Lab 3 Spatial Database: PostGIS

In this lab, we will use PostgreSQL/PostGIS to create and manipulate spatial database. PostGIS is an open source, freely available, and fairly OGC compliant spatial database extender for the PostgreSQL Database Management System. In a nutshell it adds spatial functions such as distance, area, union, intersection, and specialty geometry data types to the database. PostGIS is very similar in functionality to SQL Server 2008 Spatial support, ESRI ArcSDE, Oracle Spatial, and DB2 spatial extender. This lab is designed and developed based on the OpenGeo tutorial on PostGIS[1].

Objectives

The goals for you to take away from this lab are:

- Familiarize yourself with the PostGIS to create a spatial database
- Learn how to load GIS layers into a spatial database
- Understand and practice spatial query

Lab Data

Please download the data (data.zip) in the Lab section at Learn@UW system. If you are using Windows, put it under C:/lab/ (if you do not have a C:/lab fold, please create one). Unzip the data. This file includes New York census, street, metro station, and neighborhoods shapefiles that can be imported to the database later.

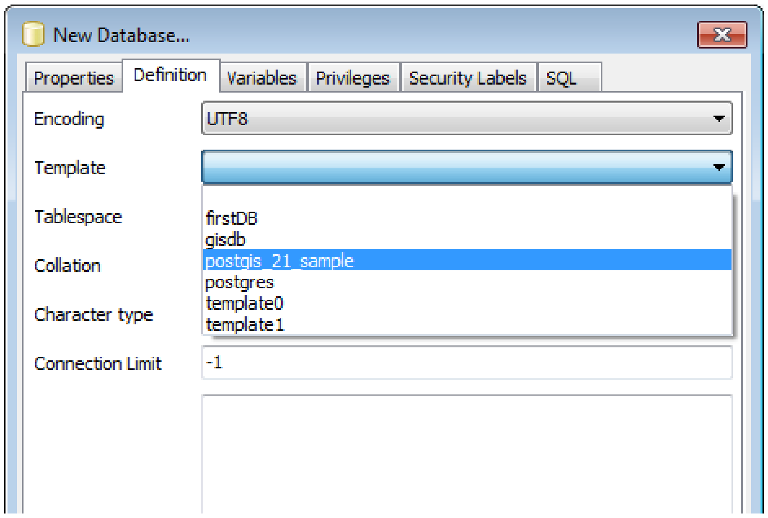
1. Creating a spatial database

Let's first use the PostgreSQL admin tool, PgAdmin III to create a new spatial database.

➔ Click Start->Programs->PostgreSQL 9.3->PgAdmin III to start PgAdmin III

Login with the super user postgres and the password you chose during install. Now for the fun part.

➔ Create your database. Call it ***nyc*** or whatever you want. If you are running PostgreSQL 9.3, choose the template database called **postgis_21_sample**.



Encoding: UTF8
Template:
firstDB
gisdb
postgis_21_sample
postgres
template0
template1
Connection Limit: -1

[1] http://workshops.boundlessgeo.com/postgis-intro/index.html

2. Loading GIS Data Into the Database

Now we have a nice fully functional GIS database with no spatial data. Now it is time to populate this database with various GIS database layers to play with.

2.1 Figure out SRID of the data

Go to the data folder where you put our lab data, you will notice one of the files it extracts is called ***nyc_census_blocks.prj***. A *.prj* is often included with ESRI shape files and tells you the projection of the data. We'll need to match this descriptive projection to an SRID (the id field of a spatial ref record in the *spatial_ref_sys* table) if we ever want to reproject our data.

Note: SRID and spatial_ref_sys

PostGIS database has a ***spatial_ref_sys*** table which is the SRID lookup table. An "SRID" stands for "Spatial Reference IDentifier." It defines all the parameters of our data's geographic coordinate system and projection. An SRID is convenient because it packs all the information about a map projection (which can be quite complex) into a single number. The PostGIS ***spatial_ref_sys*** table is an OGC-standard table that defines all the spatial reference systems known to the database. The data shipped with PostGIS, lists over 3000 known spatial reference systems and details needed to transform/re-project between them.

Open up the *.prj* file in a text editor. You'll see something like:

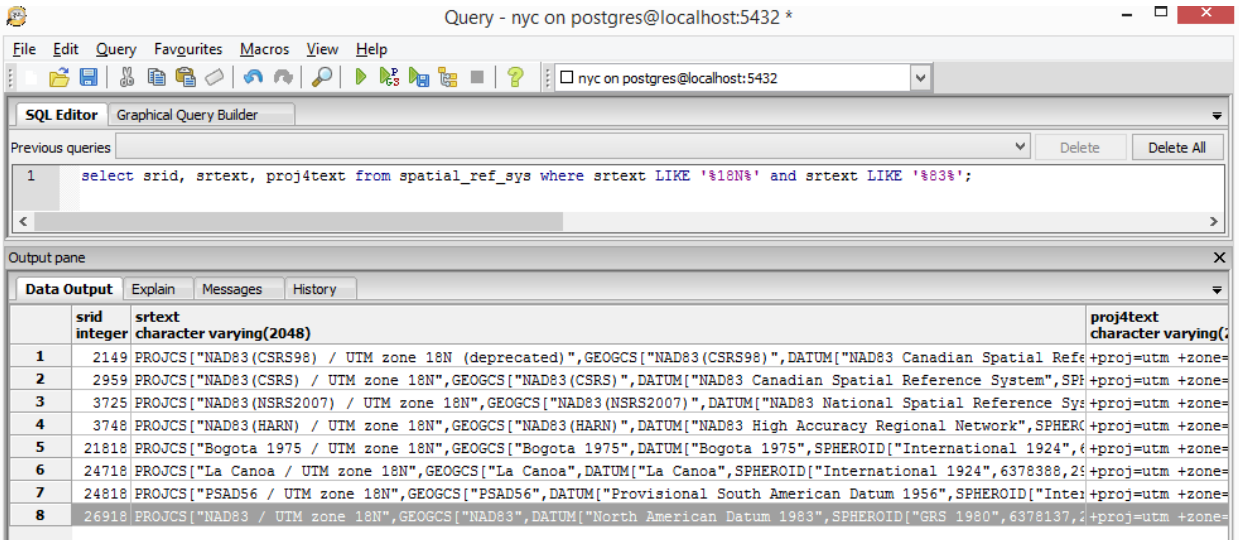
```
PROJCS["NAD83 / UTM zone 18N",
GEOGCS["NAD83",DATUM["North_American_Datum_1983",SPHEROID["GRS
1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],TOWGS84[0,0,0,0,0,0,0],A
UTHORITY["EPSG","6269"]],PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["
degree",0.0174532925199433,AUTHORITY["EPSG","9122"]],AUTHORITY["EPSG","4269"]]
,UNIT["metre",1,AUTHORITY["EPSG","9001"]],PROJECTION["Transverse_Mercator"],PA
RAMETER["latitude_of_origin",0],PARAMETER["central_meridian",-
75],PARAMETER["scale_factor",0.9996],PARAMETER["false_easting",500000],PARAMET
ER["false_northing",0],AUTHORITY["EPSG","26918"],AXIS["Easting",EAST],AXIS["No
rthing",NORTH]]
```

Based on those file, we can use the *spatial_ref_sys* table to get the SRID for our ***nyc_census_blocks.shp*** layer.

