information system) to view the data held in the PostgreSQL database. The application we'll use is QGIS – which is open source and available for Windows, Linux, and Mac operating systems.

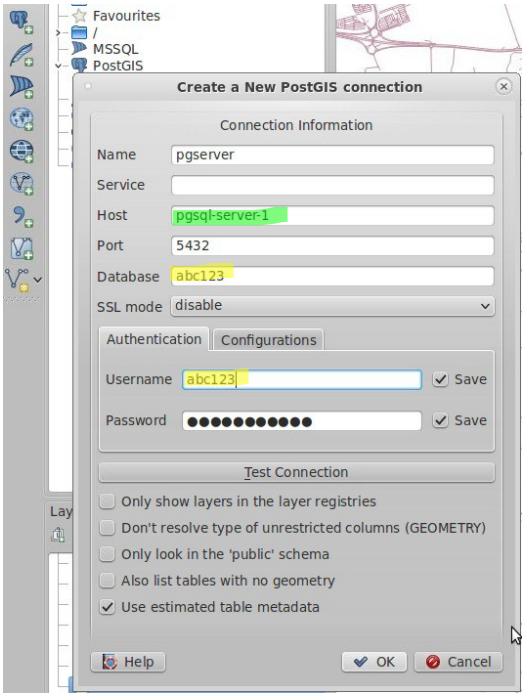
In Edinburgh use the **GRID Windows Lab** as QGIS ver 3.x is installed on the desktop- **this is the preferred option for Edinburgh students**. You should find QGIS from the START menu.

Note: you can also access an older version of QGIS (ver 2.x) on the LINUX server using the VM (GRID lab) or X2GO - it's found under the Applications Menu > Education > QGIS Desktop. It's recommended you use the

same version throughout to avoid versioning issues with moving map project files about.

6.1 Setting up QGIS to connect to the PostgreSQL server

Once QGIS is loaded click on the ELEPHANT icon and NEW CONNECTION. (Substitute for your PostgreSQL server IP address if not using MACS server – eg localhost on the MACS VM)

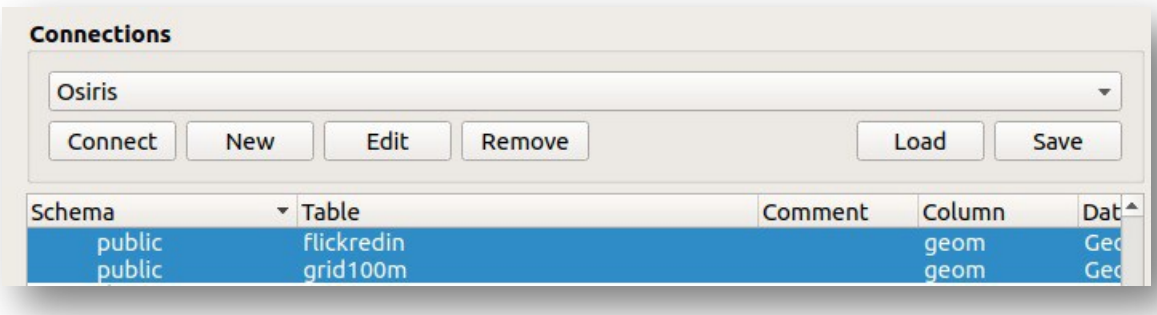


Set up the PostgreSQL connection as shown
(replacing abc123 with your username and pgsql-server-1.macs.hw.ac.uk with your PostgreSQL host details)

