Published on *GEOG 868: Spatial Databases* (https://www.e-education.psu.edu/spatialdb)

Lesson 4: Advanced Postgres/PostGIS Topics

Overview

Overview

The real power of a spatial database is in its ability to conduct spatial analysis. This lesson focuses on the many spatial functions that are made available in the Postgres environment by the PostGIS extension. These functions are categorized in Chapter 8 of the online PostGIS manual [1] based on the jobs they do:

- Management Functions
- Geometry Constructors
- Geometry Accessors
- Geometry Editors
- Geometry Outputs
- Spatial Relationships and Measurements
- Geometry Processing Functions
- Miscellaneous Functions

We won't discuss every function in all of these categories, but we will go through most of the more useful ones with plenty of examples.

Objectives

At the successful completion of this lesson, students should be able to:

- add geometry columns to Postgres tables;
- correct undefined or incorrectly defined spatial reference info in a table's metadata;
- create new point, line, polygon, and envelope geometries using PostGIS functions;
- retrieve properties of existing geometries;
- transform geometries from one spatial reference to another;
- output geometries to a number of industry formats;
- examine proximity and containment relationships between geometries;
- retrieve measures such as area, length, and perimeter from geometries.

Questions?

If you have any questions now or at any point during this week, please feel free to post them to the Lesson 4 Discussion Forum.

Checklist

Checklist

Lesson 4 is one week in length. See the Canvas Calendar for specific due dates. To finish this lesson, you must complete the activities listed below:

1. Download the data needed for Project 4 [2].
2. Work through Lesson 4.
3. Complete Project 4 and upload its deliverables to the Project 4 Dropbox.

4. Complete the Lesson 4 Quiz.

Management Functions

Management Functions

AddGeometryColumn()
We used this function in the previous lesson. Recall that we used this function instead of adding the geometry column through the Postgres table definition GUI because PostGIS requires an entry in the geometry_columns view. AddGeometryColumn() handles this for us.

DropGeometryColumn()
As the name implies, this function does the opposite of AddGeometryColumn().

UpdateGeometrySRID()
This function is the equivalent of Esri's Define Projection tool. Use it if a geometry column has its SRID defined incorrectly. As with the Define Projection tool, this function has no effect on coordinate values; it simply modifies the metadata for the geometry column.

Geometry Constructors

Geometry Constructors

The constructor functions are used to create new geometries. While it's natural to think of using these functions to populate a table's geometry column, you should keep in mind that they can also be used to create short-lived geometries that are used only within a query. For example, you might use the ST_MakeEnvelope() constructor function to create a bounding box that you then use to select intersecting geometries in some table.

ST_GeomFromText()
We used this function throughout Lesson 3 to create various types of geometries. In the documentation, you may see that it takes Well-Known Text (WKT) as its input. The "Well-Known" in Well-Known Text refers to the fact that the text follows an industry standard for specifying geometries.

There are a number of other "FromText" functions (e.g., ST_LineFromText, ST_PointFromText) that produce the same result as ST_GeomFromText(). The difference in these functions is that they validate the input text to ensure that it matches the expected geometry type. Thus, ST_PointFromText() will return Null if it is passed something other than a POINT geometry. If you don't see any benefit to this validation, then you're better served using the more generic ST_GeomFromText() as it performs a bit better.

ST_GeogFromText()
At the end of Lesson 3, we talked about the *geography* data type. You can use ST_GeogFromText() to populate geography columns in the same way we used ST_GeomFromText() to populate *geometry* columns. Both functions allow for specifying or omitting the SRID of the coordinates. If omitted, ST_GeomFromText() makes no assumption about the SRID; it is set to 0. ST_GeogFromText(), on the other hand, assumes that the SRID is 4326 unless specified otherwise.

The "Make" functions
An alternative to creating geometries with ST_GeomFromText() are the "Make" functions: ST_MakePoint(), ST_MakeLine() and ST_MakePolygon(). These functions offer better performance than ST_GeomFromText() and their syntax is a bit more compact. So, why does ST_GeomFromText() even exist? Well, ST_GeomFromText() provides compliance with the Open Geospatial Consortium (OGC)'s Simple Features specification. The advantage to using ST_GeomFromText() is that other RDBMSs (Oracle, SQL Server) offer nearly identical functions; your familiarity with the function would transfer to those other environments. One drawback of the "Make" functions is that they do not take SRID as an input, which results in the returned geometry having an undefined SRID. Thus calls to these functions are normally nested inside a call to ST_SetSRID.

Given the advantages offered by the “Make” functions, let’s have a look at their use.