➔ Open up your pgAdmin III query tool and type in the following statement:

select srid, srtext, proj4text from spatial_ref_sys where srtext LIKE '%18N%' and srtext LIKE '%83%';

➔ And then click the green arrow. This will bring up about 10 records.

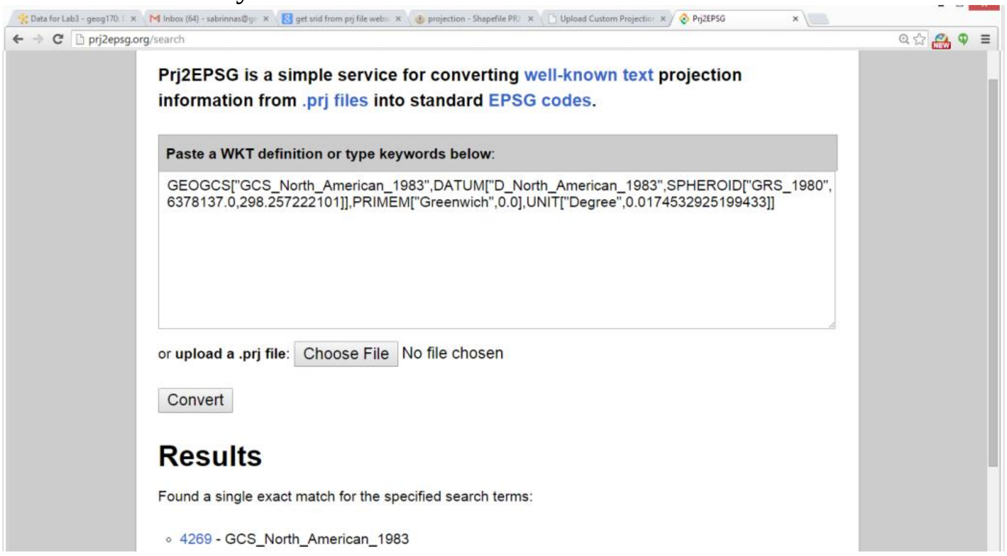


```
select srid, srtext, proj4text from spatial_ref_sys where srtext LIKE '%18N%' and srtext LIKE '%83%';
```

	srid integer	srtext character varying(2048)	proj4text character varying(
1	2149	PROJCS["NAD83(CSRS98) / UTM zone 18N (deprecated)",GEOGCS["NAD83(CSRS98)",DATUM["NAD83 Canadian Spatial Refe	+proj=utm +zone=
2	2959	PROJCS["NAD83(CSRS) / UTM zone 18N",GEOGCS["NAD83(CSRS)",DATUM["NAD83 Canadian Spatial Reference System",SPH	+proj=utm +zone=
3	3725	PROJCS["NAD83(NSRS2007) / UTM zone 18N",GEOGCS["NAD83(NSRS2007)",DATUM["NAD83 National Spatial Reference Sys	+proj=utm +zone=
4	3748	PROJCS["NAD83(HARN) / UTM zone 18N",GEOGCS["NAD83(HARN)",DATUM["NAD83 High Accuracy Regional Network",SPHERC	+proj=utm +zone=
5	21818	PROJCS["Bogota 1975 / UTM zone 18N",GEOGCS["Bogota 1975",DATUM["Bogota 1975",SPHEROID["International 1924",(	+proj=utm +zone=
6	24718	PROJCS["La Canoa / UTM zone 18N",GEOGCS["La Canoa",DATUM["La Canoa",SPHEROID["International 1924",6378388,2(	+proj=utm +zone=
7	24818	PROJCS["PSAD56 / UTM zone 18N",GEOGCS["PSAD56",DATUM["Provisional South American Datum 1956",SPHEROID["Inte	+proj=utm +zone=
8	26918	PROJCS["NAD83 / UTM zone 18N",GEOGCS["NAD83",DATUM["North American Datum 1983",SPHEROID["GRS 1980",6378137,2	+proj=utm +zone=

➔ Note the srid of the closest match. In this case it is **26918**. NOTE: srid is not just a PostGIS term. It is an OGC standard so you will see SRID mentioned a lot in other spatial databases, GIS web services and applications. Most of the common spatial reference systems have globally defined numbers.

A web service to get srid: http://prj2epsg.org/search. From this link, you can upload your .prj file, and it will return the srid for you.



Prj2EPSG is a simple service for converting well-known text projection information from .prj files into standard EPSG codes.

Paste a WKT definition or type keywords below:

GEOGCS["GCS_North_American_1983",DATUM["D_North_American_1983",SPHEROID["GRS_1980",6378137.0,298.257222101]],PRIMEM["Greenwich",0.0],UNIT["Degree",0.0174532925199433]]

or upload a .prj file: Choose File No file chosen

Convert

Results

Found a single exact match for the specified search terms:

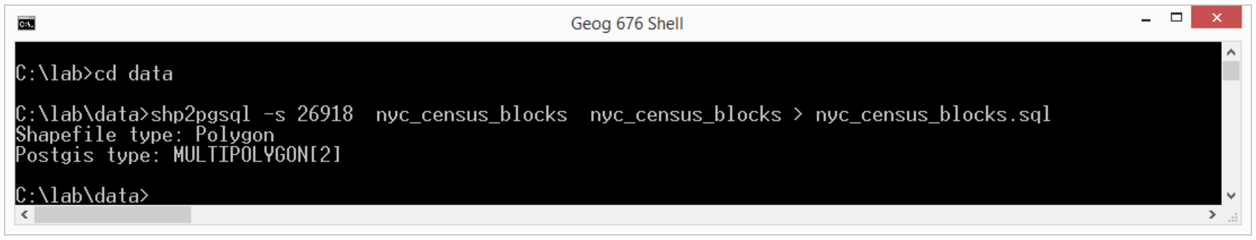
- 4269 - GCS_North_American_1983

2.1.1 Loading the Data with the shell command

The easiest data to load into PostGIS is ESRI shape data since PostGIS comes packaged with a nice command line tool called ***shp2pgsql*** which converts ESRI shape files into PostGIS specific SQL statements that can then be loaded into a PostGIS database.