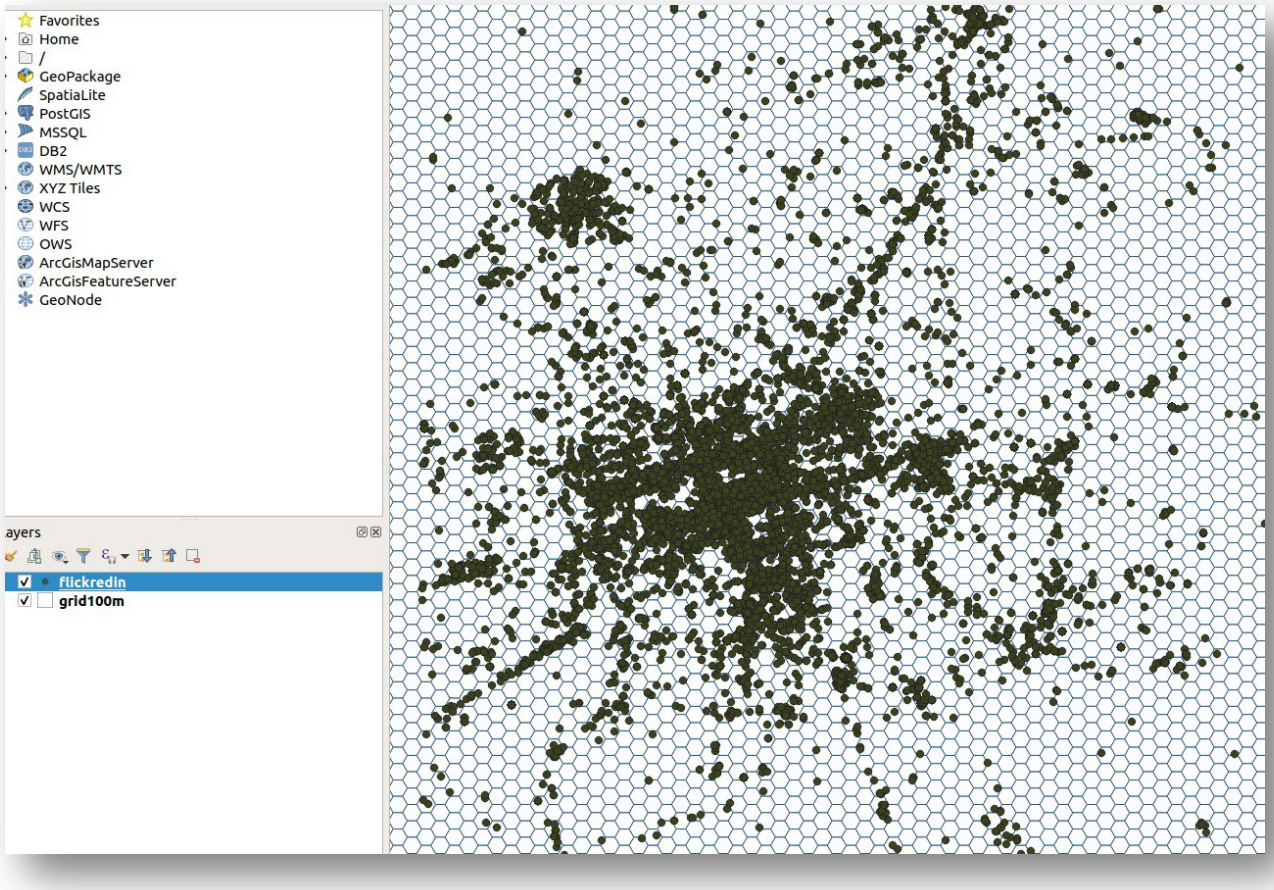
```
Host: pgsql-server-1.macs.hw.ac.uk
Port: 5432

Database: abc123

Username: abc123

Password: {your postgresql database password)
```

If the database has large tables it's a good idea to tick the use estimated table metadata (not needed here) – as this speeds up the process of scanning for geometry columns held in tables on the server.

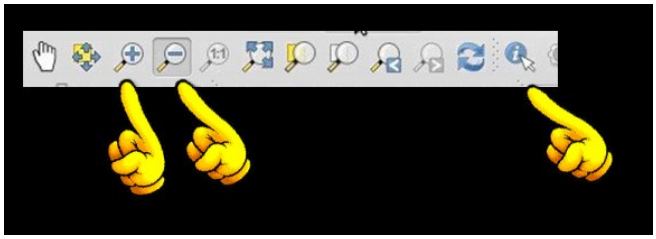
Now connect to the server and highlight the layers you wish to view (e.g. **flickr_edin, grid100m**) and click **Add**



You can now re-order the layers -by dragging them in the Table of Contents. Usually this would be polygons at the base, then polylines, then points on top. You should be able to see something like this.

