Geographical Databases: PostGIS

Introduction

PostGIS is an extension of PostgresSQL 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.

References

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

Connection parameters

We'll be using PostGIS 2.0.1 on Postgres 9.2.1.

Host: 164.15.78.59
Database name: infoh415_2 — infoh415_20
Donwload scripts at http://cs.ulb.ac.be/public/teaching/infoh415/tp

Creating a new database

A PostGIS database is essentially a Postgres database containing several predefined datatypes and tables. The infoh415_## databases are empty at the moment. Choose a number and use the following command to make them postgis databases:

```
psql –h 164.15.78.59 -d infoh415_1 -c "CREATE EXTENSION
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 –h 164.15.78.59 -d infoh415_1
psql (9.2.1)
Type "help" for help.

postgis=# \d
```

```
               List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | geography_columns | view  | sboucher
 public | geometry_columns  | view  | sboucher
 public | raster_columns    | view  | sboucher
 public | raster_overviews  | view  | sboucher
 public | spatial_ref_sys   | table | sboucher
(5 rows)

postgis=# \d geography_columns
     View "public.geography_columns"
       Column       |  Type   | Modifiers
--------------------+---------+-----------
 f_table_catalog    | name    |
 f_table_schema     | name    |
 f_table_name       | name    |
 f_geography_column | name    |
 coord_dimension    | integer |
 srid               | integer |
 type               | text    |
```

Creating a geographical database

Build your own

At first we'll create a small database from scratch, containing the Brussels Region as a polygon, and several cities in and around it.

Creating the tables

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 http://postgis.org/docs/AddGeometryColumn.html) :

```
SELECT
AddGeometryColumn('regions','geom','4326','MULTIPOLYGON',2);
SELECT AddGeometryColumn('cities','geom','4326','POINT',2);
```

Question: what does the ‘4326’ refer to?

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 geographical 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
```

Visualizing data

Obviously it is extremely hard to make any sense of geographical data just by reading geographical 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. Create a new connection with the following parameters:

Host: 164.15.78.59
Port: 5432
Database: // your database

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.

Exporting and Importing

Exporting

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

```
pgsql2shp –h 164.15.78.59 postgis regions
pgsql2shp –h 164.15.78.59 postgis cities
```

Look at how many files you get for each table. What does each of them do ?

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

Importing

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 Features` tool.

SRID

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

Spatial queries

Write down and execute the following queries:

1. Get the SRID from table `cities`.
2. Get a textual description for this SRID.
3. Get the dimension of geographical objects in that table.
4. Get the geometry type of these objects.
5. Compute the distance between the cities of IXELLES and BRUGES.
6. Compute the bounding rectangle for the BRABANT province.
7. Compute the (geographical) union of the bel_regn and bel_prov tables.

8. Compute the length of each river
9. Using a buffer, create a table containing all cities that stand less than 1000m from a river.
10. For each river, compute the length of its path inside each province it traverses.
    Hints:
    a. Compute the intersection between rivers and provinces
    b. Create a table containing for each river and each province the geometry of the river inside it
    c. Create a new table filtering the previous table for rows without a geometry
    d. Compute lengths for each of these rows
    e. Display the final result