Introduction to the Dimensionally Extended 9 Intersection Model (DE-9IM) in PostgreSQL/PostGIS

Tutorial

Germán Carrillo
gcarrillo@uni-muenster.de
geotux_tuxman@linuxmail.org

Objectives

Following this tutorial you will be able to:

- Create spatial databases
- Create spatial tables (alfanumeric tables + geometry)
- Check topological relations on geometries
- Run spatial operators on geometries (spatial analysis)
- Create spatial tables from spatial operators

Requirements

- Data Base Manager System (DBMS) PostgreSQL and its spatial extension PostGIS installed.
- PostgreSQL client pgAdmin installed.
- Edit permissions on PostgreSQL (to create and modify database objects).
- Optionally, a pgAdmin's plugin installed to display geometries. The plugin is called "PostGIS viewer".
- SQL basic knowledge.

Let's start!

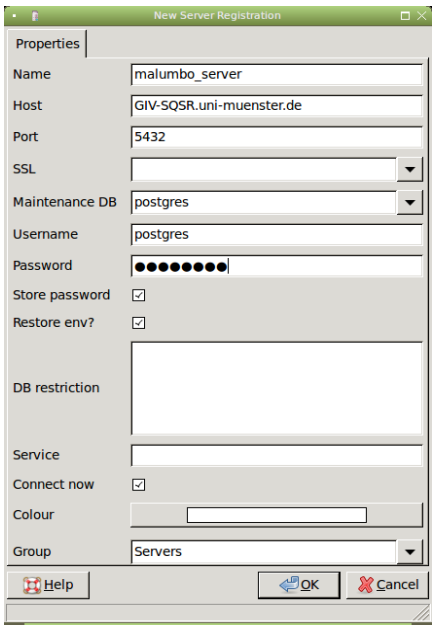
1. Connecting pgAdmin to the server

Open *pgAdmin3*.

In linux you can press Alt+F2 and write pgadmin3.
In Windows you can find it in the PostgreSQL folder.

PgAdmin can act as client for a number of PostgreSQL servers at the same time. At least you need one server, it may be your own server (if PostgreSQL is installed in your computer) or the server previously configured for the tutorial (ask for the connection details).

You may need to add a server:



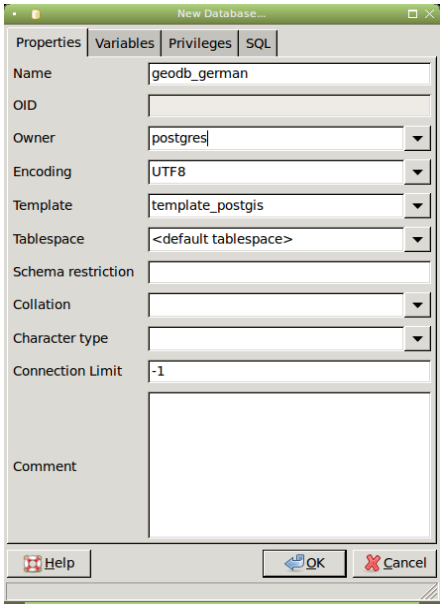
The DBMS has a superuser, which is a role with all the permissions over the server (so, be careful with it!). In the configured server, the superuser is *postgres* and its password is *postgres*.

Once the server is added, you can connect to it by double-clicking over its name in the tree on the left.

You will get the databases, tablespaces and roles (users) of that server.

2. Creating your spatial database

Right click on the item *Databases(2)* and select *New Database*... You will get this window:



Fill the fields according to the figure. Don't introduce spaces or non-ASCII characters.

Click on Ok.

If everything is ok, you have your own database in the server. That database *(geodb_german)* is also a spatial database since you used a PostGIS template database (*template_postgis*) to create it.

3. Introducing PostgreSQL databases

Databases are core objects in DBMS such as PostgreSQL. They allow us to store, query and manage vast amounts of related data. For every object in your data model you may create a table in PostgreSQL.

