INFO-H-415 - Advanced Databases

Session 10 - Geographical Databases (1/3)

1 Introduction

PostGIS is an extension of PostgreSQL for storing and analyzing spatial data. It defines data types and operations to process (mostly) vector data. It has extensive support for importing and exporting geographical data from other platforms and supporting visualization tools.

These exercises will provide an overview of what PostGIS, and more generally Geographical Information Systems can do and how to use them. We'll build a new spatial database from a conceptual schema and export the data as Shapefiles, visualize them in a GIS viewer, then import other shapefiles into PostGIS and query them through PostGIS's spatial operators.

We will suppose in this document that PostGIS, QGIS and PostgreSQL are succesfully installed on your system.

1.1 References

- PostgreSQL: `http://www.postgresql.org/docs/9.2/interactive/index.html`
- PostGIS: `http://postgis.org/documentation/manual-2.0/` `http://postgis.net/docs/reference.html`
- QuantumGIS: `http://www.qgis.org/`

2 Database Initialisation and creation

2.1 Initialisation of the environment

A PostGIS database is essentially a Postgres database containing several predefined datatypes and tables.

According to your system and version, you may need to start postgres first. In your system shell, enter:

```
$ initdb                                    # Create a new Postgres database cluster
$ pg_ctl start                              # start postgres
```

We can now create a new database:

```
$ createdb infoh415                         # create your database
$ psql infoh415 -c "CREATE EXTENSION postgis;"      # make your database postgis
```

This builds a number of tables that will be used to store geographic data. Observe the created tables using postgres' shell client. From the OS shell, "psql - -help" displays most of the commands you'll need.

```
$ psql infoh415                             # launch an interactive client
```

Now you should have enter PostgreSQL console (noticable by the new prompt `infoh415=#`). Let's have a look at the special tables created by the PostGIS extension:

```
infoh415=# \d
             List of relations
 Schema |       Name        | Type  | Owner
--------+-------------------+-------+-------
 public | geography_columns | view  | gdejaege
 public | geometry_columns  | view  | gdejaege
 public | raster_columns    | view  | gdejaege
 public | raster_overviews  | view  | gdejaege
 public | spatial_ref_sys   | table | gdejaege
(5 rows)
```

Note: Press "TAB" to use autocompletion and enter \q to quit psql.

We can have a look to a particular table:

```
infoh415=# \d geometry_columns
                          Vue « public.geometry_columns »
      Colonne      |          Type          | Collationnement | NULL-able | Par défaut
-------------------+------------------------+-----------------+-----------+------------
 f_table_catalog   | character varying(256) |                 |           |
 f_table_schema    | name                   |                 |           |
 f_table_name      | name                   |                 |           |
 f_geometry_column | name                   |                 |           |
 coord_dimension   | integer                |                 |           |
 srid              | integer                |                 |           |
 type              | character varying(30)  |                 |           |
```

This table will contain information about all the geometry columns that will be present in our database (see next sections).

2.2 Tables creation

We already created an empty database *infoh415*. We will make it contain the Brussels Region as a polygon, and several cities in and around it.

First, create the tables without geometry columns, as you'd do for any SQL database:

```
CREATE TABLE "regions" (
  "id"   serial primary key,
  "name" varchar(17));

CREATE TABLE "cities" (
  "id"   serial primary key,
  "name" varchar(20));
```

Then, add geometry columns. This relies on a PostGIS function, see

```
ALTER TABLE regions ADD COLUMN geom geometry(MULTIPOLYGON,4326);
ALTER TABLE cities ADD COLUMN geom geometry(POINT,4326);
```

Question: what does 4326 refers to ?