➔ Open up a command prompt.
➔ Cd to the folder you extracted your data, and run the following command:

```
shp2pgsql -s 26918  nyc_census_blocks  nyc_census_blocks > nyc_census_blocks.sql
```



```
C:\lab>cd data

C:\lab\data>shp2pgsql -s 26918  nyc_census_blocks  nyc_census_blocks > nyc_census_blocks.sql
Shapefile type: Polygon
Postgis type: MULTIPOLYGON[2]

C:\lab\data>
```

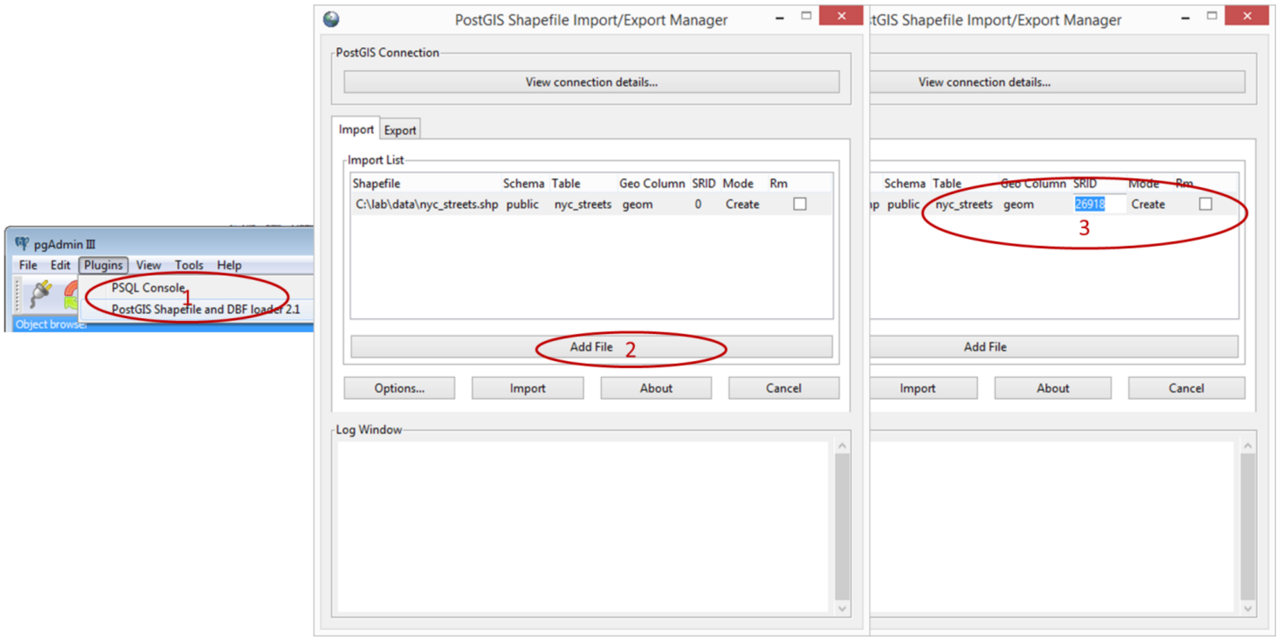
➔ Load into the database with this command:

psql -d *nyc* -U postgres -f *nyc_census_blocks.sql*

2.1.2 Loading the Data with the pgAdmin console

Alternatively you can use the pgAdmin III GUI to load the data. In this case, we load the ***nyc_streets*** shapefile.

➔ Click *Plugins* from the menu, select *PostGIS shapefile and DBF loader 2.1* (shown as step 1).
➔ Click *Add File*, and select the ***nyc_streets.shp*** from your data folder (shown as step 2).
➔ Make sure you change the *SRID* to ***26918*** (step 3).

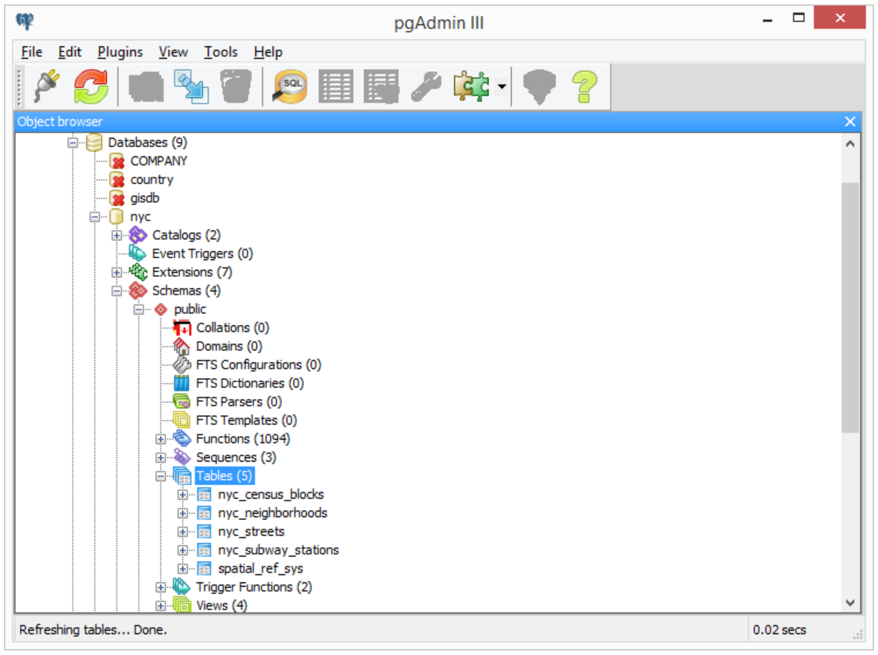

pgAdmin III
File Edit Plugins View Tools Help
PSQL Console
PostGIS Shapefile and DBF loader 2.1
1
PostGIS Shapefile Import/Export Manager
PostGIS Connection
View connection details...
Import Export
Import List
Add File
2
Options... Import About Cancel
Log Window
3

Shapefile	Schema	Table	Geo Column	SRID	Mode	Rm
C:\lab\data\nyc_streets.shp	public	nyc_streets	geom	0	Create	☐
	public	nyc_streets	geom	26918	Create	☐

➔ Click *Import*. (You can load multiple files in one import by adding multiple files before pressing the *Import* button)

Repeat the import process (using EITHER command line OR pgAdmin III) for another two shapefiles in the data directory, including ***nyc_neighborhoods.shp***, and ***nyc_subway_stations.shp***.

After successfully loading all the layers, you will have five tables in your spatial database as follows.



```
Databases (9)
    COMPANY
    country
    gisdb
    nyc
        Catalogs (2)
        Event Triggers (0)
        Extensions (7)
        Schemas (4)
            public
                Collations (0)
                Domains (0)
                FTS Configurations (0)
                FTS Dictionaries (0)
                FTS Parsers (0)
                FTS Templates (0)
                Functions (1094)
                Sequences (3)
                Tables (5)
                    nyc_census_blocks
                    nyc_neighborhoods
                    nyc_streets
                    nyc_subway_stations
                    spatial_ref_sys
                Trigger Functions (2)
                Views (4)
Refreshing tables... Done.                0.02 secs
```

3. Simple SQL

Now that we've loaded data into our database, let's use SQL to ask questions of the data! For example, "What are the names of all the neighborhoods in New York City?" (*See **Appendix** for shapefile fields descriptions*)

Now let's use the SQL query to answer the following questions:

➔ **"What is the population of the City of New York?"**

To answer this question, open up the SQL query window in pgAdmin III by clicking the SQL button, then enter the following query in to the query window:

```
SELECT SUM(popn_total) AS population FROM nyc_census_blocks;
```

You will see the output as:

```
8175032
```

Note: refresh our memory for "alias"

You can give a table or a column another name by using an alias. Aliases can make queries easier to both write and to read. So instead of our outputted column name as ***sum*** we write it **AS** the more readable ***population***.