To organize the data, PostgreSQL uses *Schemas*, which are data containers such as the directories in your computer's file system. The diference is that Schemas can not be nested, i.e. you can not create Schemas inside Schemas.

PostgreSQL uses a default Schema called *public*, where all the database objects, such as tables, will be created (you can change that by creating your own schemas).

Thus, we will be working on the *public* schema of your database.

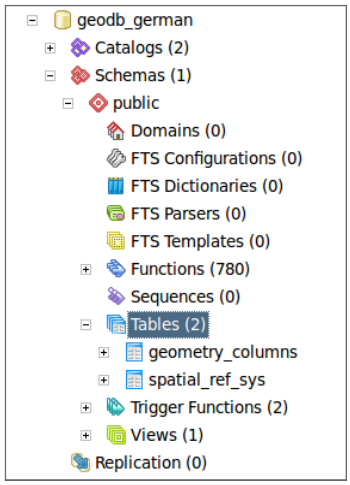
4. Introducing PostGIS

PostgreSQL is not intended to work with spatial objects. It actually can work with basic geometries but it does not understand things like Spatial Reference Systems (SRS).

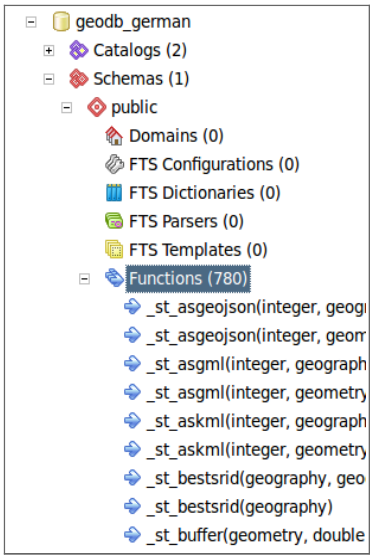
PostGIS is a spatial extension to PostgreSQL allowing it to deal with a wide variety of geometry types (more than Points, Lines and Polygons), understand SRS and provide functions to query, combine and modify geometries.

The way PostGIS stores geometries and SRS, and names functions is given by the *Simple Feature Access for SQL* technical specification from the Open Geospatial Consortium (OGC).

Go to the public schema and explore its tables. Those are tables created by PostGIS to store geometries and SRS.



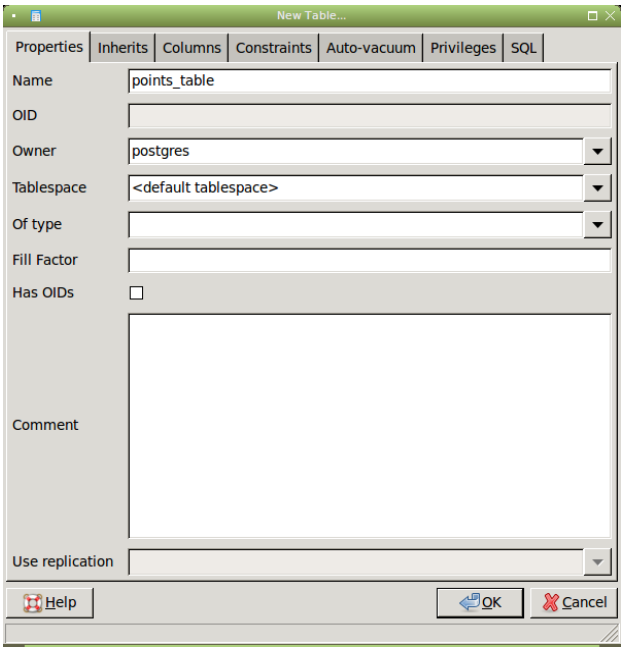
Go to the functions section in the *public* schema. Those are PostGIS functions to create, drop, edit, query, process, and relate geometries. Are you familiar with some of those functions?