2.3 Tables populating

It's time to populate the database. PostGIS stores coordinates in a binary format, but we can generate it using the ST_GeomFromText function:

```
INSERT INTO "cities" ("name", geom) VALUES
('Ixelles', ST_GeomFromText('POINT(4.377307 50.828844)',
4326)),
('Anderlecht', ST_GeomFromText('POINT(4.311476
50.838283)', 4326)),
('Jette', ST_GeomFromText('POINT(4.336345 50.882989)',
4326)),
('Uccle', ST_GeomFromText('POINT(4.372532 50.796875)',
4326)),
('Sint-Pieters-Woluwe', ST_GeomFromText('POINT(4.434936
50.838888)', 4326)),
('Watermaal-Bosvoorde', ST_GeomFromText('POINT(4.418119
50.799759)', 4326)),
('Zaventem', ST_GeomFromText('POINT(4.474544 50.888983)',
4326));

INSERT INTO regions (name, geom)
VALUES ('Brussel-Hoofstad',
```

```
ST_GeomFromText('MULTIPOLYGON(((4.479645 50.822743,4.457515
50.820229,4.456853 50.817054,4.45187 50.813328,4.45194
50.811219,4.457141 50.812188,4.460937 50.813218,4.458989
50.810348,4.456116 50.808439,4.486724 50.797359,4.475172
50.791591,4.454373 50.783902,4.439379 50.778243,4.435901
50.779246,4.421607 50.775625,4.416667 50.774577,4.402251
50.77087,4.387639 50.765453,4.387318 50.767396,4.384095
50.772136,4.383319 50.772448,4.375609 50.774116,4.364969
50.775656,4.35601 50.776492,4.347556 50.777496,4.338407
50.776461,4.333619 50.778414,4.329975 50.780467,4.324307
50.789305,4.322575 50.79504,4.317525 50.800722,4.309047
50.802291,4.308363 50.803579,4.308465 50.804311,4.30964
50.80717,4.311257 50.811984,4.311522 50.815316,4.307527
50.816388,4.303343 50.815591,4.301754 50.813618,4.301211
50.810931,4.294128 50.809113,4.289011 50.809436,4.285491
50.811489,4.282386 50.812657,4.277631 50.813553,4.268556
50.813893,4.265531 50.816359,4.263138 50.818998,4.256857
50.819707,4.254612 50.82178,4.250028 50.821461,4.251942
50.824982,4.256682 50.828063,4.259648 50.830789,4.260635
50.835269,4.262463 50.837653,4.268154 50.839206,4.275046
50.839319,4.277955 50.840096,4.288646 50.841081,4.290085
50.843701,4.291939 50.845435,4.291858 50.847707,4.290091
50.850681,4.292963 50.852513,4.292229 50.855179,4.291524
50.857034,4.294124 50.859349,4.286354 50.862307,4.284119
50.867627,4.285793 50.870819,4.294436 50.875666,4.299716
50.878105,4.305949 50.882506,4.307293 50.884232,4.305059
50.885981,4.303219 50.887413,4.301968 50.890233,4.302662
50.892273,4.307749 50.893004,4.319537 50.89579,4.324096
50.896998,4.330009 50.899689,4.337384 50.904432,4.341859
50.904421,4.352447 50.905075,4.365646 50.904144,4.378209
50.903121,4.383628 50.901663,4.387152 50.903259,4.390386
50.909639,4.393336 50.913093,4.400666 50.91564,4.41137
50.916696,4.417836 50.914521,4.417774 50.912572,4.420493
50.911558,4.42832 50.906887,4.428396 50.904615,4.430256
50.902533,4.431213 50.90068,4.435338 50.895791,4.431019
50.891181,4.439619 50.882381,4.434035 50.881243,4.431428
50.878931,4.429613 50.875901,4.42742 50.868888,4.430542
50.863415,4.434736 50.864046,4.43563 50.86406,4.439634
50.862739,4.44479 50.861355,4.448267 50.860514,4.451311
50.857312,4.460166 50.856063,4.464033 50.854983,4.465028
50.851913,4.469804 50.846463,4.471775 50.844788,4.471915
50.840488,4.470134 50.836404,4.475079 50.829658,4.479645
50.822743)))', 4326));
```

You can either copy and paste the following, or execute the insertion.sql script, thus:

```
\i insertion.sql
```

Finally, indices are especially important for geometry data and get absolutely essential even for relatively small row counts. (A few thousands.) Add indices to your geometry columns using the following syntax:

```
create index cities_geom_idx on cities USING GIST (geom);
create index regions_geom_idx on regions USING GIST (geom);
vacuum analyze; -- resets query optimizer statistics
```

Let's now check what is inside our tables:

```
infoh415=# select * from regions;
```

You should normally obtain an unreadable result as the geometry colunm seems to contain some random bytes representing the region of Brussels that we just inserted. You may however find a few information about how POSTGIS will interpret these bytes if we look at the content of the `geometry_columns` table:

```
infoh415=# select * from geometry_columns;
 f_table_catalog | f_table_schema | f_table_name | f_geometry_column | coord_dimension | srid |     type
-----------------+----------------+--------------+-------------------+-----------------+------+--------------
 infoh415        | public         | regions      | geom              |               2 | 4326 | MULTIPOLYGON
 infoh415        | public         | cities       | geom              |               2 | 4326 | POINT
```

We see that the geometry column of the table regions countains multipolygons is the 4326 referential system in 2 dimensions.