Now use SQL to answer the following questions (*report your SQL statements and results*):

Q1: What is the population of the 'Manhattan'? (Note: The "*nyc_census_blocks*" table includes a "*boroname*" field which is the borough each block is in. 'Manhattan' is a borough.)

Q2: For each borough, what percentage of the population is native American? (The "*nyc_census_blocks*" table includes a "*pop_nativ*" field that is the number of self-identified native American in each block.)

Descriptions on the columns in the "nyc_census_blocks" table:

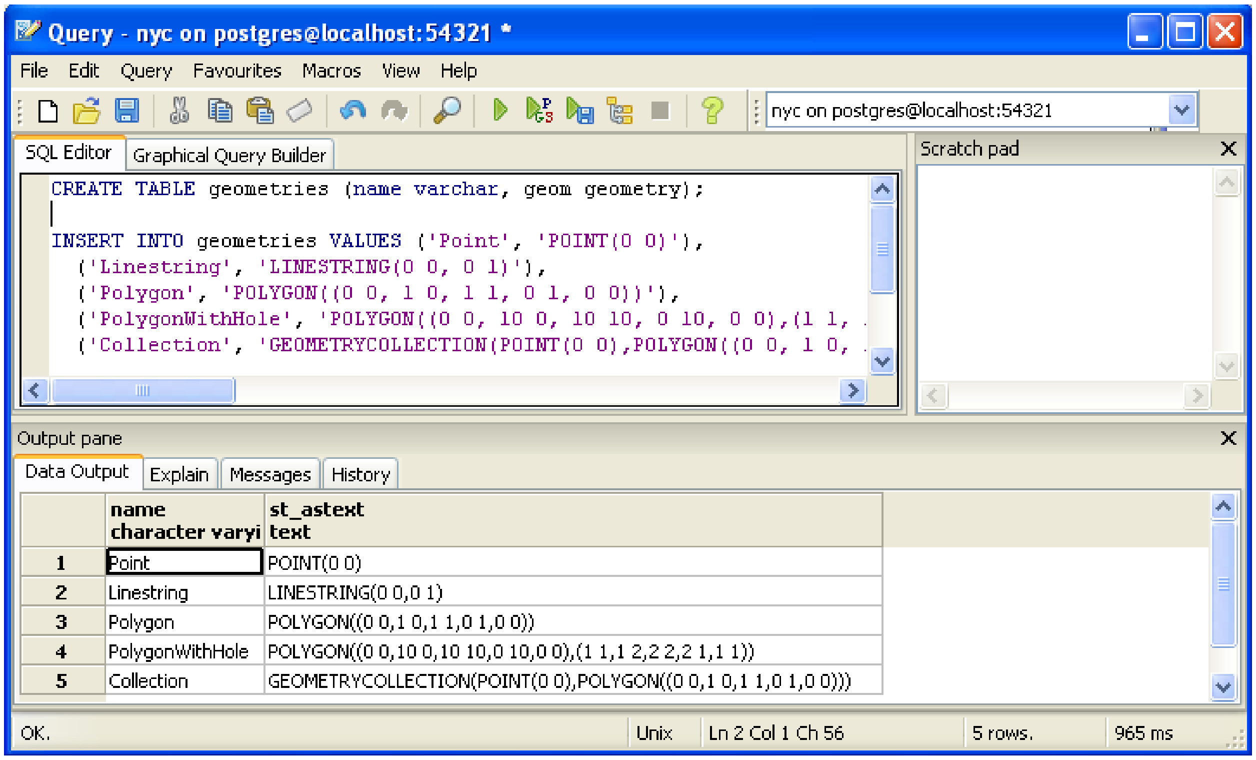
blkid	A 15-digit code that uniquely identifies every census block. Eg: 360050001009000
popn_total	Total number of people in the census block
popn_white	Number of people self-identifying as "white" in the block
popn_black	Number of people self-identifying as "black" in the block
popn_nativ	Number of people self-identifying as "native american" in the block
popn_asian	Number of people self-identifying as "asian" in the block
popn_other	Number of people self-identifying with other categories in the block
boroname	Name of the New York borough: Manhattan, The Bronx, Brooklyn, Staten Island, Queens
geom	MultiPolygon boundary of the block

4. Geometries

In the ***SECTION 2***, we loaded a variety of data and we did some simple SQL query in the ***SECTION 3***. Now let's have a look at some simple spatial query.

In pgAdmin, once again select the ***nyc*** database and open the SQL query tool. Paste this example SQL code into the pgAdmin SQL Editor window and then execute.

```
CREATE TABLE geometries (name varchar, geom geometry);
INSERT INTO geometries VALUES
  ('Point', 'POINT(0 0)'),
  ('Linestring', 'LINESTRING(0 0, 1 1, 2 1, 2 2)'),
  ('Polygon', 'POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))'),
  ('PolygonWithHole', 'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, 1 2, 2 2, 2 1, 1 1))'),
  ('Collection', 'GEOMETRYCOLLECTION(POINT(2 0),POLYGON((0 0, 1 0, 1 1, 0 1, 0 0)))');
SELECT name, ST_AsText(geom) FROM geometries;
```


Query - nyc on postgres@localhost:54321 *
File Edit Query Favourites Macros View Help
nyc on postgres@localhost:54321
SQL Editor
Graphical Query Builder
Scratch pad
Output pane
Data Output
Explain
Messages
History

```
CREATE TABLE geometries (name varchar, geom geometry);

INSERT INTO geometries VALUES ('Point', 'POINT(0 0)'),
  ('Linestring', 'LINESTRING(0 0, 0 1)'),
  ('Polygon', 'POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))'),
  ('PolygonWithHole', 'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, .
  ('Collection', 'GEOMETRYCOLLECTION(POINT(0 0),POLYGON((0 0, 1 0, .
```

	name character varyi	st_astext text
1	Point	POINT(0 0)
2	Linestring	LINESTRING(0 0,0 1)
3	Polygon	POLYGON((0 0,1 0,1 1,0 1,0 0))
4	PolygonWithHole	POLYGON((0 0,10 0,10 10,0 10,0 0),(1 1,1 2,2 2,2 1,1 1))
5	Collection	GEOMETRYCOLLECTION(POINT(0 0),POLYGON((0 0,1 0,1 1,0 1,0 0)))

```
OK.    Unix    Ln 2 Col 1 Ch 56    5 rows.    965 ms
```

The above example *CREATEs* a table (***geometries***) then *INSERTs* five geometries: a point, a line, a polygon, a polygon with a hole, and a collection. Finally, the inserted rows are *SELECTed* and displayed in the Output pane.