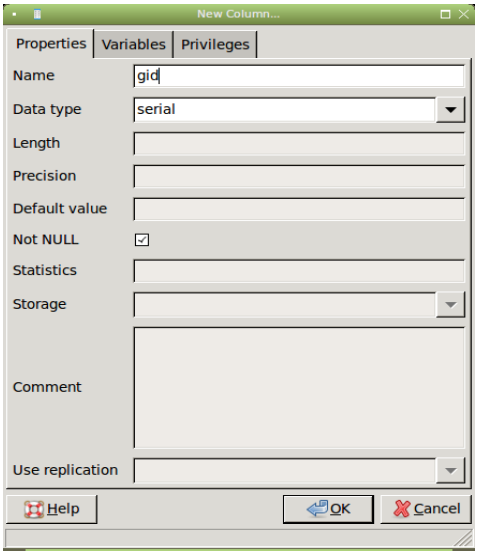
5. Creating tables

We will use pgAdmin to create a table by means of a Graphical User Interface (you could also create it using purely SQL instructions).

Right click on the *Tables (2)* tree item and select *Create Table...* You will get this window (only click on the *OK* button when it is indicated):

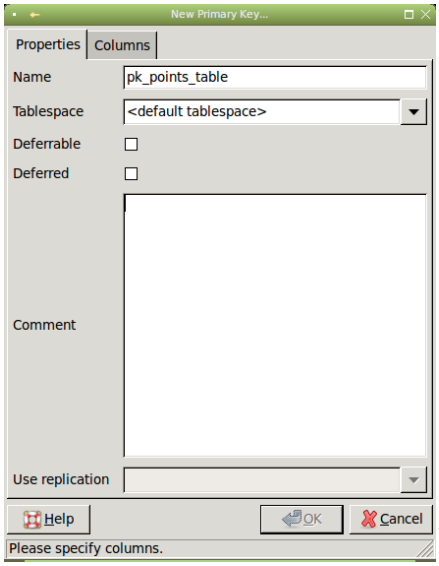


Fill the *Properties* tab as you see in the figure, the table will be called *points*_table. Now let's define the columns (fields) of the table. To do so, click on the *Columns* tab and *Add* a new column, this way:

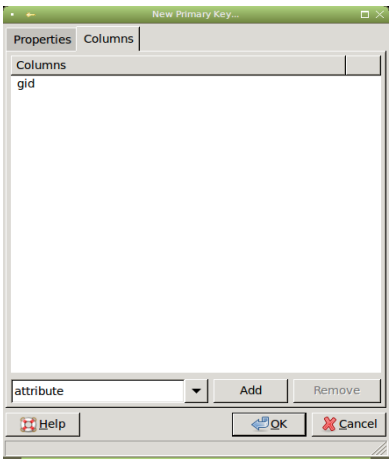


Add another column to the table and name it *attribute*, with Data type *character* and lenght 3.

Now we need to set a field as PRIMARY KEY, which acts as a unique code to distinguish the registers of the table (rows) and access them. A table with no PRIMARY KEY can not be edited. Click on the *Constraints* tab. When you add a primary key you need to specify the name of the constraint, name it *pk_points_table:*



Click on the *Columns* tab and add the *gid* column. You get something like this:



Click on *Ok* to confirm the constraints and click on *Ok* to confirm the creation of the table.

Now you have a table called *points_table* in your database.

Just for your information, the equivalent set of SQL instructions (to create the same table with the same field definition) would be:

```
CREATE TABLE points_table (
    gid serial PRIMARY KEY,
    attribute char(3)
);
```

Which way do you prefer (GUI or SQL)?

We have almost finished, but as you may have noticed, we have no spatial capabilities on the table. To achieve this, we need to add a new column to store the geometry. We will see how to do it in the next step of the tutorial.

6. Adding spatial capabilities to the table

To add geometry columns we can use a *PostGIS* function called *AddGeometryColumn*(). Try to find it in the functions list and identify the arguments of such a function.

