CHAPTER 15

Module: PostGIS

PostGIS is an extension to PostgreSQL which allows it to handle and process geographic data. In this module, we'll learn how to set up and use the geographic functions that PostGIS offers.

While working through this section, you may want to keep a copy of the PostGIS cheat sheet available at Boston GIS user group[1]. Another good friend is the online[2] PostGIS documentation.

See also PostGIS online[3].

15.1 Lesson: PostGIS Setup

Setting up PostGIS functions will allow you to access spatial functions from within PostgreSQL.

The goal for this lesson: To install spatial functions and briefly demo their effects.

15.1.1 Installing under Ubuntu

Postgis is easily installed from apt.

```
sudo apt-get install postgis
sudo apt-get install postgresql-9.1-postgis
```

Really, it's that easy...

15.1.2 Installing under Windows

Visit the download page[4].

[1] http://www.bostongis.com/postgis_quickguide.bqg
[2] http://postgis.refractions.net/documentation/manual-1.5/
[3] http://postgisonline.org/
[4] http://www.postgresql.org/download/

Now follow this guide[5].

A little more complicated, but still not hard. Note that you need to be online to install the postgis stack.

15.1.3 Install plpgsql

Note: You can ensure that any database created on your system automatically gets these spatial extensions by running these commands (from this and the next two sections) on the `template1` system database *before* you create any of your own databases.

PostgreSQL can use various procedural languages. What is a procedural language? It is an 'in database' language that can be used to extend the functionality of the database. For example you can write database functions that are called when events happen - such as when a record is inserted into the database. (Recall when this was done in the previous module.)

PostGIS requires the PLPGSQL procedural language to be installed. So do this:

```
createlang plpgsql address
```

Where the third argument is the name of the database that the procedural language should be installed into.

Note: You will need administrative permissions for your database to be able to do this.

15.1.4 Install postgis.sql

PostGIS can be thought of as a collection of in database functions that extend the core capabilities of PostgreSQL so that it can deal with spatial data. By 'deal with', we mean store, retrieve, query and manipulate. In order to do this, a number of functions are installed into the database. Do this:

```
psql address < /usr/share/postgresql/9.1/contrib/postgis-1.5/postgis.sql
```

or

```
psql address < /usr/share/postgresql/9.1/contrib/postgis-2.0/postgis.sql
```

depending on which PostGIS version you have installed. Now do:

```
psql address
```

and, once you're in the psql prompt:

```
\df
```

We will discuss these functions in more detail as we proceed with this course.

[5] http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgis_tut01

15.1.5 Install spatial_refsys.sql

In addition to the PostGIS functions, a second helper SQL script needs to be run that will load the database with a collection of spatial reference system (SRS) definitions as defined by the European Petroleum Survey Group (EPSG). These are used during operations such as coordinate reference system (CRS) conversions.

You can add more to the SRS list later if needed, but the list provided should cover just about every SRS you will need (Google Mercator and lo are notable exceptions).

To load the SRS table, first ensure that you're at a normal prompt (i.e., quit the database first with `q`), then do this:

```
psql address < /usr/share/postgresql/9.1/contrib/postgis-1.5/spatial_ref_sys.sql
```

replacing `1.5` with `2.0` if necessary.

The above command adds a table to our database. We can see the schema of this table by entering the following command in the psql prompt:

```
address=# \d spatial_ref_sys
```

The result should be this:

```
Table "public.spatial_ref_sys"
   Column   |          Type           | Modifiers
 -----------+-------------------------+-----------
  srid      | integer                 | not null
  auth_name | character varying(256)  |
  auth_srid | integer                 |
  srtext    | character varying(2048) |
  proj4text | character varying(2048) |
  Indexes:
"spatial_ref_sys_pkey" PRIMARY KEY, btree (srid)
```

You can use standard SQL queries (as we have learned from our introductory sections), to view and manipulate this table - though we suggest you do not update or delete any records unless you know what you are doing.

One SRID you may be interested in is EPSG:4326 - the geographic / lat lon reference system using the WGS 84 ellipsoid. Let's take a look at it:

```
select * from spatial_ref_sys where srid=4326;
```

Result

```
srid      | 4326
auth_name | EPSG
auth_srid | 4326
srtext    | GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS
84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],TOWGS84[0,
0,0,0,0,0,0],AUTHORITY["EPSG","6326"]],PRIMEM["Greenwich",0,
AUTHORITY["EPSG","8901"]],UNIT["degree",0.01745329251994328,
```

```
AUTHORITY["EPSG","9122"]],AUTHORITY["EPSG","4326"]]
proj4text | +proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
```

The srtext is the projection definition in well known text (you may recognise this from .prj files in your shapefile collection).

15.1.6 Looking at the installed PostGIS functions

Good - our PostgreSQL database is now geospatially enabled, thanks to PostGIS. We are going to delve a lot deeper into this in the coming days, but let's give you a quick little taster. Let's say we want to create a point from text. First we use the psql command to find functions relating to point:

```
\df *point*
```

Here is one that caught my eye: `st_pointfromtext`

So let's give that a try:

```
address=# select st_pointfromtext('POINT(1 1)');
```

Result:

```
st_pointfromtext
----------------------------------------------
0101000000000000000000F03F000000000000F03F
(1 row)
```

So there are a couple of interesting things going on here:

- we defined a point at position 1,1 (EPSG:4326 is assumed) using `POINT(1 1)`
- we ran an sql statement, but not on any table, just on data entered from the SQL prompt
- the resulting row looks kinda strange

The resulting row is looking strange because its in the OGC format called 'Well Known Binary' (WKB) - more on that coming in the next section.

To get the results back as text, I do a quick scan through the function list for something that returns text:

```
\df *text
```

One that catches my eye is `st_astext`. Let's combine it with the previous query:

```
select st_astext(st_pointfromtext('POINT(1 1)'));
```

Result:

```
 st_astext
------------
  POINT(1 1)
  (1 row)
```

So what's happened here? We entered the string `POINT(1,1)`, turned it into a point using `st_pointfromtext()`, and turned it back into a human-readable form with `st_astext()`, which gave us back our original string.

One last example before we really get into the detail of using PostGIS:

```
select st_astext(st_buffer(st_pointfromtext('POINT(1 1)'),1.0));
```

What did that do? It created a buffer of 1 degree around our point, and returned it as text. Nifty hey?

15.1.7 In conclusion

You now have PostGIS functions installed in your copy of PostgreSQL. With this you'll be able to make use of PostGIS' extensive spatial functions.

15.1.8 What's next?

Next you'll learn how spatial features are represented in a database.

15.2 Lesson: Simple Feature Model

How can we store and represent geographic features in a database? In this lesson we'll cover one approach, the Simple Feature Model as defined by the OGC.