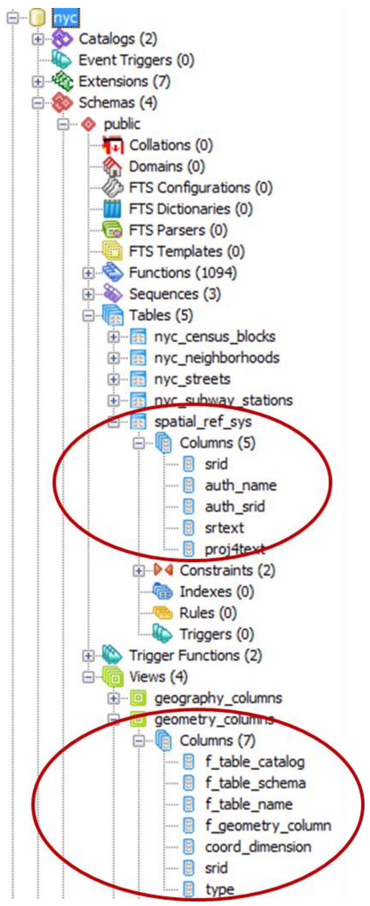
4.1. Metadata Tables

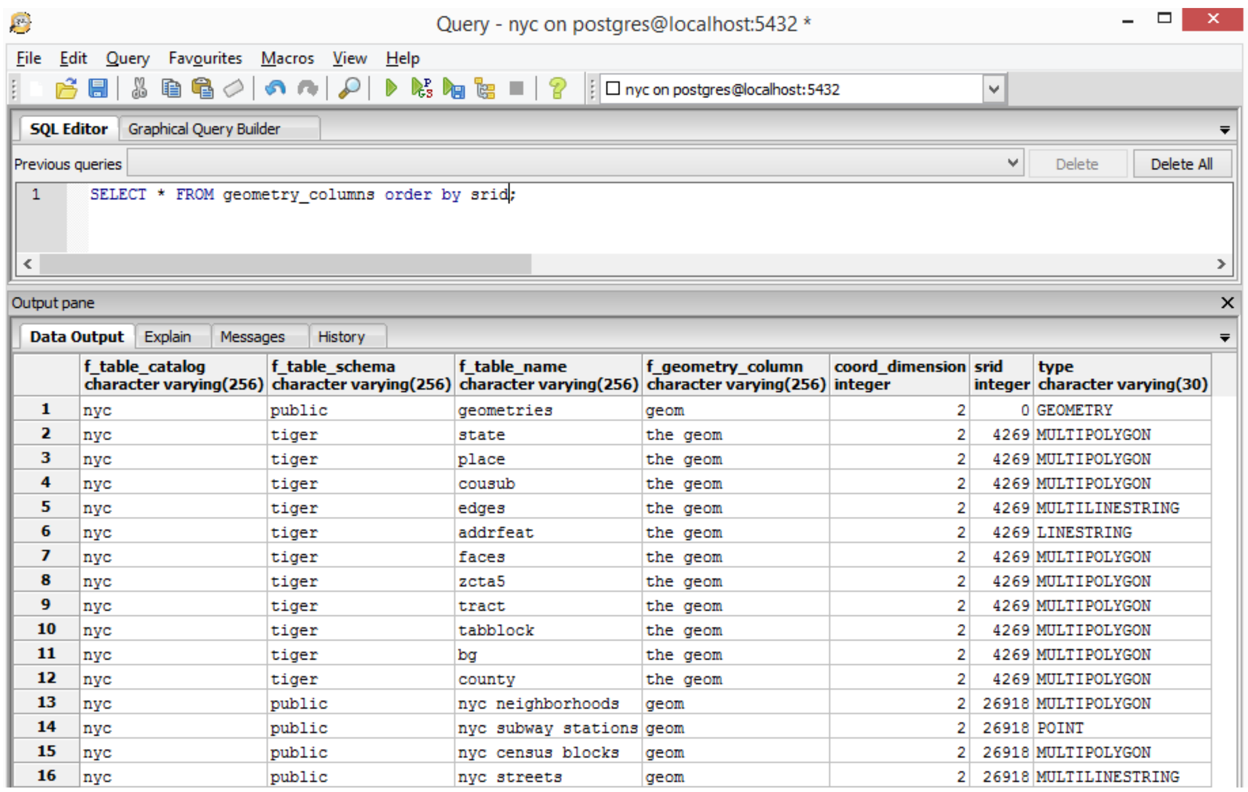
To support the spatial data types and spatial functions that make up a standard spatial database, PostGIS provides two tables to track and report on the geometry types available in a given database (shown as the right figure).

- The first table***, spatial_ref_sys***, defines all the spatial reference systems known to the database and will be described in greater detail later.
- The second table (actually, a view), ***geometry_columns***, provides a listing of all “features” (defined as an object with geometric attributes), and the basic details of those features.

Let’s have a look at the *geometry_columns* table in our database. Paste this command in the Query Tool as before:

```
SELECT * FROM geometry_columns;
```




Query - nyc on postgres@localhost:5432 *
File Edit Query Favourites Macros View Help
nyc on postgres@localhost:5432
SQL Editor
Graphical Query Builder
Previous queries
Delete
Delete All
Output pane
Data Output
Explain
Messages
History

```
SELECT * FROM geometry_columns order by srid;
```

	f_table_catalog character varying(256)	f_table_schema character varying(256)	f_table_name character varying(256)	f_geometry_column character varying(256)	coord_dimension integer	srid integer	type character varying(30)
1	nyc	public	geometries	geom	2	0	GEOMETRY
2	nyc	tiger	state	the_geom	2	4269	MULTIPOLYGON
3	nyc	tiger	place	the_geom	2	4269	MULTIPOLYGON
4	nyc	tiger	cousub	the_geom	2	4269	MULTIPOLYGON
5	nyc	tiger	edges	the_geom	2	4269	MULTILINESTRING
6	nyc	tiger	addrfeat	the_geom	2	4269	LINESTRING
7	nyc	tiger	faces	the_geom	2	4269	MULTIPOLYGON
8	nyc	tiger	zcta5	the_geom	2	4269	MULTIPOLYGON
9	nyc	tiger	tract	the_geom	2	4269	MULTIPOLYGON
10	nyc	tiger	tabblock	the_geom	2	4269	MULTIPOLYGON
11	nyc	tiger	bg	the_geom	2	4269	MULTIPOLYGON
12	nyc	tiger	county	the_geom	2	4269	MULTIPOLYGON
13	nyc	public	nyc_neighborhoods	geom	2	26918	MULTIPOLYGON
14	nyc	public	nyc_subway_stations	geom	2	26918	POINT
15	nyc	public	nyc_census_blocks	geom	2	26918	MULTIPOLYGON
16	nyc	public	nyc_streets	geom	2	26918	MULTILINESTRING

By querying this table, GIS clients and libraries can determine what to expect when retrieving data and can perform any necessary projection, processing or rendering without needing to inspect each geometry. Table *geometry_columns* has the following columns:

- *f_table_catalog*, *f_table_schema*, and *f_table_name* provide the fully qualified name of the feature table containing a given geometry. Because PostgreSQL doesn't make use of catalogs, *f_table_catalog* will tend to be empty.
- *f_geometry_column* is the name of the column that geometry containing column – for feature tables with multiple geometry columns, there will be one record for each.
- *coord_dimension* and *srid* define the dimension of the geometry (2-, 3- or 4-dimensional) and the Spatial Reference system IDentifier that refers to the spatial_ref_sys table respectively.
- The *type* column defines the type of geometry, such as Point, Linestring, Polygon, etc.

