Exercises: PostgreSQL and PostGIS in a Nutshell

Michael Wagner (mwagner@allspatial.info)

July 14, 2016

Contents

List of Figures

List of Listings

List of Abbreviations

DBMS	Database Management System
OSS	Open Source Software
SQL	Structured Query Language

1 Introduction

The main purpose of this training is to get you ready to work with PostgreSQL and PostGIS swiftly. Because of that almost all theory on Database Management System (DBMS) is skipped and touched only when it is absolutely necessary for performing a certain task.

PostgreSQL is the most advanced and powerful DBMS among the DBMSs based on Open Source Software (OSS). It can also compete with well-known proprietary DBMSs such as Oracle, SQL Server, Informix, etc. PostGIS is the spatial extension to PostgreSQL. It adds new data types for geometry and geography and spatial capabilities to the database. As such, it turns a standard database into a spatial database / geodatabase. As PostgreSQL PostGIS is OSS.

From the *Software* folder install the PostgreSQL version that fits your operating system (32 or 64 Bit). The installation is straightforward and you can just accept all default settings. You will be asked to set a password for the database superuser *postgres* at some point during the installation. Remember that password well or write it down. On completion of the installation skip/cancel the offer to launch the *Application Stack Builder*. In the *Software* folder you will also find the installer for PostGIS. Pick the installer that fits your operating system version (32 or 64 Bit) and run it. At the end of the PostGIS installation process you will be asked three questions – you can answer all of them with *Yes*. If everything went well you should now have PostgreSQL/PostGIS up and runniing on your computer.

2 Basic administrative tasks

Basic administrative tasks include creating new users and groups, databases, schemas and tables as the most frequent ones. For the administration of a PostgreSQL DBMS there are plenty of tools available (OSS and proprietary). A great and free one that is installed automatically with PostgreSQL is *pgAdmin*. Find in the list of installed applications on your computer and launch it. In the *Object Browser* on the left side under *Server Groups/Servers* you will find by default one server connection called *PostgreSQL 9.5 (localhost:5432)*. This server connection is auto-

matically set up for the database superuser (and currently your only user) *postgres*. Double-click that connection to actually connect. You will be asked to enter the super user password that you assigned during the PostgreSQL installation (Figure 1).

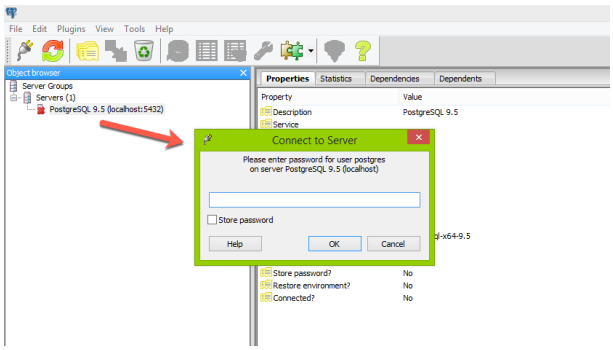

Figure 1: Connecting as superuser

You can choose to store the password so you won't be asked again in the future. Once connected you will see one single database listed that is called *postgres*. This is a system database that PostgreSQL needs to work properly and it cannot be modified or deleted. Under *Login Roles* you should see one single role *postgres*, that's the database superuser you just used to log on. Select the *postgres* database and click the SQL icon in the toolbar to open an SQL query dialog (Figure 2). Structured Query Language (SQL) is a standard to query databases and to create and manipulate database objects (such as users, tables, etc.). In the SQL dialog you can run arbitrary SQL statements.

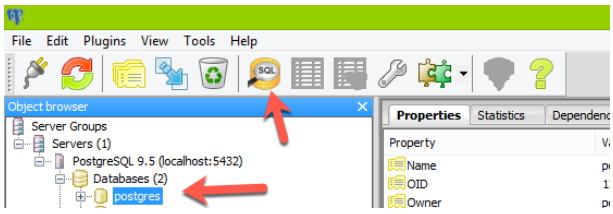

Figure 2: Open an SQL query dialog

2.1 Creating users and groups

Users in PostgreSQL can have the following main privileges:

- SUPERUSER: All possible privileges
- CREATEDB: Can create databases
- CREATEROLE: Can create users and groups
- LOGIN: Can log on to a database

A user role should have the minimum privileges required for the work that user is supposed to do. Thus, it is not a good idea to work as user *postgres* all the time. Let's create another user *geodb_admin* who has the privileges to create new users, databases and to log on.