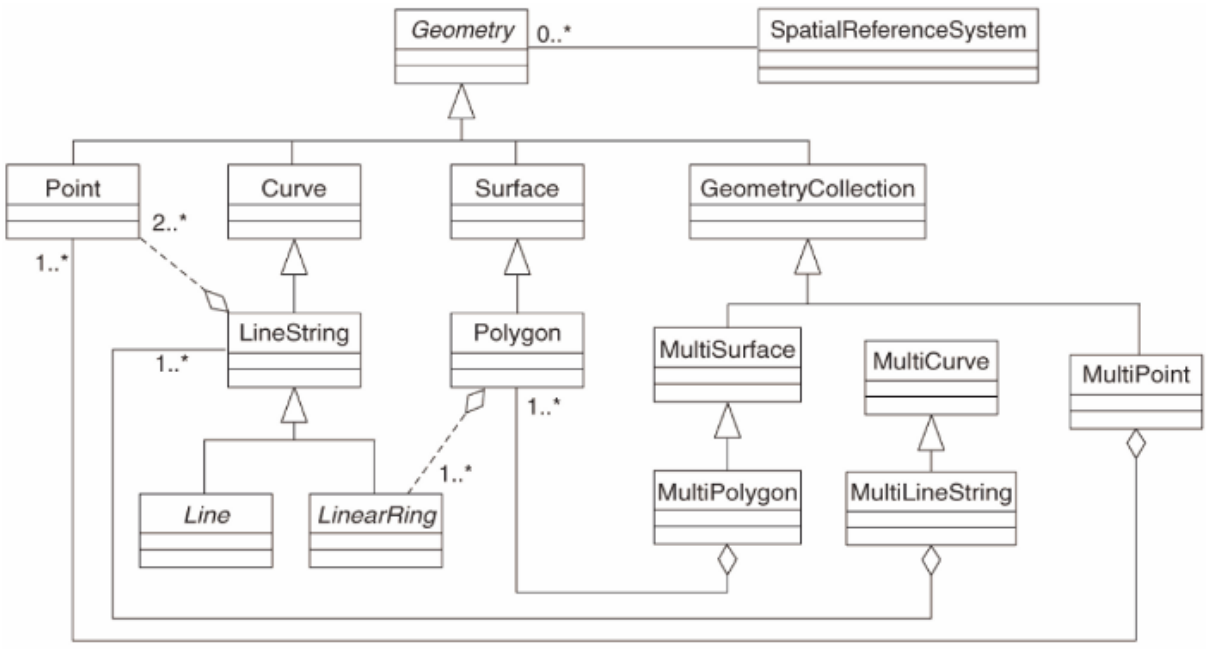
The goal for this lesson: To learn what the SFS Model is and how to use it.

15.2.1 What is OGC

> The Open Geospatial Consortium (OGC), an international voluntary consensus standards organization, originated in 1994. In the OGC, more than 370+ commercial, governmental, nonprofit and research organizations worldwide collaborate in an open consensus process encouraging development and implementation of standards for geospatial content and services, GIS data processing and data sharing. - *Wikipedia*

15.2.2 What is the SFS Model

The Simple Feature for SQL (SFS) Model is a *non-topological* way to store geospatial data in a database and defines functions for accessing, operating, and constructing these data.



The model defines geospatial data from Point, Linestring, and Polygon types (and aggregations of them to Multi objects).

For further information, have a look at the OGC Simple Feature for SQL[6] standard.

15.2.3 Add a geometry field to table

Let's add a point field to our people table:

```
alter table people add column the_geom geometry;
```

15.2.4 Add a constraint based on geometry type

You will notice that the geometry field type does not implicitly specify what *type* of geometry for the field - for that we need a constraint.

```
alter table people
add constraint people_geom_point_chk
    check(st_geometrytype(the_geom) = 'ST_Point'::text OR the_geom IS NULL);
```

What does that do? It adds a constraint to the table that prevents anything except a point geometry or a null.

Now you try:

> Create a new table called cities and give it some appropriate columns, including a geometry field for storing polygons (the city boundaries). Make sure it has a constraint enforcing geometries to be polygons.

[6] http://www.opengeospatial.org/standards/sfs

Check your results

15.2.5 Populate geometry_columns table

At this point you should also add an entry into the `geometry_columns` table:

```
insert into geometry_columns values
  ('','public','people','the_geom',2,4326,'POINT');
```

Why? `geometry_columns` is used by certain applications to be aware of which tables in the database contain geometry data.

Note: If the above `INSERT` statement causes a complaint, run this query first:

```
select * from geometry_columns;
```

If the column `f_table_name` contains the value `people`, then this table has already been registered and you don't need to do anything more.

The value `2` refers to the number of dimensions; in this case, two: **x** and **y**.

The value `4326` refers to the projection we are using; in this case, WGS 84, which is referred to by the number 4326 (refer to the earlier discussion about the EPSG).

Add an appropriate geometry_columns entry for your new cities layer

Check your results

15.2.6 Add geometry record to table using SQL

Now that our tables are geo-enabled, we can store geometries in them!

```
insert into people (name,house_no, street_id, phone_no, the_geom)
        values ('Fault Towers',
                34,
                3,
                '072 812 31 28',
                'SRID=4326;POINT(33 -33)');
```

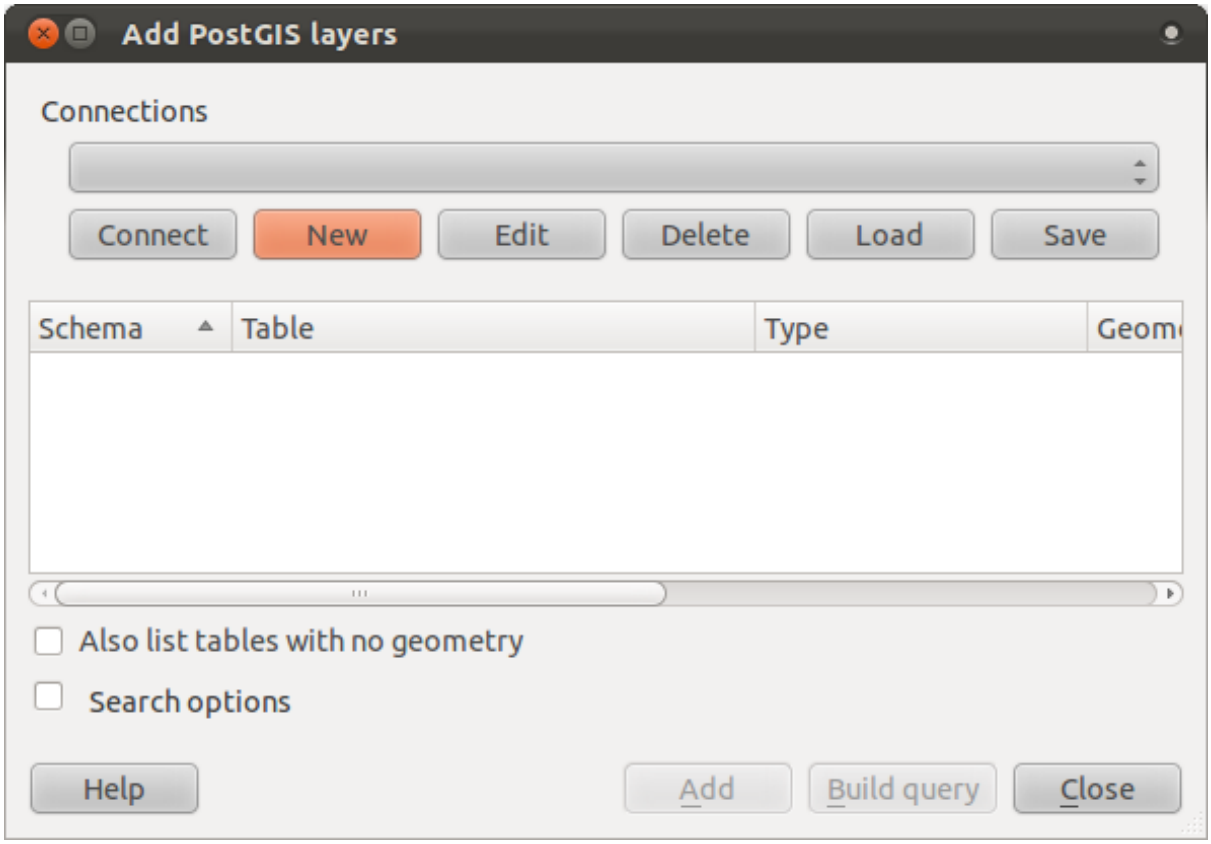
Note: In the new entry above, you will need to specify which projection (SRID) you want to use. This is because you entered the geometry of the new point using a plain string of text, which does not automatically add the correct projection information. Obviously, the new point needs to use the same SRID as the dataset it is being added to, so you need to specify it.