Right, there are three functions on the list called *AddGeometryColumn*(). The difference is the set of arguments they receive. As we are working on the *public* schema, we can use the function that receives the smallest number of arguments. The arguments of that function are:

- **table_name**: Name of the table you want to modify by including the geometry column.
- **column_name**: Name of the column that will hold the geometry. Usually called *geom*.
- **srid**: EPSG code of the SRS (Get a complete list of SRS descriptions at http://spatialreference.org). Find the EPSG code of the WGS84 SRS.
- **type**: Geometry type of the table. POINT, LINESTRING or POLYGON, among others.
- **dimension**: Dimension of the SRS used (number of coordinates you use to define a point). As we are going to work with Latitude and Longitude, the dimension will be 2.

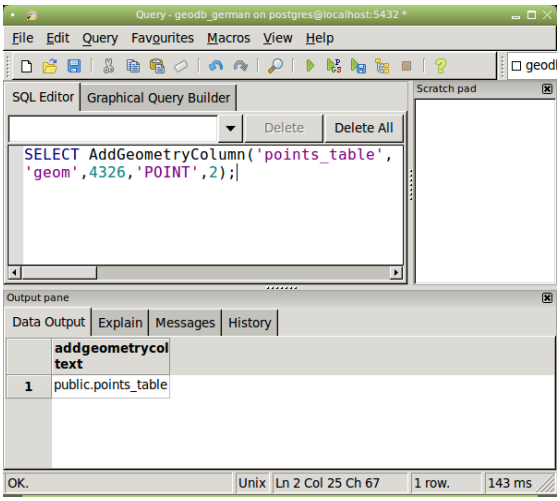
For a better comprehension of these arguments, you can check the *Simple Feature Access for SQL* specification, it has been given to you in the same zip file of this tutorial.

To run the PostGIS functions you need the SQL Query Editor. Click on the button which looks like this:

In the SQL Editor Box type:

```
SELECT AddGeometryColumn('points_table','geom',4326,'POINT',2);
```

And run it by pressing **F5**.



If everything went fine, your table has spatial capabilities, i.e. it can store geographic data. Check the *geometry_columns* table, now it includes your table as a geometry table (*geometry_columns* is a metadata table, it tells you which of the database tables have spatial capabilities).

7. Adding spatial data to the table

To add geographic data to the table you can use a variety of geometry formats. One of them is *Well Known Text* (WKT), in which every geometry is defined in a human-readable way, here are some examples:

```
'POINT(0 0)'
'LINESTRING(1 1, 2 2, 3 4)'
'POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'
```

According to the geometry type you need to use a PostGIS function to read the geometry: *ST_PointFromText*, *ST_LineFromText* or *ST_PolygonFromText*, among others.

Add data to the table using the common SQL instruction *INSERT INTO*:

```
INSERT INTO points_table (geom,attribute) VALUES
 (ST_PointFromText('POINT(-10 5)',4326),'Pt1');

INSERT INTO points_table (geom,attribute) VALUES
  (ST_PointFromText('POINT(5 5)',4326),'Pt2');

INSERT INTO points_table (geom,attribute) VALUES
 (ST_PointFromText('POINT(-20 20)',4326),'Pt3');
```

8. Create line and polygon data

Following the steps to create the *points_table*, create a *lines_table* and a *polygons*_table. You need to create the *gid* (serial) and the *attribute* (character, 3) field for the table. Also add the geometry field by using the AddGeometryColumn function (be careful with the arguments you provide to the function).

Now add line and polygon data to your tables. The WKT representation for each geometry is:

WKT Geometry	Attribute
`'LINESTRING(-10 5, 5 5)'`	'Li1'
`'LINESTRING(-10 -20, 10 -20, 20 25, 40 10, 40 -5)'`	'Li2'
`'POLYGON((-20 10, 10 10, 10 -20, -20 10))'`	'Po1'
`'POLYGON((0 0, 0 30, 30 0, 0 0))'`	'Po2'

You have to construct the *INSERT INTO* SQL instructions to load the data to your line and polygon tables.

Once you have finished this point you will be able to start using PostGIS functions to perform spatial analysis on your data.