Write a SQL query to get answer for the following question:

Q3: What are the geometric data types and srid for table *nyc_streets* and *nyc_subway_stations*?

4.2. Spatial Query

Here's a list of the ***spatial functions*** that would be useful for the exercises!

Functions	Description

Sum([field])	An aggregate function that returns the total value of the field over all records in the query.
LIMIT n	Restrict the query to return only the first "n" rows.
ORDER BY [field]	Return the query in order sorted by the field.

ORDER BY [field] DESC
ORDER BY [field] ASC
	Return the query in descending/ascending order sorted by the field.

ST_X(point)	Returns the X coordinate of the point
ST_Y(point)	Returns the Y coordinate of the point
ST_Length(geometry)	Returns the length of the geometry
ST_Area(geometry)	Returns the area of the geometry
ST_StartPoint(line)	Returns the first point in the line
ST_EndPoint(line)	Returns the last point in the line
ST_NumPoints(line)	Returns the number of vertices in a linestring
ST_NumInteriorRings(polygon)	Returns the number of interior rings (holes) in a polygon
ST_NumGeometries(collection)	Returns the number of sub-geometries in any geometry collection (MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION)
ST_GeometryN(geometry, n)	Returns the n'th geometry in the collection (starting from 1)
ST_AsGML(geometry)	Returns the GML representation
ST_AsKML(geometry)	Returns the KML representation
ST_AsGeoJSON(geometry)	Returns the GeoJSON representation
ST_AsText(geometry)	Returns the well-known-text representation

Now let's do the following ***practice*** using the *spatial functions* built in the PostGIS.

➔ **"What is the area of the 'West Village' neighborhood?"**

```
SELECT ST_Area(geom) FROM nyc_neighborhoods WHERE name = 'West Village';
```

Note: The area is given in square meters. To get an area in hectares, divide by 10000; To get an area in acres, divide by 4047.

➔ **"What is the area of Manhattan in acres?"** (Hint: both *nyc_census_blocks* table and *nyc_neighborhoods* table have a boroname in them.)

```
SELECT          Sum(ST_Area(geom)) / 4047       FROM nyc_neighborhoods
WHERE boroname = 'Manhattan';
```

or...

```
SELECT          Sum(ST_Area(geom)) / 4047       FROM nyc_census_blocks
WHERE boroname = 'Manhattan';
```

➔ **"How many census blocks in New York City have a hole in them?"**

```
SELECT          Count(*)                FROM nyc_census_blocks
```

```
WHERE ST_NumInteriorRings(ST_GeometryN(geom,1)) > 0;
```

Note:

The ST_ NumInteriorRings () functions might be tempting, but it also counts the exterior rings of multi-polygons as well as interior rings. In order to run ST_NumInteriorRings() we need to convert the MultiPolygon geometries of the blocks into simple polygons, so we extract the first polygon from each collection using ST_GeometryN(). Yuck!

➔ **"What is the total length of streets (in kilometers) in New York City?"** (Hint: The units of measurement of the spatial data are meters, there are 1000 meters in a kilometer.)

```
SELECT Sum(ST_Length(geom)) / 1000  FROM nyc_streets;
```

➔ **"How long is 'Columbus Cir' (Columbus Circle) Street?**

```
SELECT Sum(ST_Length(geom))   FROM nyc_streets  WHERE name = 'Columbus Cir';
```

➔ **"What is the length of streets in New York City, summarized by type?"**

```
SELECT type, SUM(ST_Length(geom)) AS length      FROM nyc_streets
GROUP BY type
ORDER BY length  DESC;
```

➔ **"What is the JSON representation of the boundary of the 'West Village' neighborhood?"**

```
SELECT ST_AsGeoJSON(geom) FROM nyc_neighborhoods  WHERE name = 'West Village';
```

Now use spatial SQL to answer the following questions (*report your SQL statements and results*):

Q4: "What is the area of the 'East Village' neighborhood?" (*Note*: *nyc_neighborhoods* table includes a *name* field.)

Q5: "What is the area of 'Brooklyn' borough in acres?" (Note: table *nyc_census_blocks* has a *boroname* field.)

Q6: "What is the total length for the '5th Ave' street? (*nyc_streets* table includes a *name* field.)

Q7: What is the geometry type of 'Pelham St' street? How long is it?

Q8: What is the GML representation of the 'Broad St' subway station? What about the KML? Why are they different? (Note: *nyc_subway_stations* table includes a *name* field)

Q9: "How many polygons are in the 'Red Hook' neigborhood? (Note: *nyc_neigborhoods* table includes a *name* field)

Q10: What is the length of streets for *residential* street type in New York City? (Note: *nyc_streets* table include a *type* field, which categorizes the street into residential, footway, service etc.)

5. Spatial Relationship

So far we have only used spatial functions that measure (**ST_Area**, **ST_Length**), serialize (**ST_GeomFromText**) or deserialize (**ST_AsGML**) geometries. What these functions have in common is that they only work on one geometry at a time.

Spatial databases are powerful because they not only store geometry, they also have the ability to compare *relationships between geometries*.

Questions like "Which are the closet bike racks to a park?" or "Where are the intersections of subway lines and streets?" can only be answered by comparing geometries representing the bike racks, streets, and subway lines.

Here is a list of ***spatial operations to compare geometries***.

Functions	Description
Sum([field])	An aggregate function that Returns the total value of the field over all records in the query.
ST_Contains(A, B)	Returns true if geometry A contains geometry B
ST_Crosses(A, B)	Returns true if geometry A crosses geometry B
ST_Disjoint(A, B)	Returns true if the geometries do not "spatially intersect"
ST_Distance(A, B)	Returns the minimum distance between geometry A and geometry B
ST_DWithin(A, B, d)	Returns true if geometry A is distance or less from geometry B
ST_Equals(A, B)	Returns true if geometry A is the same as geometry B
ST_Intersects(A, B)	Returns true if geometry A intersects geometry B
ST_Overlaps(A, B)	Returns true if geometry A and geometry B share space, but are not completely contained by each other
ST_Touches(A, B)	Returns true if geometry A and geometry B share space, but are not completely contained by each other
ST_Within(A, B)	Returns true if geometry A is within geometry B

Now let's do the following practice using the ***spatial functions*** built in the PostGIS.