If at this point you were using a graphical interface, for example, specifying the projection for each point would be automatic. In other words, you usually won't need to worry about using the correct projection for every point you want to add if you've already specified it for that dataset, as we did earlier.

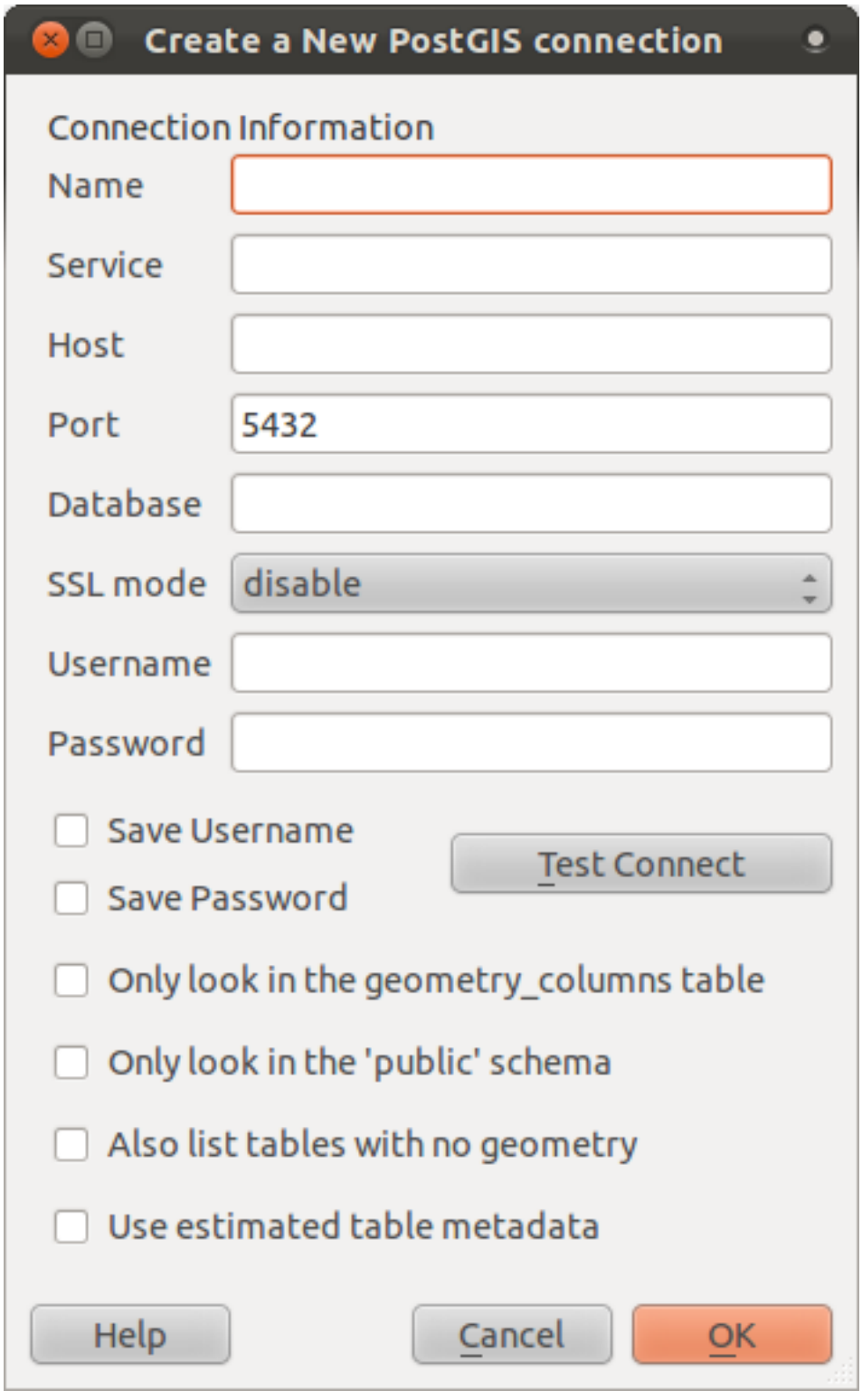
Now is probably a good time to fire up QGIS and try to view your `people` table. Also, we should try editing / adding / deleting records and then performing select queries in the database to see how the data has changed.

To load a PostGIS layer in QGIS, use the *Layer* → *Add PostGIS Layers* menu option or toolbar button:

This will open the dialog:



Click on the *New* button to open this dialog:



Then define a new connection, e.g.:

```
Name: myPG
Service:
Host: localhost
Port: 5432
Database: address
User:
```

```
Password:
```

To see whether QGIS has found the `address` database and that your username and password are correct, click *Test Connect*. If it works, check the boxes next to *Save Username* and *Save Password*. Then click *OK* to create this connection.

Back in the *Add PostGIS Layers* dialog, click *Connect* and add layers to your project as usual.

> Formulate a query that shows a person's name, street name and position (from the the_geom column) as plain text.

Check your results

15.2.7 In conclusion

You have seen how to add spatial objects to your database and view them in GIS software.

15.2.8 What's next?

Next you'll see how to import data into, and export data from, your database.

15.3 Lesson: Import and Export

Of course, a database with no easy way to migrate data into it and out of it would be no fun. Even more so for spatial data! Fortunately, there are a number of tools that will let you easily move data into and out of PostGIS.