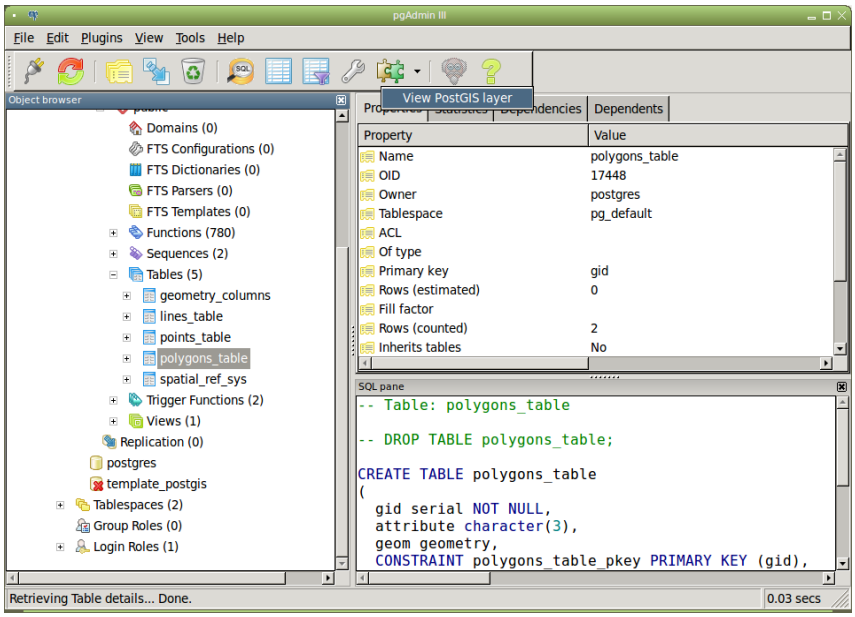
9. Displaying your spatial data (Optional step)

Basically you could use any thick client (Quantum GIS, Kosmo, gvSIG, uDig, among others) to display your PostGIS data. However, we will use another approach, a pgAdmin's plugin.

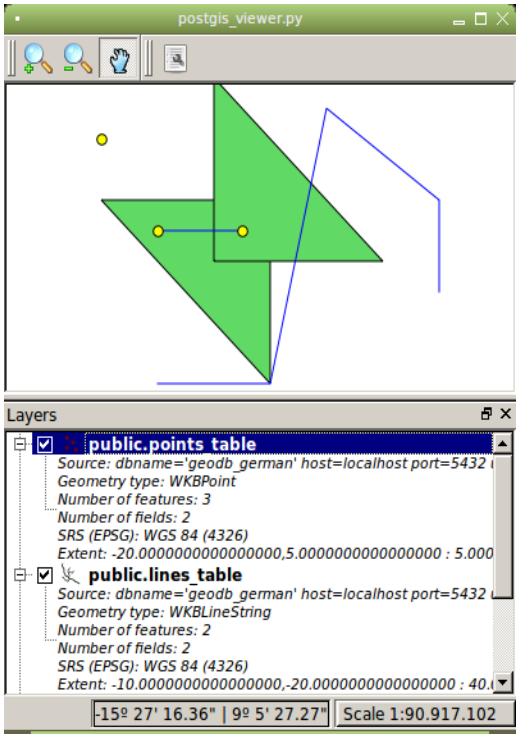
You can display your spatial data in pgAdmin by using the *PostGIS Viewer* plugin. You can find the installation instructions at:

http://geotux.tuxfamily.org/index.php?option=com_myblog&task=view&id=278&Itemid=59&lang=en

To display a spatial table, select it in the main tree and click on the plugin.



You will get this window:



To add more spatial tables to the same map repeat the process (select the table and click on the plugin) with each table.

Try to identify each point, line and polygon you have loaded and which *attribute* corresponds to it. For instance, according to the coordinates the point 'Pt3' is in the upper left part of the map.

10. Topological predicates (from DE-9IM)

Topological predicates are functions to check topological relations between two geometries. As they check relations, they just return boolean values, i.e. True (t) or False (f).