➔ **"What is the geometry value for the street named 'Atlantic Commons'?"**

```
SELECT ST_AsText(geom) FROM nyc_streets WHERE name = 'Atlantic Commons';
```

You will see the result as:

```
"MULTILINESTRING((586781.701577724 4504202.15314339,586863.51964484 4504215.9881701))"
```

➔ **"What neighborhood and borough is 'Atlantic Commons' in?"**

```
SELECT name, boroname FROM nyc_neighborhoods
WHERE ST_Intersects(geom,ST_GeomFromText('MULTILINESTRING((586782 4504202,586864 45042
16))', 26918));
```

Since we only have one line segment, we can use the following statements:

```
SELECT name, boroname FROM nyc_neighborhoods
WHERE ST_Intersects(geom,ST_GeomFromText('LINESTRING(586782 4504202,586864 4504216)', 26
918));
```

➔ **"What streets does Atlantic Commons join with?"**

```
SELECT name FROM nyc_streets WHERE ST_DWithin( geom, ST_GeomFromText('LINESTRING(58
6782 4504202,586864 4504216)', 26918),0.1);
```

Owners Corporation
Cumberland St
S Oxford St
Atlantic Commons
160
368
363
1
Mind-Sweeper

➔ **"Approximately how many people live on (within 50 meters of) Atlantic Commons?"**

```
SELECT SUM(popn_total)  FROM nyc_census_blocks  WHERE ST_DWithin( geom,
  ST_GeomFromText('LINESTRING(586782 4504202,586864 4504216)', 26918),50 );
```

Now use spatial SQL to answer the following questions (*report your SQL statements and results*):

Q11: What is the geometry value for the street named ‘Adlai Cir’?

Q12: What neighborhood and borough is ‘Adlai Cir’ street in?

Q13: What streets does ‘Adlai Cir’ join with?

Q14: Approximately how many people live on (within 1km of) ‘Atlantic Commons’ street?

6. Spatial Join

6.1. Join

Spatial joins are the bread-and-butter of spatial databases. They allow you to combine information from different tables by using spatial relationships as the join key. Much of what we think of as “standard GIS analysis” can be expressed as spatial joins.

In the previous section, we explored spatial relationships using a two-step process: first we extracted a subway station point for ‘Broad St’; then, we used that point to ask further questions such as “what neighborhood is the ‘Broad St’ station in?”

Using a spatial join, we can answer the question in one step, retrieving information about the subway station and the neighborhood that contains it:

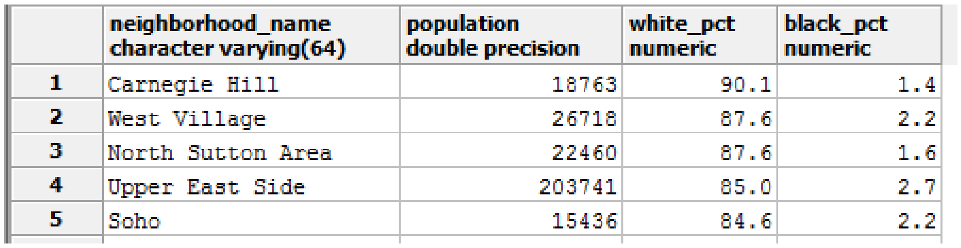
```
SELECT  subways.name AS subway_name,
  neighborhoods.name AS neighborhood_name,
  neighborhoods.boroname AS borough
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_subway_stations AS subways
ON ST_Contains(neighborhoods.geom, subways.geom)
WHERE subways.name = 'Broad St';
```

We could have joined every subway station to its containing neighborhood, but in this case we wanted information about just one. Any function that provides a true/false relationship between two tables can be used to drive a spatial join, but the most commonly used ones are: ST_Intersects**,** ST_Contains**, and** ST_DWithin**.**

```
SELECT  neighborhoods.name AS neighborhood_name, SUM(census.popn_total) AS population,
  Round((100.0 * SUM (census.popn_white) / SUM (census.popn_total))::numeric,1) AS white_pct,
  Round((100.0 * SUM (census.popn_black) / SUM (census.popn_total))::numeric,1) AS black_pct
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_census_blocks AS census
ON ST_Intersects(neighborhoods.geom, census.geom)
```

```
WHERE neighborhoods.boroname = 'Manhattan'
GROUP BY neighborhoods.name
ORDER BY white_pct DESC;
```

The result would be something like:



	neighborhood_name character varying(64)	population double precision	white_pct numeric	black_pct numeric
1	Carnegie Hill	18763	90.1	1.4
2	West Village	26718	87.6	2.2
3	North Sutton Area	22460	87.6	1.6
4	Upper East Side	203741	85.0	2.7
5	Soho	15436	84.6	2.2

What's going on here? Notionally (the actual evaluation order is optimized under the covers by the database) this is what happens:

1. The JOIN clause creates a virtual table that includes columns from both the neighborhoods and census tables.
2. The WHERE clause filters our virtual table to just rows in Manhattan.
3. The remaining rows are grouped by the neighborhood name and fed through the aggregation function to **SUM()** the population values.
4. After a little arithmetic and formatting (e.g., GROUP BY, ORDER BY) on the final numbers, our query spits out the percentages.

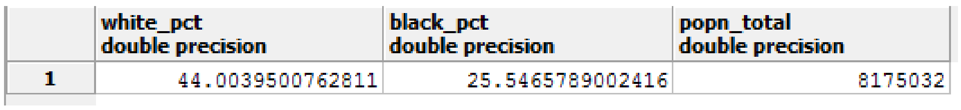
Note: Refresh our memory about Join

The JOIN clause combines two FROM items. By default, we are using an INNER JOIN, but there are four other types of joins. For further information see the join_type definition in the PostgreSQL documentation.

We can also use distance tests as a join key, to create summarized "all items within a radius" queries. Let's explore the racial geography of New York using distance queries.

First, let's get the baseline racial make-up of the city.

```
SELECT  100.0 * SUM(popn_white) / SUM(popn_total) AS white_pct, 100.0 * SUM(popn_black) / SUM(po
pn_total) AS black_pct,  SUM(popn_total) AS popn_total
FROM nyc_census_blocks;
```



	white_pct double precision	black_pct double precision	popn_total double precision
1	44.0039500762811	25.5465789002416	8175032

So, of the 8M people in New York, about 44% are "white" and 26% are "black".

Duke Ellington once sang that "You / must take the A-train / To / go to Sugar Hill way up in Harlem." As we saw earlier, Harlem has far and away the highest African-American population in Manhattan (80.5%). Is the same true of Duke's 'A-train' line?

First, note that in the *nyc_subway_stations* table, *routes* field is what we are interested in to find the 'A-train' line. The values in there are a little complex.

```
SELECT DISTINCT routes FROM nyc_subway_stations;
```

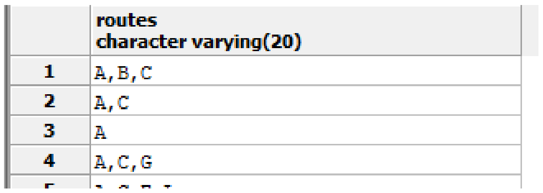
Note: Refresh our memory about DISTINCT

The DISTINCT keyword eliminates duplicate rows from the result. Without the DISTINCT keyword, the query above identifies 491 results instead of 73.

Let's summarize the racial make-up of within 200 meters of the 'A-train' line.