15.3.1 shp2pgsql

shp2pgsql is a commandline tool to import ESRI shapefiles to the database. Under Unix, you can use the following command for importing a new PostGIS table:

```
shp2pgsql -s <SRID> -c -D -I <path to shapefile> <schema>.<table> | \
  psql -d <databasename> -h <hostname> -U <username>
```

Under Windows, you have to perform the import process in two steps:

```
shp2pgsql -s <SRID> -c -D -I <path to shapefile> <schema>.<table> > import.sql
psql psql -d <databasename> -h <hostname> -U <username> -f import.sql
```

You may encounter this error:

```
ERROR:  operator class "gist_geometry_ops" does not exist for access method
"gist"
```

This is a known issue regarding the creation *in situ* of a spatial index for the data you're importing. To avoid the error, exclude the `-I` parameter. This will mean that no spatial index is being created directly, and you'll need to create it in the database after the data have been imported. (The creation of a spatial index will be covered in the next lesson.)

15.3.2 pgsql2shp

pgsql2shp is a commandline tool to export PostGIS Tables, Views or SQL select queries. To do this under Unix:

```
pgsql2shp -f <path to new shapefile> -g <geometry column name> \
  -h <hostname> -U <username> <databasename> <table | view>
```

To export the data using a query:

```
pgsql2shp -f <path to new shapefile> -g <geometry column name> \
  -h <hostname> -U <username> "<query>"
```

15.3.3 ogr2ogr

ogr2ogr is a very powerful tool to convert data into and from postgis to many data formats. ogr2ogr is part of the GDAL/OGR Software and has to be installed separately. To export a table from PostGIS to GML, you can use this command:

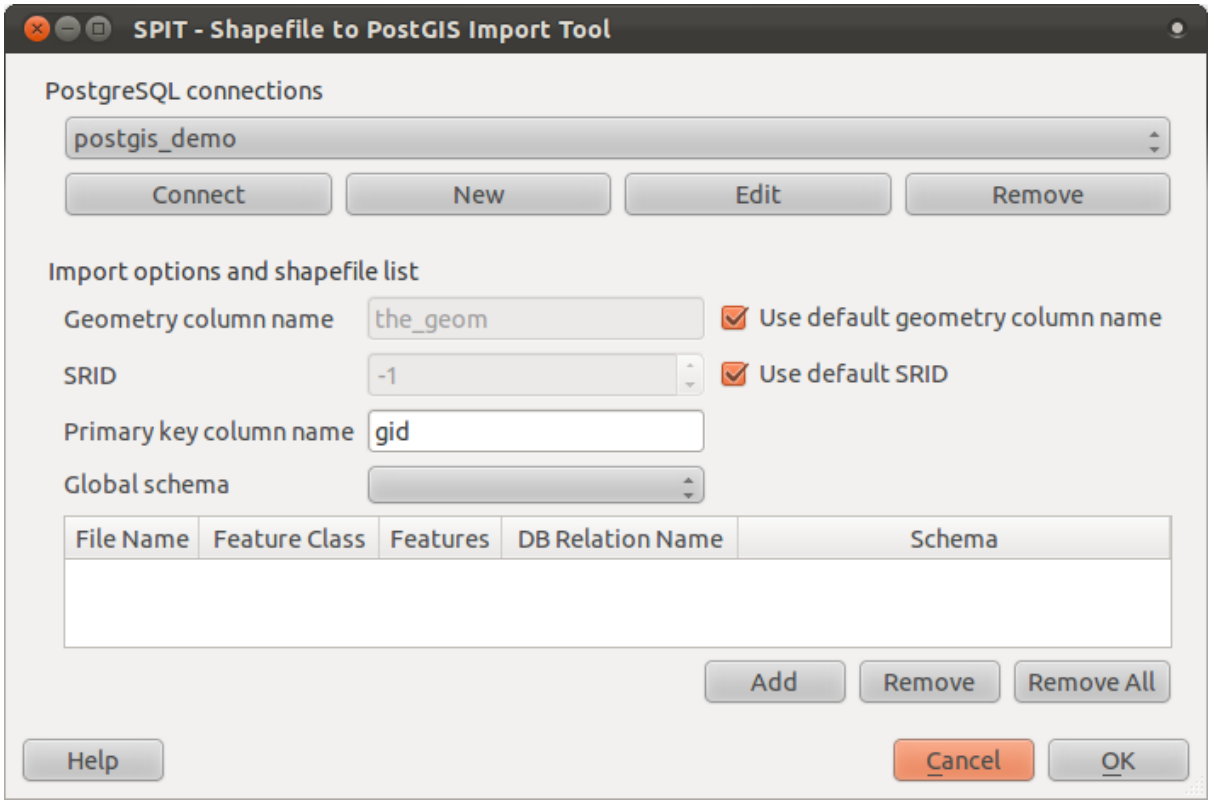
```
ogr2ogr -f GML export.gml PG:'dbname=<databasename> user=<username>
        host=<hostname>' <Name of PostGIS-Table>
```

15.3.4 SPIT

SPIT is a QGIS plugin which is delivered with QGIS. You can use SPIT for uploading ESRI shapefiles to PostGIS.

Once you've added the SPIT plugin via the *Plugin Manager*, look for this button:

Clicking on it will give you the SPIT dialog:



You can add shapefiles to the database by clicking the *Add* button, which will give you a file browser window.

15.3.5 In conclusion

Importing and exporting data to and from the database can be done in many various ways. Especially when using disparate data sources, you will probably use these functions (or others like them) on a regular basis.

15.3.6 What's next?

Next we'll look at how to query the data we've created before.

15.4 Lesson: Spatial Queries

Spatial queries are no different from other database queries. You can use the geometry column like any other database column. With the installation of PostGIS in our database, we have additional functions to query our database.

The goal for this lesson: To see how spatial functions are implemented similarly to "normal" non-spatial functions.

15.4.1 Spatial Operators

When you want to know which points are within a distance of 2 degrees to a point(X,Y) you can do this with:

```
select *
from people
where st_distance(the_geom,'SRID=4326;POINT(33 -34)') < 2;
```

Result:

```
 id |     name      | house_no | street_id |   phone_no     |   the_geom
----+---------------+----------+-----------+----------------+------------------
  6 | Fault Towers |       34 |         3 | 072 812 31 28 | 01010008040C0
(1 row)
```

Note: the_geom value above was truncated for space on this page. If you want to see the point in human-readable coordinates, try something similar to what you did in the section "View a point as WKT", above.

How do we know that the query above returns all the points within 2 *degrees*? Why not 2 *meters*? Or any other unit, for that matter?