Listing 1: Creating a user

```
1 CREATE ROLE geodb_admin CREATEROLE CREATEDB LOGIN PASSWORD '
    geodb_admin';
```

Type that statement in the SQL query dialog and hit the button with the green arrow to run/execute the statement. Statements you have already run you can comment out by putting a double-dash in front of the line. This way you "preserve" them but they will be ignored during query execution, e.g.

Listing 2: Commenting out an SQL statement

```
1 --CREATE ROLE mwagner LOGIN PASSWORD 'mwagner';
```

Create another user named *user_name* (replace user_name with your name) with the privilege to log on only. Create two groups *gis_view* and *gis_update*. A group is created by creating a role without any privilege (Listing 3).

Listing 3: Creating groups

```
1 CREATE ROLE gis_view;
2 CREATE ROLE gis_update;
```

A user is made member of a group by granting the group to the user. Make your role *user_name* member of group *gis_update*:

Listing 4: Adding a user to a group

```
1 GRANT gis_update TO user_name;
```

Group members automatically inherit the privileges granted to the group. It is good practice to grant rights on tables, views, etc. to groups instead of individual user roles. The advantage is that when changing or revoking a privilege is required, it has to be done once only for the group and not for each individual user.

To delete a user or group role use the statement as follows:

Listing 5: Deleting a role

```
1 DROP ROLE role_name;
```

By the end of this exercise you should have 3 user roles (*postgres*, *geodb_admin*, *user_name*) and 2 group roles (*gis_update*, *gis_view*) with role *user_name* being a member of group *gis_update*. Refresh the *Group Roles* and *Login Roles* in the *Object Browser* in *pgAdmin's* main window to verify that the group and user roles were created.

2.2 Creating databases

2.2.1 Preparing a template database

Create a database named *template_postgis* like so:

Listing 6: Creating a database

```
1 CREATE DATABASE template_postgis;
```

This database will serve as a template for all geodatabases you will ever create, i.e. new geodatabases will be created as copy of database *template_postgis*. Refresh *Databases* in the *Object Browser* of *pgAdmin's* main window to verify that the database was created. Select the new database *template_postgis* in the *Object browser* and open an SQL query dialog for this database. Turn this database into a spatially enabled database by loading the PostGIS extension:

Listing 7: Creating the PostGIS extension

```
1 CREATE EXTENSION postgis;
```

This will add spatial capabilities to database *template_postgis*. Disallow that anyone can create tables and other database objects in schema *public*:

Listing 8: Revoking a privilege

```
1 REVOKE CREATE ON SCHEMA public FROM public;
```

Schemas will be discussed in detail in the next chapter. For the moment just run the statement like this. Finally declare database *template_postgis* as template database. This is required for the database to serve as a template for other databases:

Listing 9: Declaring a template database

```
1 UPDATE pg_database SET datistemplate = true WHERE datname = '
    template_postgis';
```

With this you are done setting up a template database for all geodatabases to be created in the future. Save the SQL statements you have run so far and close the SQL query dialog. In the *Object Browser* right-click on database *template_postgis* and select *Disconnect database*. This is important for the database to be copied as described in the next exercise.

2.2.2 Creating the training database

Use the user role *geodb_admin* you created during the previous exercise to create a new database named *my_first_geodb* based on database *template_postgis*. To run an SQL statement as *geodb_admin* you have to create a new server connection in *pgAdmin* first. Click the first button in the toolbar of pgAdmin's main window and set the connection parameters as shown in Figure 3.

You will now find two server connections in the *Object Browser*, one as user *postgres* and one as user *geodb_admin*. Under the connection of *geodb_admin* select database *postgres* in the *Object Browser* and open a new SQL query dialog. Run the statement as follows to create the database *my_first_database*:

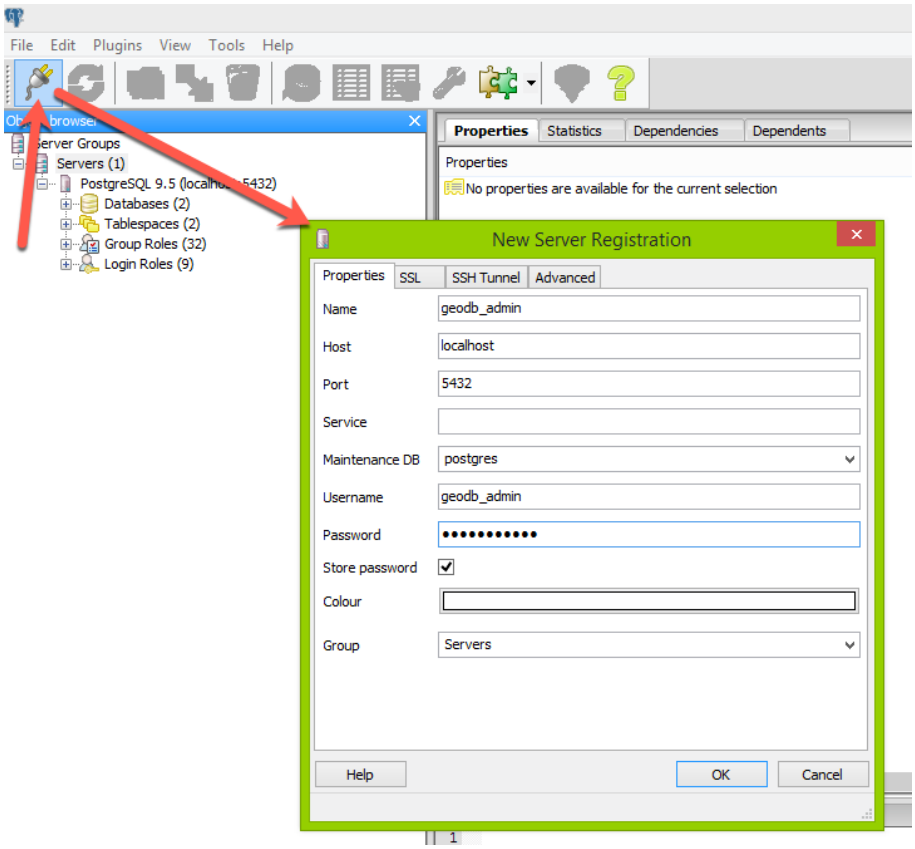

Figure 3: Creating a new server connection

Listing 10: Creating the training database

```
1 CREATE DATABASE my_first_geodb TEMPLATE = template_postgis;
```

This will create the new database as copy of *template_postgis*. Hence, *my_first_geodb* is a geodatabase without any additional steps required. Finally grant a CREATE privilege on the database to user *user_name*:

Listing 11: Granting a CREATE privilege on a database

```
1 GRANT CREATE ON DATABASE my_first_geodb TO user_name;
```

This will allow the user *user_name* to create a schema in the database in a later exercise. Deleting a database is done like so:

Listing 12: Deleting a database

```
1 DROP DATABASE my_db;
```

2.3 Creating schemas

A database may contain one or more named schemas. A schema is a bit like a folder in a file system but schemas cannot be nested. Schemas are used to:

- Allow many users to use one database without interfering with each other (Two tables can have the same name if they exist in different schemas.)
- Organize database objects into logical groups

A schema is created like so:

Listing 13: Creating a schema

```
1 CREATE SCHEMA cadastre;
```

A table in a schema can be accessed by *schema.table* e.g. a table *parcel* in a schema *cadastre*:

Listing 14: Accessing objects in a schema

```
1 SELECT * FROM cadastre.parcel;
```

For anyone to access any object in a schema a USAGE privilege has to be granted on that schema by the owner of the schema:

Listing 15: Granting USAGE privilege

```
1 GRANT USAGE ON SCHEMA cadastre TO public;
```

As user *geodb_admin* create two schemas *vector_data* and *raster_data* in database *my_first_geodb*. To do that you have to select database *my_first_geodb* in the *Object Browser* and then open a new SQL query dialog.