So to find the A-train, we will want any row in routes that has an 'A' in it. We can do this a number of ways, but today we will use the fact that **strpos(routes,'A')** will return a non-zero number if 'A' is in the routes field.

```
SELECT DISTINCT routes
FROM nyc_subway_stations AS subways
WHERE strpos(subways.routes,'A') > 0;
```

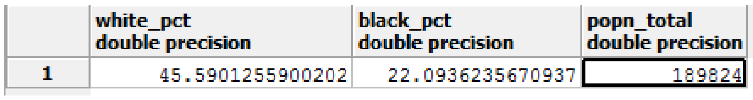


	routes character varying(20)
1	A,B,C
2	A,C
3	A
4	A,C,G

Q15: If you are not allowed to use *strops()* function, how can you get all the routes that include A-train? (Hint: use keyword *LIKE*)

Let's summarize the racial make-up of within 200 meters of the A-train line.

```
SELECT
  100.0 * SUM(popn_white) / SUM (popn_total) AS white_pct,
  100.0 * SUM (popn_black) / SUM (popn_total) AS black_pct,
  SUM (popn_total) AS popn_total
FROM nyc_census_blocks AS census
JOIN nyc_subway_stations AS subways
ON ST_DWithin(census.geom, subways.geom, 200)
WHERE strpos(subways.routes,'A') > 0;
```



	white_pct double precision	black_pct double precision	popn_total double precision
1	45.5901255900202	22.0936235670937	189824

So the racial make-up along the A-train isn't radically different from the make-up of New York City as a whole.

6.2. Advanced Join

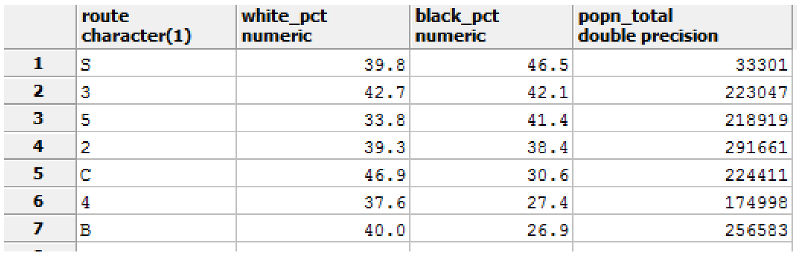
In the last section we saw that the A-train didn't serve a population that differed much from the racial make-up of the rest of the city. Are there any trains that have a non-average racial make-up?

To answer that question, we'll add another join to our query, so that we can simultaneously calculate the make-up of many subway lines at once. To do that, we'll need to create a new table that enumerates all the lines we want to summarize.

```
CREATE TABLE subway_lines ( route char(1) );
INSERT INTO subway_lines (route) VALUES
  ('A'),('B'),('C'),('D'),('E'),('F'),('G'),
  ('J'),('L'),('M'),('N'),('Q'),('R'),('S'),
  ('Z'),('1'),('2'),('3'),('4'),('5'),('6'),
  ('7');
```

Now we can join the table of subway lines onto our original query.

```
SELECT
  lines.route,
  Round((100.0 * SUM(popn_white) / SUM(popn_total))::numeric, 1) AS white_pct,
  Round((100.0 * SUM(popn_black) / SUM(popn_total))::numeric, 1) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks AS census INNER JOIN nyc_subway_stations AS subways
ON ST_DWithin(census.geom, subways.geom, 200)
INNER JOIN subway_lines AS lines ON strpos(subways.routes, lines.route) > 0
GROUP BY lines.route
ORDER BY black_pct DESC;
```



	route character(1)	white_pct numeric	black_pct numeric	popn_total double precision
1	S	39.8	46.5	33301
2	3	42.7	42.1	223047
3	5	33.8	41.4	218919
4	2	39.3	38.4	291661
5	C	46.9	30.6	224411
6	4	37.6	27.4	174998
7	B	40.0	26.9	256583

As before, the joins create a virtual table of all the possible combinations available within the constraints of the JOIN ON restrictions, and those rows are then fed into a GROUP summary. The spatial magic is in the ST_DWithin function that ensures only census blocks close to the appropriate subway stations are included in the calculation.

6.3. Exercise

➔ **"What subway station is in 'Little Italy'? What subway route is it on?"**

```
SELECT s.name, s.routes FROM nyc_subway_stations AS s
INNER JOIN nyc_neighborhoods AS n ON ST_Contains(n.geom, s.geom)
WHERE n.name = 'Little Italy';
```

➔ **"What are all the neighborhoods served by the 6-train?"** (Hint: The routes column in the nyc_subway_stations table has values like 'B,D,6,V' and 'C,6')

```
SELECT DISTINCT n.name, n.boroname FROM nyc_subway_stations AS s
INNER JOIN nyc_neighborhoods AS n
ON ST_Contains(n.geom, s.geom)
WHERE strpos(s.routes,'6') > 0;
```

➔ **"After 9/11, the 'Battery Park' neighborhood was off limits for several days. How many people had to be evacuated?"**

```
SELECT SUM(popn_total)
FROM nyc_neighborhoods AS n
INNER JOIN nyc_census_blocks AS c
ON ST_Intersects(n.geom, c.geom)
WHERE n.name = 'Battery Park';
```

➔ **"What are the population density (people / km^2) of the 'Upper West Side' and 'Upper East Side'?"** (Hint: There are 1000000 m^2 in one km^2.)

```
SELECT  n.name, SUM (c.popn_total) / (ST_Area(n.geom) / 1000000.0) AS popn_per_sqkm
FROM nyc_census_blocks AS c INNER JOIN nyc_neighborhoods AS n ON ST_Intersects(c.geom, n.geom)
WHERE n.name = 'Upper West Side' OR n.name = 'Upper East Side'
GROUP BY n.name, n.geom;
```

Now use spatial SQL to answer the following questions (*report your SQL statements and results*):

Q16: What subway stations are in 'East Village' neighborhood? What subway route is it on?

Q17: What are all the neighborhoods served by the ‘A-train’ line? (Hint: *routes* field in *nyc_subway_stations* table has values like ‘A’)

Q18: If there is a terrorist attack within the ‘central park’ neighborhood. How many people had to be evacuated?

Q19: What are the population density (people / km^2) of the ‘Upper East Side’, ‘Upper West Side’ , ‘Lower East Side’ neighborhood? (Hint: use keyword IN)

Q20: What neighborhood has the highest population density (persons/km^2)? The lowest? There are 1,000,000 m^2 in a km^2.

Useful SQL

The pattern for a spatial join is commonly

```
SELECT a.field, b.field
FROM table_a AS a
JOIN table_b AS b
ON ST_Something(a.geom, b.geom)
WHERE a.field = ‘SOMETHING’;
```

For spatial joins that aggregate results over the whole set, the pattern is commonly

```
SELECT Sum(a.field), b.field
FROM table_a AS a
JOIN table_b AS b
ON ST_Something(a.geom, b.geom)
WHERE a.field = ‘SOMETHING’

GROUP BY b.field;
```

Note the aggregate function around one term and the grouping on the other.

Reference:

Introduction to PostGIS:

http://workshops.boundlessgeo.com/postgis-intro/index.html

Getting Started With PostGIS: An almost Idiot's Guide:

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

A web service to get srid from the .prj file: http://prj2epsg.org/search