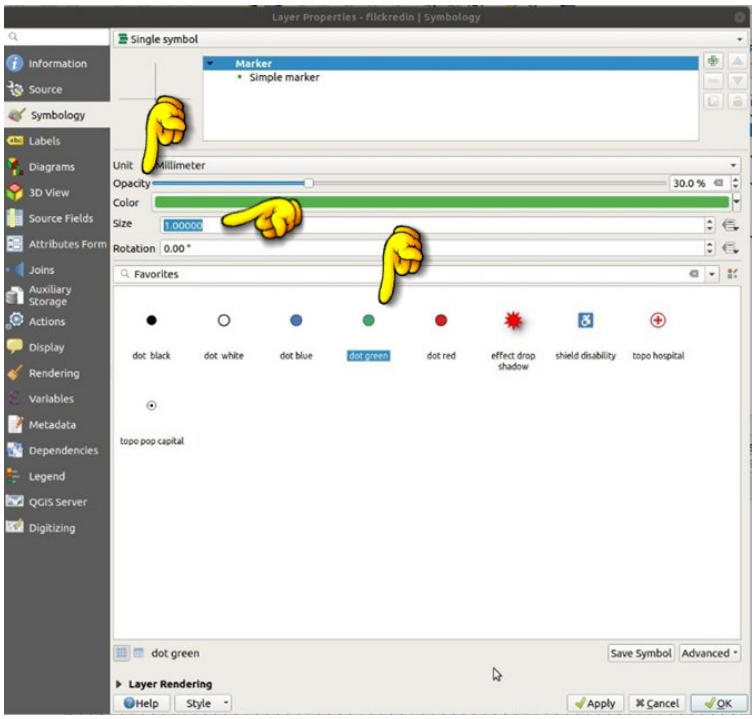


Use the **PAN** tool (**hand**) and **ZOOM** tools (**magnifying glass**) to move around the map. Also the **IDENTIFY** (i) tool is useful for clicking on a feature to see the attributes for that item – based on which layer is highlighted in the Table of Contents at the time you click on the feature.