Listing 16: Creating example schemas

```
1 CREATE SCHEMA vector_data;
2 CREATE SCHEMA raster_data;
```

Grant the USAGE right on both schemas to *public*:

Listing 17: Granting USAGE privilege on example schemas

```
1 GRANT USAGE ON SCHEMA vector_data TO public;
2 GRANT USAGE ON SCHEMA raster_data TO public;
```

A schema and all the contained objects (tables, views, etc.) are deleted as follows:

Listing 18: Deleting a schema

```
1 DROP SCHEMA cadastre CASCADE;
```

2.4 Creating tables

Tables are created using the CREATE TABLE statement:

Listing 19: Creating a spatial table

```
1  CREATE TABLE vector_data.river (
2  gid SERIAL PRIMARY KEY,
3  name VARCHAR(50),
4  length INT,
5  depth DECIMAL(3, 2),
6  polluted BOOLEAN DEFAULT FALSE,
7  date_last_checked DATE
8  );
9
10 SELECT AddGeometryColumn('vector_data', 'river', 'geometry', 32740,
       'MULTILINESTRING', 2);
```

Lines 1-8 will create a standard (i.e. not a spatial) table *river* in schema *vector_data*. To add a geometry column to this table the statement in line 10 is required. The arguments to be provided to the *AddGeometryColumn* functions are these:

- Schema name
- Table name
- Geometry column name
- Spatial Reference System ID
- Geometry type
- Geometry dimension

Any time you want to add a geometry column to a table you use the statement in line 10 and adjust it accordingly.

By default only the table owner has the privilege to select, insert, update or delete data from a table. For other users to access the table's data the table owner (e.g. *geodb_admin*) has to grant the required privileges:

Listing 20: Granting privileges on table and sequence

```
1 GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE vector_data.river TO
    gis_update;
2 GRANT SELECT ON TABLE vector_data.river TO gis_view;
3 GRANT USAGE ON SEQUENCE vector_data.river_gid_seq TO gis_update;
```

In line 1 full access is granted to group *gis_update* (and to anyone being a member). In line 2 a SELECT privilege is granted to group *gis_view* (and to anyone being a member). In line 3 a USAGE privilege is granted to group *gis_update* on a sequence. The *river* table's primary key is a serial number i.e. a number that is automatically advanced by 1 every time a new row is inserted into the table. PostgreSQL creates a so called sequence for each serial number that requires a USAGE privilege to be used (i.e. to be advanced). The final thing you will want to do is to create a spatial index on the geometry column. This will speed up spatial queries involving this table significantly:

Listing 21: Creating a spatial index

```
CREATE INDEX river_geometry_idx ON vector_data.river USING gist(
    geometry);
```

A table is deleted as follows:

Listing 22: Deleting a schema

```
DROP TABLE vector_data.river;
```

Create the *river* table, grant the privileges and create the spatial index. Launch QGIS and load the *district* Shapefile from the *Data/Shapefiles* folder. Now load the *river* table from your database. You will have to create a new connection to a PostgreSQL/PostGIS database first (Figure 4). Use your username as the username to connect.

In the *Add PostGIS Table(s)* dialog click the *Connect* button, you should see the *river* table you previously created under schema *vector_data* (Figure 5). Select and add the table to QGIS. Toggle editing and digitise ten fictive rivers on Mahe. Set the *polluted* field to true for some of the rivers. Save your changes. Use the *Field Calculator* to populate the *length* field of the river table. Again, save your changes.

2.5 Creating views

You can create views over queries you need to run frequently to:

- Avoid typing the query each time the query is needed
- Give a name to the query

If the view contains a geometry column it becomes a spatial view that can be loaded and visualised in QGIS, e.g.:

Listing 23: Creating a view

```
CREATE OR REPLACE VIEW polluted_rivers AS SELECT * FROM vector_data
    .river WHERE polluted = true;
```