Check your results

15.4.2 Spatial Indexes

We also can define spatial indexes. A spatial index makes your spatial queries much faster. To create a spatial index on the geometry column use:

```
CREATE INDEX people_geo_idx
  ON people
  USING gist
  (the_geom);
```

Result:

```
address=# \d people
Table "public.people"
   Column   |         Type          |                Modifiers
 -----------+-----------------------+-----------------------------------------
  id        | integer               | not null default
            |                       | nextval('people_id_seq'::regclass)
  name      | character varying(50) |
  house_no  | integer               | not null
  street_id | integer               | not null
  phone_no  | character varying     |
  the_geom  | geometry              |
Indexes:
  "people_pkey" PRIMARY KEY, btree (id)
  "people_geo_idx" gist (the_geom)  <-- new spatial key added
  "people_name_idx" btree (name)
Check constraints:
  "people_geom_point_chk" CHECK (st_geometrytype(the_geom) = 'ST_Point'::text
  OR the_geom IS NULL)
Foreign-key constraints:
  "people_street_id_fkey" FOREIGN KEY (street_id) REFERENCES streets(id)
```

Now you try - modify the cities table so its geometry column is spatially indexed.

Check your results

15.4.3 PostGIS Spatial Functions Demo

In order to demo PostGIS spatial functions, we'll create a new database containing some (fictional) data.

To start, create a new database:

```
createdb postgis_demo
```

Remember to install PLPGSQL:

```
createlang plpgsql postgis_demo
```

Then install the PostGIS functions and the spatial reference system. For example, on Linux with PostgreSQL 9.1 and PostGIS 1.5:

```
psql postgis_demo < /usr/share/postgresql/9.1/contrib/postgis-1.5/postgis.sql
psql postgis_demo < /usr/share/postgresql/9.1/contrib/postgis-1.5/spatial_ref_sy
```

Next, import the data provided in the `exercise_data/postgis/` directory. Refer back to the previous lesson for instructions. You can import from the terminal or via SPIT. Import the files into the following database tables:

- `points.shp` = `building`
- `lines.shp` = `road`
- `polygons.shp` = `region`

Load these three database layers into QGIS via the *Add PostGIS Layers* dialog, as usual. When you open their attribute tables, you'll note that they have both an `id` field and a `gid` field created by the PostGIS import.

Now that the tables are imported, we can use PostGIS to query the data. Go back to your terminal (command line) and enter the psql prompt by doing:

```
psql postgis_demo
```

We'll demo some of these select statements by creating views from them, so that you can open them in QGIS and see the results.

Select by location

Get all the buildings in the KwaZulu region.

```
SELECT a.id, a.name, st_astext(a.the_geom) as point
  FROM building a, region b
    WHERE WITHIN(a.the_geom, b.the_geom)
    AND b.name = 'KwaZulu';
```

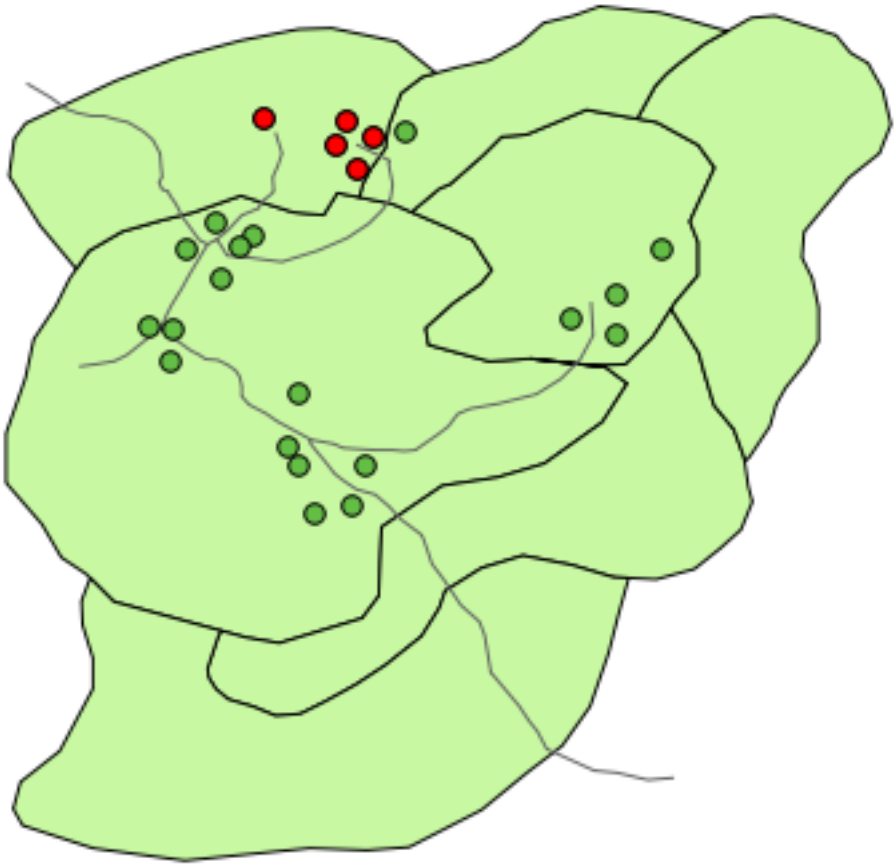
Result:

```
 id | name |                   point
----+------+-------------------------------------------
 30 | York | POINT(1622345.23785063 6940490.65844485)
 33 | York | POINT(1622495.65620524 6940403.87862489)
 35 | York | POINT(1622403.09106394 6940212.96302097)
 36 | York | POINT(1622287.38463732 6940357.59605424)
 40 | York | POINT(1621888.19746548 6940508.01440885)
(5 rows)
```

Or, if we create a view from it:

```
CREATE VIEW vw_select_location AS
  SELECT a.gid, a.name, a.the_geom
    FROM building a, region b
      WHERE WITHIN(a.the_geom, b.the_geom)
      AND b.name = 'KwaZulu';
```

And view it in QGIS:



Select neighbors

Show a list of all the names of regions adjoining the Hokkaido region.

```
SELECT b.name
  FROM region a, region b
    WHERE TOUCHES(a.the_geom, b.the_geom)
    AND a.name = 'Hokkaido';
```

Result:

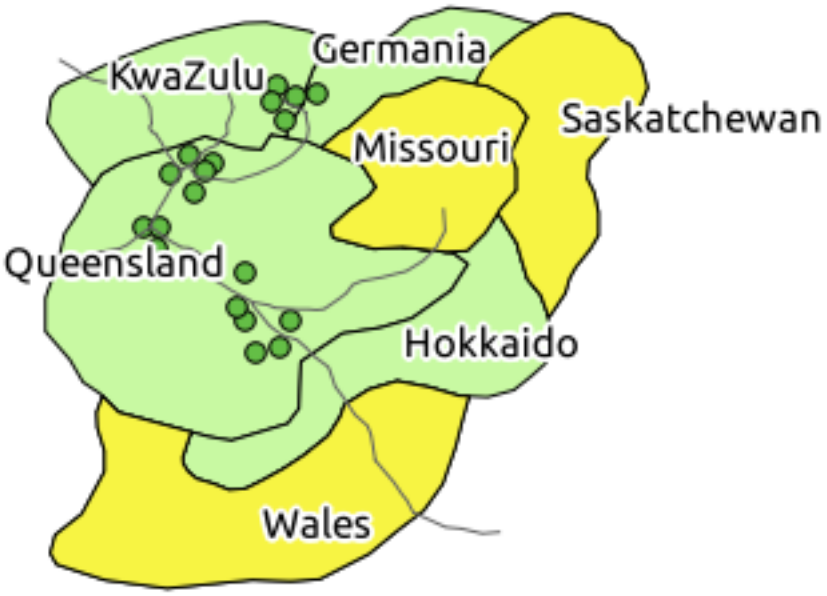
```
    name
-------------
 Missouri
 Saskatchewan
 Wales
(3 rows)
```