DE-9IM defines the following topological predicates:

Topological Predicate	Meaning
Equals	The Geometries are topologically equal
Disjoint	The Geometries have no point in common
Intersects	The Geometries have at least one point in common (the inverse of Disjoint)
Touches	The Geometries have at least one boundary point in common, but no interior points
Crosses	The Geometries share some but not all interior points, and the dimension of the intersection is less than that of at least one of the Geometries
Overlaps	The Geometries share some but not all points in common, and the intersection has the same dimension as the Geometries themselves
Within	Geometry A lies in the interior of Geometry B
Contains	Geometry B lies in the interior of Geometry A (the inverse of Within)

Let's run one of the functions:

```
SELECT p.attribute, l.attribute, ST_Intersects(p.geom, l.geom)
    FROM points_table p, lines_table l;
```

The query extracts the *attribute* field of two tables (*points_table* and *lines_table)* as well as whether their geometries are related through an intersection. After pressing F5 you will get:

	attribute character(3)	attribute character(3)	st_intersects boolean
1	Pt1	Li1	t
2	Pt1	Li2	f
3	Pt2	Li1	t
4	Pt2	Li2	f
5	Pt3	Li1	f
6	Pt3	Li2	f

Try to infer what would be the result of the *Touches* predicate over the *polygons_table* and the *lines_table*. Now build a query to find the answer. You should get something like this:

	attribute character(3)	attribute character(3)	st_touches boolean
1	Po1	Li1	f
2	Po1	Li2	t
3	Po2	Li1	f
4	Po2	Li2	f

Can you apply the *Overlaps* predicate on the two polygons from the *polygons_table*? You should get they overlap.

	attribute character(3)	attribute character(3)	st_overlaps boolean
1	Po1	Po2	t

Now apply the *Overlaps* predicate on the two lines of the *lines_table*.

	attribute character(3)	attribute character(3)	st_overlaps boolean
1	Li1	Li2	f

10.a. The DE-9IM matrix

According to the *Simple Feature Access for SQL* specification: "the basic approach to comparing two geometries is to make pair-wise tests of the **intersections between the Interiors, Boundaries and Exteriors of the two geometries** and to classify the relationship between the two geometries based on the entries in the resulting 'intersection' matrix." The 'intersection matrix' is known as DE-9IM matrix.

There is one PostGIS function that returns the DE-9IM matrix: *ST_Relate*.

```
SELECT p.attribute, l.attribute, ST_Relate(p.geom, l.geom)
    FROM points_table p, lines_table l;
```

The result is:

	attribute character(3)	attribute character(3)	st_relate text
1	Pt1	Li1	F0FFFF102
2	Pt1	Li2	FF0FFF102
3	Pt2	Li1	F0FFFF102
4	Pt2	Li2	FF0FFF102
5	Pt3	Li1	FF0FFF102
6	Pt3	Li2	FF0FFF102

The first row, which is the relation between the point 1 (Pt1) and the line 1 (Li1), can be read as:

$$\begin{array}{cc} & \boldsymbol{Li1} \\ & \begin{array}{ccc} I(l) & B(l) & E(l) \end{array} \\ \boldsymbol{Pt1} \begin{array}{c} I(p) \\ B(p) \\ E(p) \end{array} & \begin{bmatrix} F & 0 & F \\ F & F & F \\ 1 & 0 & 2 \end{bmatrix} \end{array}$$

Where *I()* is the interior, *B()* the boundary and *E()* the exterior, and *p* and *l* are the point and the line respectively. Boundary, interior and exterior are defined as follows (Strobl, 2007):

Boundary of a geometry object: Is a set of geometries of the next lower dimension.
Interior of a geometry object: Consists of those points that are left (inside) when the boundary points are removed.
Exterior of a geometry object: Consists of points not in the interior or boundary.