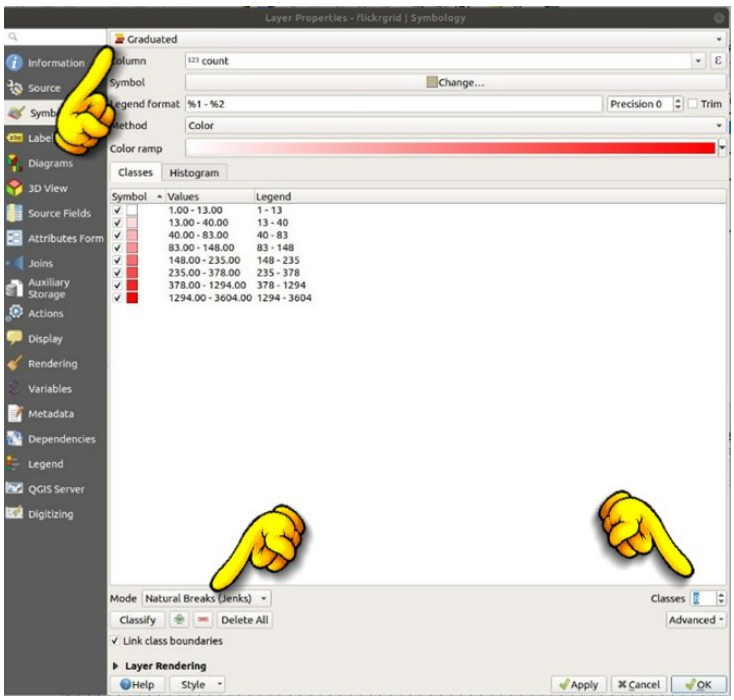


6.2 Symbology

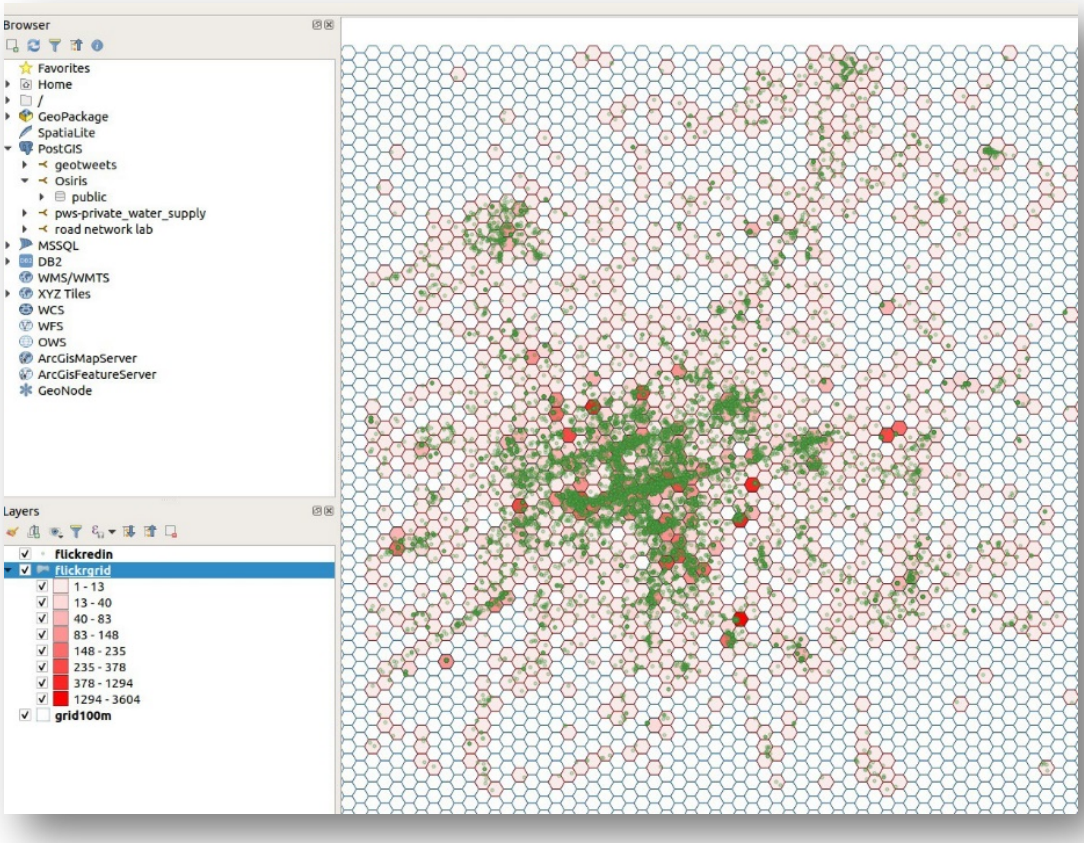
Double click (or right click > properties) on the **flickedin** layer name in the table of contents, to open the properties. Choose the **SYMBOLOGY** tab and pick the '**dot green'** item, set the size to be around **1.0**, and the **Opacity** to around **30%** - then click **OK**.



Now highlight the **flickrgrid** polygon layer - open the **symbology** settings as before. This time however instead of using a single symbology for the entire layer we'll specify **Graduated** (at the top) and then specify the **count** column. Next change the **Mode** to **Natural Breaks (Jenks)** and the number of classes to **8**. Then click **OK**.



You should now see a map something like this.



6.3 Exporting a Map as an Image

To export a map you should **include the basic cartographic elements as per the lecture** (e.g. legend, title). This is done from a PRINT LAYOUT screen.

From the PROJECT MENU > NEW PRINT (or CTRL+P)

Use ADD ITEM menu to place the map, legend, north arrow, scale bar, and title on the page.

You can now export this map as an image from: **Layout Menu** > **Export as Image.**
Have a play around with QGIS and the map design.

TIPS:

- To pan a map on the PRINT LAYOUT push the C key while using the mouse pointer to slide the map around.
- You can edit the layer names in the Table of Content (TOC) and they will update in the legend on the PRINT LAYOUT
- If you wish you can disconnect the LEGEND from the TOC and then remove items (layers) in the legend.

Some things to try:

Check LinkedIn learning (videos) and YouTube for more information about what you can do with QGIS.

YouTube videos:

- Add labels to the map based on attributes:
  https://youtu.be/Z64bzN8U_eE and https://www.youtube.com/watch?v=5-dAJ0i14I8
- Apply a filter to a layer: https://youtu.be/dGghke1QFxs?t=49

LinkedIn Learning course on QGIS:

- **Points: https://www.linkedin.com/learning/learning-qgis-2/point-layers?u=2374954**
- Labelling: https://www.linkedin.com/learning/learning-qgis-2/label-vector-data-part-1?u=2374954

6.4 Save your Map Project

You should save your map project to your personal drive. The map project contains links to the various datasets (but not the actual data), the Table of Contents order, symbology, and any layout details (e.g. labels, scale bars). Do this from **PROJECT** drop down menu > **Save**.

Well done you have now learned about PostgreSQL and PostGIS,
as well as how to display spatial data using QGIS.

------ END OF THE LAB ------