As a view:

```
CREATE VIEW vw_regions_adjoining_hokkaido AS
  SELECT b.gid, b.name, b.the_geom
    FROM region a, region b
```

```
      WHERE TOUCHES(a.the_geom, b.the_geom)
      AND a.name = 'Hokkaido';
```

In QGIS:

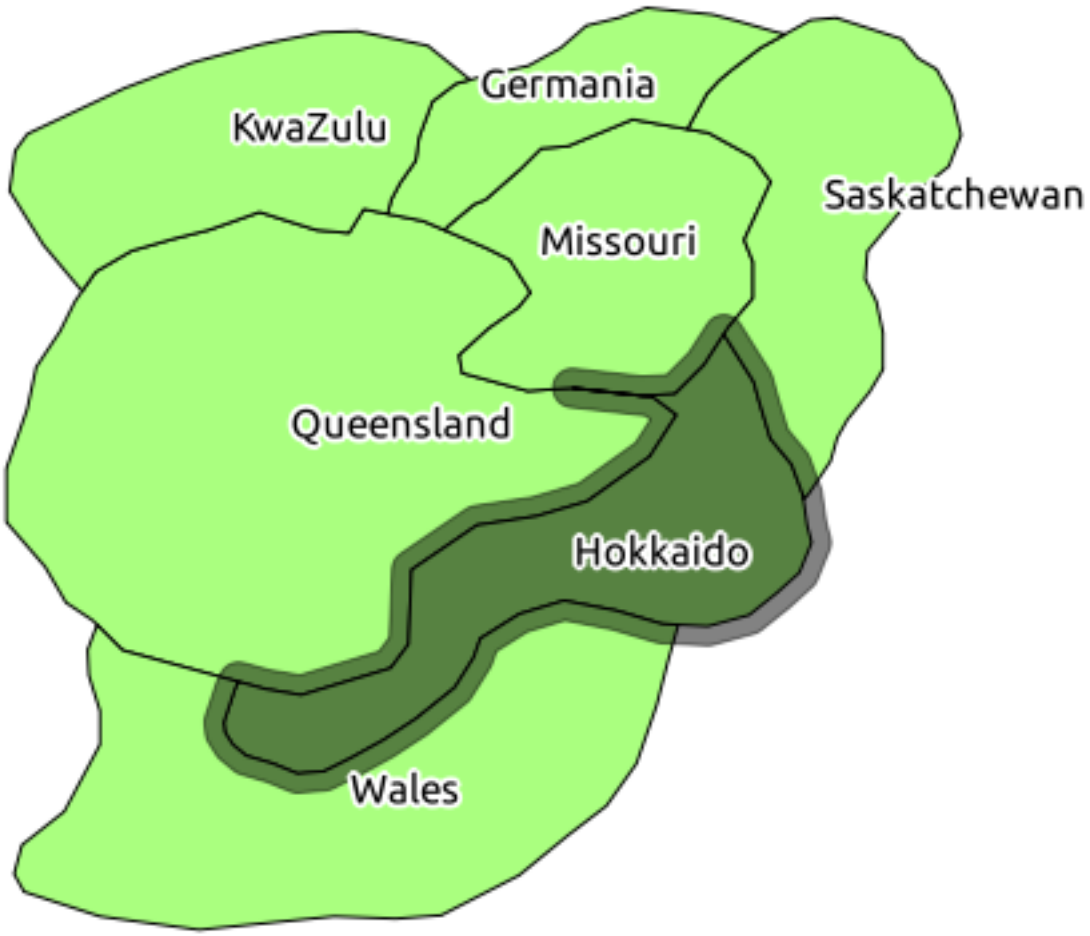


Note the missing region (Queensland). This may be due to a topology error. Artifacts such as this can alert us to potential problems in the data. To solve this enigma without getting caught up in the anomalies the data may have, we could use a buffer intersect instead:

```
CREATE VIEW vw_hokkaido_buffer AS
  SELECT gid, ST_BUFFER(the_geom, 100) as the_geom
    FROM region
      WHERE name = 'Hokkaido';
```

This creates a buffer of 100 meters around the region Hokkaido.

The darker area is the buffer:

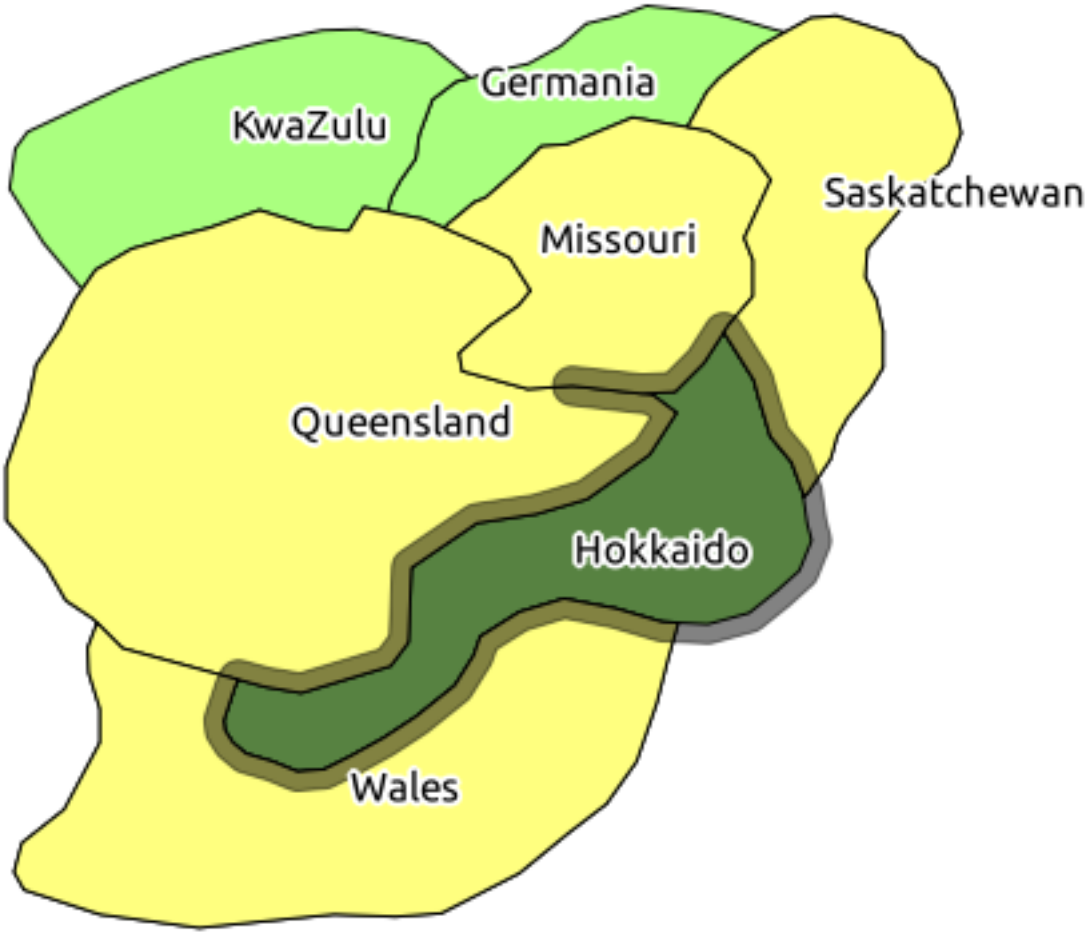


Select using the buffer:

```
CREATE VIEW vw_hokkaido_buffer_select AS
  SELECT b.gid, b.name, b.the_geom
    FROM
    (
      SELECT * FROM
        vw_hokkaido_buffer
    ) a,
    region b
    WHERE ST_INTERSECTS(a.the_geom, b.the_geom)
    AND b.name != 'Hokkaido';
```

In this query, the original buffer view is used as any other table would be. It is given the alias `a`, and its geometry field, `a.the_geom`, is used to select any polygon in the `region` table (alias `b`) that intersects it. However, Hokkaido itself is excluded from this select statement, because we don't want it; we only want the regions adjoining it.

In QGIS:



It is also possible to select all objects within a given distance, without the extra step of creating a buffer:

```
CREATE VIEW vw_hokkaido_distance_select AS
  SELECT b.gid, b.name, b.the_geom
    FROM region a, region b
      WHERE ST_DISTANCE (a.the_geom, b.the_geom) < 100
      AND a.name = 'Hokkaido'
      AND b.name != 'Hokkaido';
```

This achieves the same result, without need for the interim buffer step:

Select uniques

Show a list of unique town names for all buildings in the Queensland region.

```
SELECT DISTINCT a.name
  FROM building a, region b
    WHERE WITHIN (a.the_geom, b.the_geom)
    AND b.name = 'Queensland';
```

Result:

```
  name
---------
 Beijing
 Berlin
 Atlanta
(3 rows)
```

Further examples ...

```
CREATE VIEW vw_shortestline AS
  SELECT b.gid AS gid, ST_ASTEXT(ST_SHORTESTLINE(a.the_geom, b.the_geom)) as
    text, ST_SHORTESTLINE(a.the_geom, b.the_geom) AS the_geom
    FROM road a, building b
      WHERE a.id=5 AND b.id=22;

CREATE VIEW vw_longestline AS
  SELECT b.gid AS gid, ST_ASTEXT(ST_LONGESTLINE(a.the_geom, b.the_geom)) as
    text, ST_LONGESTLINE(a.the_geom, b.the_geom) AS the_geom
    FROM road a, building b
      WHERE a.id=5 AND b.id=22;
```

```
CREATE VIEW vw_road_centroid AS
  SELECT a.gid as gid, ST_CENTROID(a.the_geom) as the_geom
    FROM road a
      WHERE a.id = 1;

CREATE VIEW vw_region_centroid AS
  SELECT a.gid as gid, ST_CENTROID(a.the_geom) as the_geom
    FROM region a
      WHERE a.name = 'Saskatchewan';

SELECT ST_PERIMETER(a.the_geom)
  FROM region a
    WHERE a.name='Queensland';

SELECT ST_AREA(a.the_geom)
  FROM region a
    WHERE a.name='Queensland';

CREATE VIEW vw_simplify AS
  SELECT gid, ST_Simplify(the_geom, 20) AS the_geom
    FROM road;

CREATE VIEW vw_simplify_more AS
  SELECT gid, ST_Simplify(the_geom, 50) AS the_geom
    FROM road;

CREATE VIEW vw_convex_hull AS
  SELECT
    ROW_NUMBER() over (order by a.name) as id,
    a.name as town,
    ST_CONVEXHULL(ST_COLLECT(a.the_geom)) AS the_geom
    FROM building a
    GROUP BY a.name;
```

15.4.4 In conclusion

You have seen how to query spatial objects using the new database functions from PostGIS.

15.4.5 What's next?

Next we're going to investigate the structures of more complex geometries and how to create them using PostGIS.

15.5 Lesson: Geometry Construction

In this section we are going to delve a little deeper into how simple geometries are constructed in SQL. In reality, you will probably use a GIS like QGIS to create complex geometries using their digitising tools; however, understanding how they are formulated can be handy for writing queries and understanding how the database is assembled.

The goal of this lesson: To better understand how to create spatial entities directly in PostgreSQL/PostGIS.

15.5.1 Creating Linestrings

Before we start, let's get our streets table matching the others; i.e., having a constraint on the geometry, an index and an entry in the geometry_columns table.

Exercise:

- Modify the streets table so that it has a geometry column of type ST_LineString.
- Don't forget to do the accompanying update to the geometry columns table!
- Also add a constraint to prevent any geometries being added that are not LINESTRINGS or null.
- Create a spatial index on the new geometry column

Check your results

Now let's insert a linestring into our streets table. In this case I am going to update an existing street record:

```
update streets set the_geom = 'SRID=4326;LINESTRING(20 -33, 21 -34, 24 -33)'
where streets.id=2;
```

Take a look at the results in QGIS. (You may need to right-click on the streets layer in the 'Layers' panel, and choose 'Zoom to layer extent'.)

Now create some more streets entries - some in QGIS and some from the command line.

15.5.2 Creating Polygons

Creating polygons is just as easy. One thing to remember is that by definition, polygons have at least four vertices, with the last and first being co-located.

```
insert into cities (name, the_geom)
values ('Tokyo', 'SRID=4326;POLYGON((10 -10, 5 -32, 30 -27, 10 -10))');
```

Note: A polygon requires double brackets around its coordinate list; this is to allow you to add complex polygons with multiple unconnected areas. For instance:

```
insert into cities (name, the_geom)
values ('Tokyo Outer Wards', 'SRID=4326;POLYGON((20 10, 20 20, 35 20, 20 10),
        (-10 -30, -5 0, -15 -15, -10 -30))');
```

If you followed this step, you can check what it did by loading the cities dataset into QGIS, opening its attribute table, and selecting the new entry. Note how the two new polygons behave like one polygon.