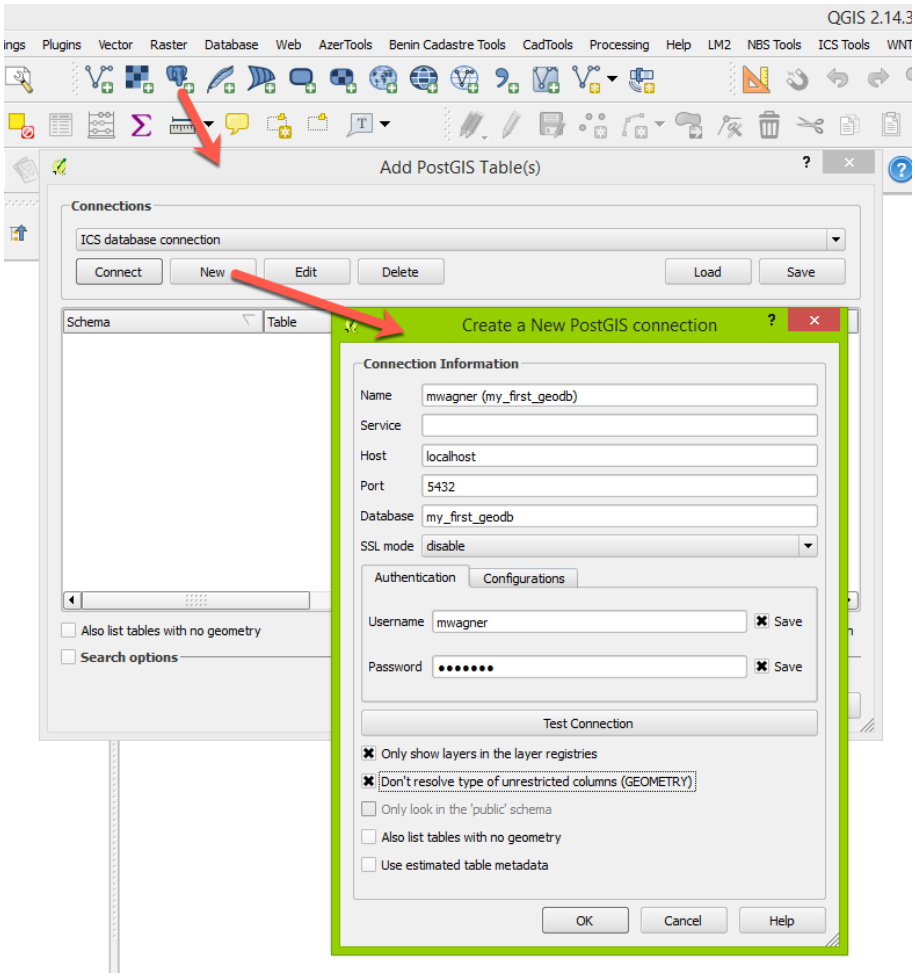

Figure 4: Connecting to PostgreSQL from within QGIS

As with the tables the required privileges have to be granted on the view for other users to access the view:

Listing 24: Granting privileges on spatial view

```
1 GRANT SELECT ON VIEW vector_data.polluted_rivers TO gis_update,
    gis_view;
```

Create the view *polluted_rivers* as *geodb_admin* and grant the privileges for others to SELECT from the view.

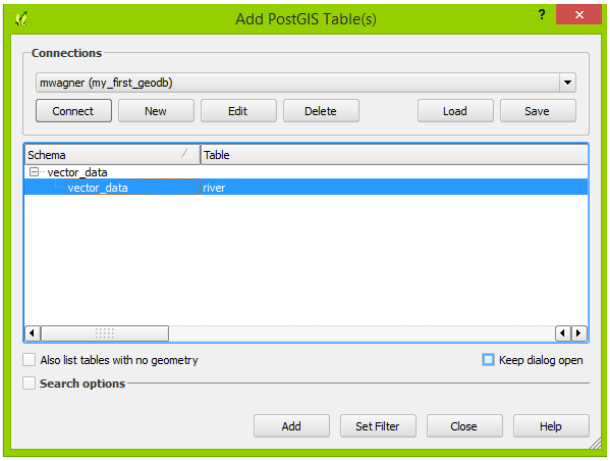

Figure 5: List of spatial tables

3 Importing spatial data

PostGIS supports the storage and analysis of vector and raster data.

3.1 Importing vector data

There is a number of tools to import vector data into a PostgreSQL/PostGIS database but we will use the *PostGIS Shapefile and DBF Loader* that came with the PostGIS installation. Find and launch the tool and enter the database connection parameters (Figure 6).

Click the *Add File* button and select all Shapefiles from the *Data/Shapefiles* folder except of the *river* file (you already have a *river* table). You can select multiple Shapefiles at once. The Loader tries to import the data from the Shapefiles into schema *public* by default. We don't want that, they should go in schema *vector_data*. Double-click in the *Schema* field of each table and change the schema from *public* to *vector_data*. The SRID is 0 by default, we have to change that to 32740. Again do that for each table entry. Finally we want to change the name of column that will hold the geometry from *geom* to *geometry* (Figure 7).