3 Visualisation

We have seen thaWe have seen that is extremely hard to make any sense of geographical data just by reading coordinates. PostGIS doesn't support visualization directly, but many commercial and open_source GIS can read data from a PostGIS database.

Launch QuantumGIS with the "qgis &" command and, in a new project, add a PostGIS layer. (Layer → add a layer → add a Postgis ...)

You then have to make a new connection to your database. Give your connection a name, and indicate the name of your database in the corresponding fields. Click Connect, select all layers and then click Add. You should see the map displayed in the main window. Try panning and zooming around, as well as hiding and reordering layers with the Layers panel.

4 Importing and Exporting

PostGIS makes it easy to export a table in the commonly used Shapefile format:

```
$ pgsql2shp infoh415 regions
$ pgsql2shp infoh415 cities
```

Look at how many files you get for each table. You can take a look to what each of them do: `https://en.wikipedia.org/wiki/Shapefile`

Open the shapefiles in a new QuantumGIS project by dragging the .shp files in the layer column.

The shp2pgsql command predictably handles the opposite operation. First check out its parameters with the "shp2pgsql –help" command. Note that shp2pgsql only generates SQL statements, it doesn't executes them by itself. To make the actual insert you need to execute the generated .sql files, or pipe shp2pgsql's output directly to psql.

Download the shapefiles.tar.gz file from the exercises web page and import the following files into your PostGIS database:

- bel_city.shp
- bel_dist.shp
- bel_prov.shp
- bel_regn.shp
- belriver.shp

Explore the created tables with psql. Start a new project in QuantumGIS and load all these layers. Define SRID. Rearrange the layers to make sure you see what's on each one. Study the different features with QuantumGIS' Identify Results tool (Ctl + Shift + I).

4.1 SRID

The Shapefiles you imported did not specify a Coordinate Reference System. In psql, update all new tables to use WGS84.

5 Spatial queries

Write down and execute the following queries:

1. Get the SRID from table cities. (ST_SRID)
2. Get a textual description for this SRID.
3. Get the dimension of geographical objects in that table. (ST_Dimension, ST_CoordDim)
4. Get the geometry type of these objects. (*Hint: explore the geometry_columns table*, ST_GeometryType)
5. Compute the distance between the cities of IXELLES and BRUGES. (ST_DistanceSphere, ST_Distance, ST_Transform)
6. Compute the bounding rectangle for the BRABANT province. (ST_Envelope)
7. Compute the geographical union of the bel_regn and bel_prov tables. (ST_Union)
8. Compute the length of each river. (ST_Length2DSpheroid)
9. Create a table containing all cities that stand less than 1000m from a river. (ST_DistanceSphere)
10. For each river, compute the length of its path inside each province it traverses. (ST_Intersection, ST_Intersects)

INFO-H-415 - Advanced Databases

Solutions for Session 10 - Geographical Databases (1/3)

4 Importing shapefiles

By creating an sql script then executing it with psql:

```
$ shp2pgsql -W "latin1" bel_city.shp > shp_insert_script.sql
$ shp2pgsql -W "latin1" bel_dist.shp >> shp_insert_script.sql
$ shp2pgsql -W "latin1" bel_prov.shp >> shp_insert_script.sql
$ shp2pgsql -W "latin1" bel_regn.shp >> shp_insert_script.sql
$ shp2pgsql -W "latin1" belriver.shp >> shp_insert_script.sql
$ psql <database_name> -f shp_insert_script.sql
```

4.1 Update SRID

```
SELECT UpdateGeometrySRID('bel_regn', 'geom', 4326);
SELECT UpdateGeometrySRID('bel_city', 'geom', 4326);
SELECT UpdateGeometrySRID('bel_dist', 'geom', 4326);
SELECT UpdateGeometrySRID('bel_prov', 'geom', 4326);
SELECT UpdateGeometrySRID('belriver', 'geom', 4326);
```