The characters returned by the function ST_Relate may have the following values:

0 : Point (The intersection has a dimension 0)
1 : Line
2 : Area
T : {0,1,2} (The intersection may have a dimension 0, 1 or 2)
F : {ø} (The intersection is an empty set)
***** : "Don't care"

Thus, the F0FFFF102 relation is telling us that the point 1 is one of the endpoints of the line1 (look at the first row and the second column of the matrix.) Note that the point 2 has the same relation with the line 1 since it is the other endpoint.

You can get the DE-9IM matrix for every pair of geometries that you provide in a SELECT query.

11. Spatial operators (Getting new geometries)

Spatial operators are functions that construct geometries from other geometries.

According to the *Simple Feature Access for SQL* specification, these are: Intersection, Difference, Union, Symetrical Difference, Buffer and Convex Hull.

Function	Description
Intersection (g1 Geometry, g2 Geometry)	return a geometric object that is the intersection of geometric objects g1 and g2
Difference (g1 Geometry, g2 Geometry)	return a geometric object that is the closure of the set difference of g1 and g2
Union	return a geometric object that is the set union of g1 and

(g1 Geometry, g2 Geometry)	g2
SymDifference (g1 Geometry, g2 Geometry)	return a geometric object that is the closure of the set symmetric difference of g1 and g2 (logical XOR of space)
Buffer (g1 Geometry, d Double Precision)	return a geometric object defined by buffering a distance d around g1, where d is in the distance units for the Spatial Reference of g1
ConvexHull (g1 Geometry)	return a geometric object that is the convex hull of g1

Let's apply the *Union* operator over the polygons:

```
SELECT p1.gid, p1.attribute as att1, p2.attribute as att2,
                ST_Union(p1.geom, p2.geom)
        FROM polygons_table p1, polygons_table p2
                WHERE p1.gid < p2.gid;
```

As you can see, you are getting new geometries but you see them as binary data, actually, as Well Known Binary (WKB).

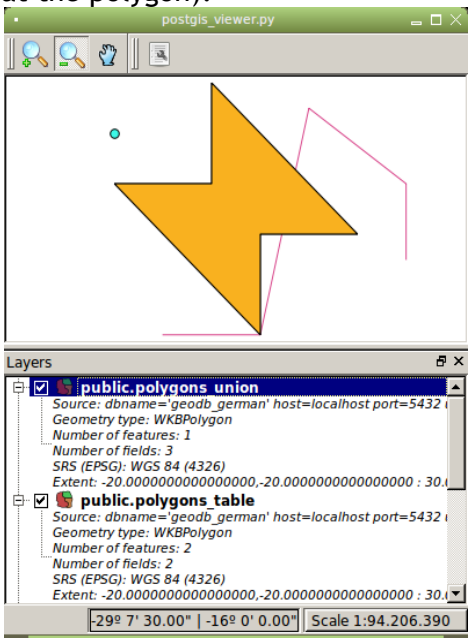
To display these geometries we need to create a table, but wait! We can create tables from *SELECT* statements, so it will not take too much time.

12. Creating spatial tables from spatial operators

We can create tables from a SELECT statement using the word AS (note that this is just for displaying purposes as we are not defining any primary key):

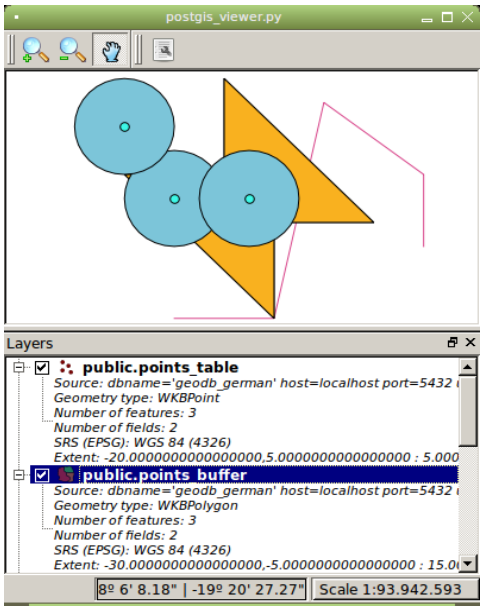
```
        CREATE TABLE polygons_union AS
SELECT p1.gid, p1.attribute as att1, p2.attribute as att2,
                ST_Union(p1.geom, p2.geom)
        FROM polygons_table p1, polygons_table p2
                WHERE p1.gid < p2.gid;
```