Once all preparation has been done click the *Import* button. The import might take a few seconds. Once finished refresh the *Object Browser* in *pgAdmin* and

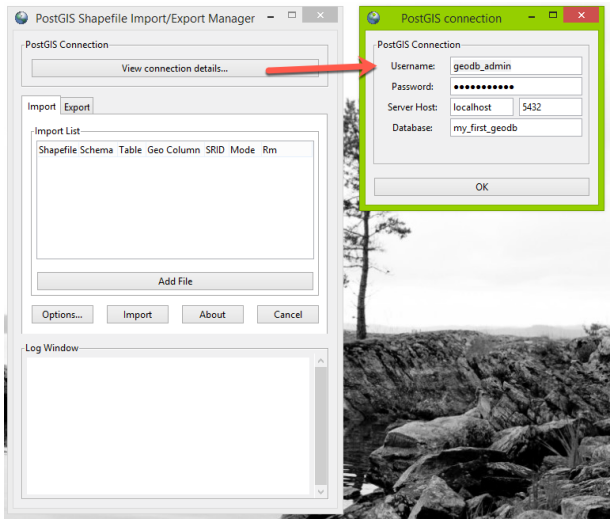

Figure 6: Setting the database connection parameters for the Loader

verify that the new tables are there. You should find 11 tables now in schema *vector_data*. During the import spatial indices were created automatically for all new tables. What is still missing is to grant the privileges for other users to be able to access the new tables. You have to grant the same privileges on each new table and sequence that you previously granted on table river and the relating sequence. You can copy and paste these statements and just modify table and sequence name. This step has to be done by *geodb_admin*, the owner of all tables.

After granting the required privileges you can load the new tables into QGIS.

3.2 Importing raster data

Importing raster data has to be done via command prompt, no graphical tool is yet provided for raster import. Open a Command Prompt by typing *cmd* in the Windows Search. Open the Windows File Explorer and navigate to the *bin* directory of your PostgreSQL installation. Copy that path from the address bar. In the Command Prompt type *cd* (change directory) and paste the path you just copied from the File Explorer. It should look similar to Listing 25. Hit Enter.

Listing 25: Changing directory

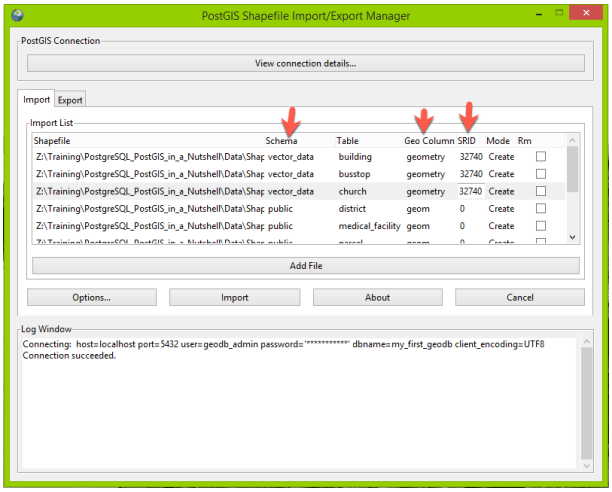

Figure 7: Correcting the import settings

```
cd "C:\Program Files\PostgreSQL\9.5\bin
```

Changing the directory is required for the raster data import tool (*raster2pgsql*) to be found. Finally type:

Listing 26: Importing raster data into the database

```
raster2pgsql -I -C -e -Y -F -s 32740 -t auto -l 2,4,8,16,32,64 Z:\
    PostgreSQL_PostGIS_in_a_Nutshell\Data\DEM\Mahe_DEM.tif
    raster_data.mahe_dem | psql -U geodb_admin -d my_first_geodb
```

Change the path to your DEM file as required end hit Enter. The import might take several minutes depending on your computer's performance. As a result you will find seven new tables in schema *raster_data*. The first one is the main table with the raster data in original resolution (*mahe_dem*). The other tables are overview levels of the original raster with lower resolutions. They will make loading the raster data in QGIS significantly faster. As for the parameters used with the import tool check the PostGIS documentation for a detailed description. Similar to the vector data tables the raster data tables cannot be accessed by other users yet directly after the import. You have to grant the required privileges first as *geodb_admin*:

Listing 27: Granting privileges on raster tables

```
GRANT SELECT ON raster_data.mahe_dem TO gis_update, gis_view;
GRANT SELECT ON raster_data.o_2_mahe_dem TO gis_update, gis_view;
GRANT SELECT ON raster_data.o_4_mahe_dem TO gis_update, gis_view;
GRANT SELECT ON raster_data.o_8_mahe_dem TO gis_update, gis_view;
GRANT SELECT ON raster_data.o_16_mahe_dem TO gis_update, gis_view;
GRANT SELECT ON raster_data.o_32_mahe_dem TO gis_update, gis_view;
GRANT SELECT ON raster_data.o_64_mahe_dem TO gis_update, gis_view;
```

A read-only (SELECT) privilege should be sufficient for the raster data tables since it is unlikely that these tables need modification. In QGIS open the *DB Manager* from the *Database* menu. Under *PostGIS* find your connection and navigate to schema *raster_data*. Drag and drop table *mahe_dem* in the QGIS map window (Figure 8).

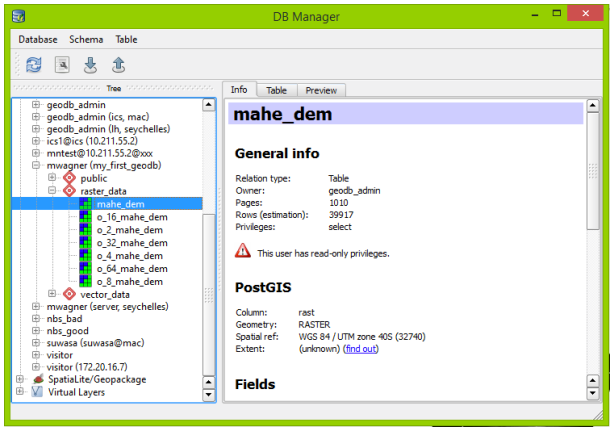

Figure 8: Add a PostGIS raster layer

4 Spatial analysis

Let the fun begin. As user *user_name* create a schema named *user_name*. Any views you will create during this exercise you create in that schema. Before creating the schema you need to create a new server connection in *pgAdmin* for user

user_name. Then select database *my_first_geodb* and open a new SQL query dialog. Create the schema and then run the queries as follows:

- List parcel number, gid and description of all parcels with invalid geometry.

```
SELECT id AS parcel_no, gid, ST_IsValidReason(geometry) AS
    reason FROM vector_data.parcel WHERE NOT ST_IsValid(
    geometry);
```

- List gid and geometry as GeoJSON for the first 3 buildings.

```
SELECT gid, ST_AsGeoJSON(geometry, 1) FROM vector_data.
    building LIMIT 3;
```

- List parcel number and area of the 5 largest parcels.

```
SELECT id, ST_Area(geometry) AS area_m2 FROM vector_data.
    parcel ORDER BY area_m2 DESC LIMIT 5;
```

- List parcel number and geometry (as plain text) of the 3 smallest parcels.

```
SELECT id, ST_AsText(geometry) FROM vector_data.parcel ORDER
    BY ST_Area(geometry) LIMIT 3;
```

- How many parcels are crossed by a river?

```
SELECT count(a.*) FROM vector_data.parcel a, vector_data.river
     b WHERE ST_Intersects(a.geometry, b.geometry);
```

- Find all donut buildings (buildings with holes) and create a spatial view to visualise them in QGIS.

```
CREATE OR REPLACE VIEW user_name.donut_buildings AS SELECT *
    FROM vector_data.building WHERE ST_NRings(geometry) > 1;
```

- List the church name and terrain elevation at each churches location.

```
SELECT b.name as church_name, round(st_value(a.rast, b.
    geometry)) AS elevation FROM raster_data.mahe_dem a,
    vector_data.church b WHERE ST_Intersects(a.rast, b.geometry
    ) ORDER BY elevation;
```

- Show the number of parcels per district.

```
SELECT a.name as district_name, count(b.*) as parcel_count
    FROM vector_data.district a, vector_data.parcel b WHERE
    ST_Intersects(a.geometry, b.geometry) group by a.name ORDER
     BY a.name;
```

Based on the SQL statements for the example queries try to solve the tasks as follows:

1. Of the rivers you created find the three longest ones.
2. Find the police station with the largest distance to any health care facility.
3. Create a spatial view to show all buildings built over a parcel's boundary.
4. What's the total length of road segments within Anse Royale (in km)?
5. Create a spatial view for a 500m buffer around each health care facility.
6. Find the schools that have no busstop within 200m?

Make use the PostGIS reference manual that you will find in the *Software* folder.