5 Requests

1. Get the SRID from table bel_city.

   ```
   SELECT ST_SRID(geom) FROM bel_city LIMIT 1;
   ```

2. Get a textual description for this SRID.

   ```
   SELECT srtext from spatial_ref_sys where srid=4326;
   ```

3. Get the dimension of geographical objects in that table.

   ```
   -- Inherent dimension:
   SELECT ST_Dimension(geom) from bel_city limit 1;
   -- Coordinate dimension:
   SELECT ST_CoordDim(geom) from bel_city limit 1;
   -- or
   SELECT coord_dimension from geometry_columns WHERE f_table_name = 'bel_city';
   -- You can check the coherence of the above results with:
   SELECT ST_AsText(ST_POINTS(geom)) from bel_city limit 1;
   -- Each entity is composed of one point -> inherent dimension = 0
   -- but the coordinates of these points are in two dimensions
   ```

4. Get the geometry type of these objects.

   ```
   -- easy way, look at the type of each geometry object of each row in
   -- the bel_city table
   SELECT ST_GeometryType(geom) FROM bel_city limit 3;
   -- more efficient way, use the geometry_columns table
   SELECT type FROM geometry_columns WHERE f_table_name = 'bel_city';
   ```

5. Compute the distance between the cities of IXELLES and BRUGES.

```
SELECT ST_Distance((SELECT geom FROM bel_city where name= 'Ixelles'),
                   (SELECT geom FROM bel_city where name='Brugge'));
```

ST_Distance gives us a result which is in the units of our spatial reference (SRID 4326 which correspond to WGS 84) and is therefore an angle. It is not really "human useful". A first possibility to obtain readable results is to change the spatial reference system of the geometries.

```
SELECT ST_Distance(
     ST_Transform((SELECT geom FROM bel_city where name= 'Ixelles'), 3812),
     ST_Transform((SELECT geom FROM bel_city where name='Brugge'), 3812));
```

SRID 3812 corresponds to Belgian Lambert 2008 which is a conic representation adapted for Belgium. Since it is a conic (plane) representation, its units are meters.

```
SELECT ST_DistanceSphere((SELECT geom FROM bel_city where name= 'Ixelles'),
                         (SELECT geom FROM bel_city where name='Brugge'));
```

ST_DistanceSphere gives an approximation of the distance on a sphere without changing the spatial reference system.

Further information can be found with the following links:

- `http://postgis.net/workshops/postgis-intro/geography.html` (Calculating distances with geographic instead of geometric objects.)
- `http://postgis.net/docs/ST_Distance.html`

6. Compute the bounding rectangle for the BRABANT province.

```
SELECT ST_AsText(ST_Envelope(geom)) FROM bel_prov WHERE name='Brabant';
```

If you want to be able to visualise the result using qgis, insert the result in a new table:

```
CREATE TABLE Temp As
SELECT ST_Envelope(geom) FROM bel_prov WHERE name='Brabant';
```

The same procedure can be used to visualize the output of other queries.

7. Compute the geographical union of the bel_regn and bel_prov tables.

```
SELECT ST_AsText(ST_Union(geom))
FROM (SELECT geom FROM bel_regn
      UNION SELECT geom FROM bel_prov) AS g;
```

8. Compute the length of each river

```
SELECT name, ST_Length2DSpheroid(geom,
                                 'SPHEROID["GRS_1980",6378137,298.257222101]')
FROM belriver;
```

9. Create a table containing all cities that stand less than 1000m from a river.

```
SELECT DISTINCT ON (c.gid) c.gid, c.id, c.name, c.geom
FROM bel_city c JOIN belriver r ON ST_DistanceSphere(c.geom, r.geom) < 1000;
```

You can also make use of the geography entity proposed by PostGIS:

```
CREATE TABLE belriver_geog AS
SELECT gid, id, name, Geography(geom) AS geog FROM belriver;
CREATE TABLE bel_city_geog AS
SELECT gid, id, name, Geography(geom) AS geog FROM bel_city;
CREATE TABLE river_cities AS
SELECT DISTINCT ON (c.gid) c.gid, c.id, c.name, c.geog FROM bel_city_geog c
    JOIN belriver_geog r ON ST_DWithin(c.geog, r.geog, 1000);
```

10. For each river, compute the length of its path inside each province it traverses.

```
SELECT r.name,
       p.name,
       ST_Length2D_Spheroid(ST_Intersection(r.geom, p.geom),
                            'SPHEROID["GRS_1980", 6378137, 298.257222101]')
FROM belriver r, bel_prov p WHERE ST_Intersects(r.geom, p.geom);
```