This should look like this (look at the polygon):



A buffer example from the points:

```
CREATE TABLE points_buffer AS
SELECT p.gid, p.attribute, ST_Buffer(p.geom, 10)
FROM points_table p;
```



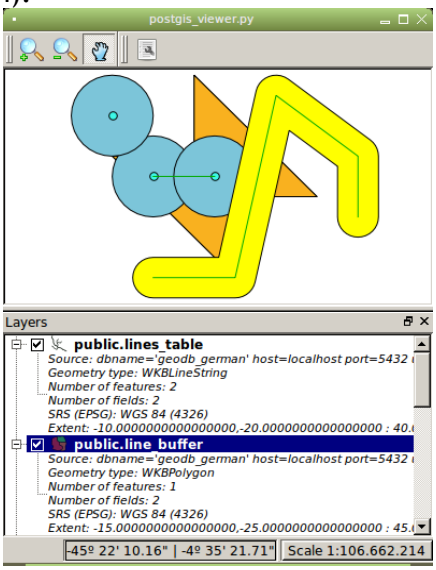
13. Using topological predicates in WHERE clauses

We have seen one of the lines touches the *polygons_table*. What if you need a buffer around that specific line? You would like to filter somehow your lines before applying the buffer.

We can use topological predicates as a constraint to apply spatial operators.

```
CREATE TABLE line_buffer AS
SELECT l.gid, l.attribute, ST_Buffer(l.geom, 5)
FROM lines_table l, polygons_table p
WHERE ST_Touches(l.geom, p.geom);
```

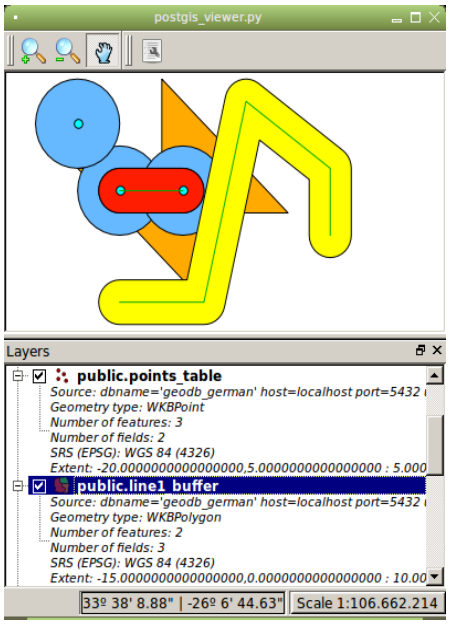
This is the result (yellow polygon):



You could also use the function *ST_Relate* to filter a query by means of the WHERE clause. Using the same example as the step 10.a., to apply a buffer around the line that holds the relation F0FFFF102 with the points of the point_table you would write:

```
CREATE TABLE line1_buffer AS
SELECT l.gid, l.attribute as line, p.attribute as point, ST_Buffer(l.geom, 5)
FROM points_table p, lines_table l
WHERE ST_Relate(p.geom, l.geom, 'F0FFFF102');
```

And this is what you get (red polygon):



That's it!

References

- Ramsey, Paul. *Tips for power users*. Available at http://2010.foss4g.org/presentations_show.php?id=3369
- Ramsey, Paul. *Introduction to PostGIS, Installation - Tutorial - Exercises*. Refractions Research. Canada, 2007.
- *PostGIS manual*. Available at http://postgis.refractions.net/documentation/manual-1.5SVN/ch04.html#DE-9IM
- Open Geospatial Consortium (OGC). *Simple Feature Access for SQL*. 2005.
- Strobl, Christian. *Dimensionally Extended Nine-Intersection Model (DE-9IM)*. 2007.