15.5.3 Exercise: Linking Cities to People

For this exercise you should do the following:

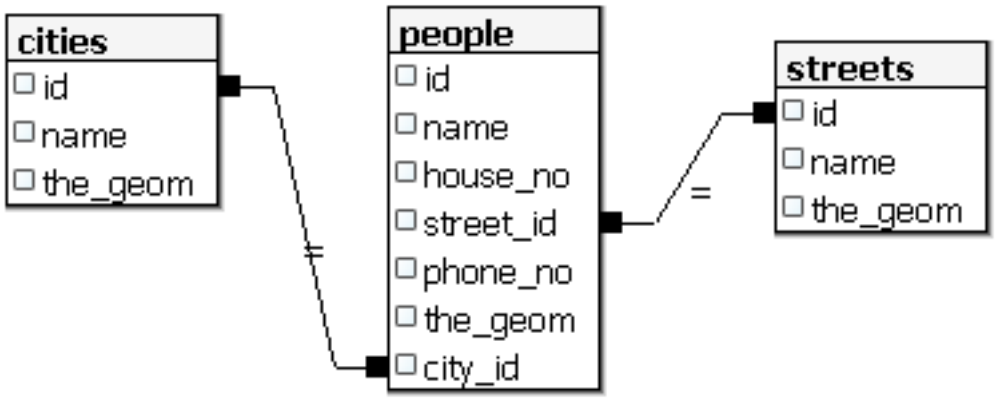
> Delete all data from your people table. Add a foreign key column to people that references the primary key of the cities table. Use QGIS to capture some cities. Use SQL to insert some new people records, ensuring that each has an associated street and city.
>
> Your updated people schema should look something like this:

```
\d people
Table "public.people"
   Column   |         Type          |                       Modifiers
 -----------+-----------------------+-------------------------------------
  id        | integer               | not null
            |                       | default nextval('people_id_seq'::regcl
  name      | character varying(50) |
  house_no  | integer               | not null
  street_id | integer               | not null
  phone_no  | character varying     |
  the_geom  | geometry              |
  city_id   | integer               | not null
Indexes:
  "people_pkey" PRIMARY KEY, btree (id)
  "people_name_idx" btree (name)
Check constraints:
  "people_geom_point_chk" CHECK (st_geometrytype(the_geom) =
                      'ST_Point'::text OR the_geom IS NULL)
Foreign-key constraints:
  "people_city_id_fkey" FOREIGN KEY (city_id) REFERENCES cities(id)
  "people_street_id_fkey" FOREIGN KEY (street_id) REFERENCES streets(id)
```

Check your results

15.5.4 Looking at our schema

By now our schema should be looking like this:



15.5.5 Access Subobjects

With the SFS-Model functions, you have a wide variety of options to access subobjects of SFS Geometries. When you want to select the first vertex point of every polygon geometry in the table myPolygonTable, you have to do this in this way:

- Transform the polygon boundary to a linestring:

```
select st_boundary(geometry) from myPolygonTable;
```

- select the first vertex point of the resultant linestring:

```
select st_startpoint(myGeometry)
from (
  select st_boundary(geometry) as myGeometry
  from myPolygonTable) as foo;
```

15.5.6 Data Processing

PostGIS supports all OGC SFS/MM standard conform functions. All these functions start with `ST_`.

15.5.7 Clipping

To clip a subpart of your data you can use the `ST_INTERSECT()` function. To avoid empty geometries, use:

```
where not st_isempty(st_intersection(a.the_geom, b.the_geom))
```

```
select st_intersection(a.the_geom, b.the_geom), b.*
from clip as a, road_lines as b
where not st_isempty(st_intersection(st_setsrid(a.the_geom,32734),
  b.the_geom));
```

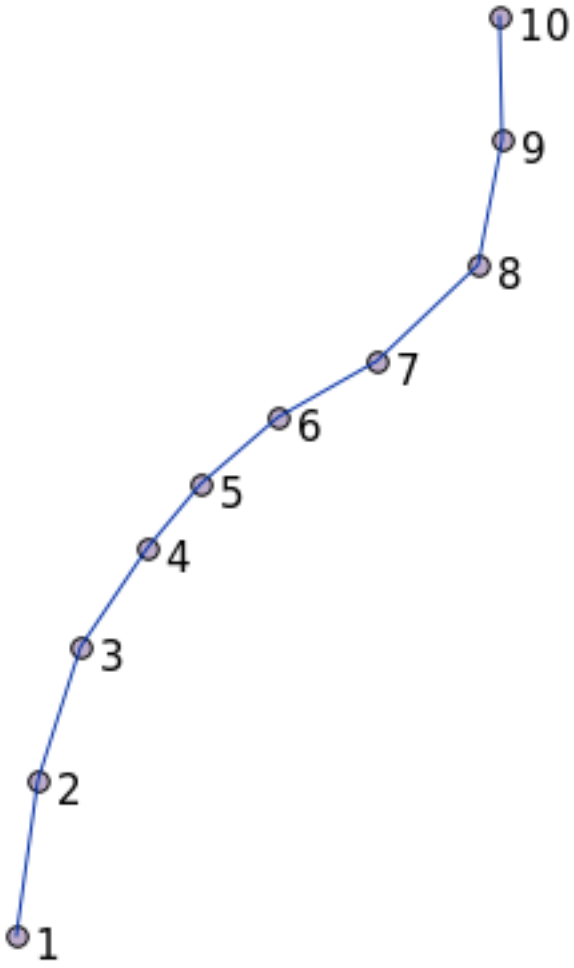
15.5.8 Building Geometries from Other Geometries

From a given point table, you want to generate a linestring. The order of the points is defined by their `id`. Another ordering method could be a timestamp, such as the one you get when you capture waypoints with a GPS receiver.

To create a linestring from a new point layer called 'points', you can run the following command:

```
select ST_LineFromMultiPoint(st_collect(the_geom)), 1 as id
from (
  select the_geom
  from points
  order by id
) as foo;
```

To see how it works without creating a new layer, you could also run this command on the 'people' layer, although of course it would make little real-world sense to do this.



15.5.9 Geometry Cleaning

You can get more information for this topic in this blog entry[7].

15.5.10 Differences between tables

To detect the difference between two tables with the same structure, you can use the PostgreSQL keyword `EXCEPT`.

```
select * from table_a
except
select * from table_b;
```

As the result, you will get all records from table_a which are not stored in table_b.

[7] http://linfiniti.com/?s=cleangeometry

15.5.11 Tablespaces

You can define where postgres should store its data on disk by creating tablespaces.

```
CREATE TABLESPACE homespace LOCATION '/home/pg';
```

When you create a database, you can then specify which tablespace to use e.g.:

```
createdb --tablespace=homespace t4a
```

15.5.12 In conclusion

You've learned how to create more complex geometries using PostGIS statements. Keep in mind that this is mostly to improve your tacit knowledge when working with geo-enabled databases through a GIS frontend. You usually won't need to actually enter these statements manually, but having a general idea of their structure will help you when using a GIS, especially if you encounter errors that would otherwise seem cryptic.