ST_MakePoint()
As you’d probably guess this function accepts an X coordinate and a Y coordinate (and optionally Z and M values) as inputs. Here is how to add the Empire State Building to our nyc_poi.pts table using ST_MakePoint():

```
INSERT INTO pts (name, geom)
    VALUES ('Empire State Building', ST_SetSRID(ST_MakePoint(-73.985744, 40.748549),4269));
```

ST_MakeLine()

This function has three alternative syntaxes. The first simply accepts two point geometries as inputs and returns a straight line connecting the two. We could use this syntax to add the Lincoln Tunnel feature:

```
INSERT INTO lines (name, geom)
    VALUES ('Lincoln Tunnel',
    ST_SetSRID(ST_MakeLine(ST_MakePoint(-74.019921, 40.767119),
    ST_MakePoint(-74.002841, 40.759773)),4269));
```

The second syntax is used when the points that make up the line are stored in a table. The field containing the points is specified as the only input to the function. This example from the PostGIS documentation shows creating a line from a set of GPS points:

```
SELECT gps.gps_track, ST_MakeLine(gps.the_geom) As newgeom
    FROM (SELECT gps_track, gps_time, the_geom FROM gps_points ORDER BY gps_track, gps_time) As gps
    GROUP BY gps.gps_track;
```

In the example, a table called gps_points stores a track identifier (gps_track), the time (gps_time) and the point geometry (the_geom). The data held in those three columns are retrieved as a subquery with an alias of gps. The parent query groups by the track identifier and passes the point geometry field to the ST_MakeLine() function to create a line from the points.

The last ST_MakeLine() syntax accepts an array of points as its input. We could use this syntax to add the Holland Tunnel:

```
INSERT INTO lines (name, geom)
    VALUES ('Holland Tunnel',
    ST_SetSRID(ST_MakeLine(ARRAY[ST_MakePoint(-74.036486,40.730121),
    ST_MakePoint(-74.03125,40.72882),
    ST_MakePoint(-74.011123,40.725958)]),4269));
```

This example demonstrates the use of the Postgres ARRAY data type. Arrays are built using the ARRAY keyword followed by a list of items enclosed in square brackets.

The documentation shows a clever use of the array syntax in which the centroids of polygons stored in a table called visit_locations are used as input to ST_MakeLine():

```
SELECT ST_MakeLine(ARRAY(SELECT ST_Centroid(the_geom) FROM visit_locations ORDER BY visit_time));
```

ST_MakePolygon()
This function takes a LINESTRING representing the polygon’s exterior ring as an input. Optionally, an array of interior ring LINESTRINGs can be included as a second input. Let’s have a look at an example of both, starting with the simpler case:

```
INSERT INTO polys (name, geom)
    VALUES ('Central Park',ST_SetSRID(ST_MakePolygon(ST_GeomFromText('LINESTRING(-73.973057 40.764356,
    -73.981898 40.768094,
    -73.958209 40.800621,
    -73.949282 40.796853,
    -73.973057 40.764356)')),4269));
```

In this example, I used ST_GeomFromText() to create the LINESTRING because supplying the string of points is much easier than if I had taken the ST_MakePoint() approach used for the Holland Tunnel example. In our previous uses of ST_GeomFromText(), we included the optional SRID argument but in this example I omitted it.

Why? Because ST_MakePolygon() will return an SRID-less geometry no matter what, so it's sensible to specify the SRID just once in the call to the ST_SetSRID() function.

And now, here's an example that uses ST_MakePolygon() to cut out the reservoir from the Central Park polygon:

```
INSERT INTO polys (name, geom)
    VALUES ('Central Park',ST_SetSRID(ST_MakePolygon(ST_GeomFromText('LINESTRING(-73.973057 40.764356,
    -73.981898 40.768094,
    -73.958209 40.800621,
    -73.949282 40.796853,
    -73.973057 40.764356)'),
    ARRAY[ST_GeomFromText('LINESTRING(-73.966681 40.785221,
    -73.966058 40.787674,
    -73.9649 40.788291,
    -73.963913 40.788194,
    -73.963333 40.788291,
    -73.962539 40.788259,
    -73.962153 40.788389,
    -73.96181 40.788714,
    -73.961359 40.788909,
    -73.960887 40.788925,
    -73.959986 40.788649,
    -73.959492 40.788649,
    -73.958913 40.78873,
    -73.958269 40.788974,
    -73.957797 40.788844,
    -73.957497 40.788568,
    -73.957497 40.788259,
    -73.957776 40.787739,
    -73.95784 40.787057,
    -73.957819 40.786569,
    -73.960801 40.782394,
    -73.961145 40.78215,
    -73.961638 40.782036,
    -73.962518 40.782199,
    -73.963076 40.78267,
    -73.963677 40.783661,
    -73.965694 40.784457,
    -73.966681 40.785221)')]),4269));
```

ST_MakeEnvelope()
The ST_MakeEnvelope() function is used to create a rectangular box from a list of bounding coordinates: the box's minimum x value, minimum y value, maximum x value, and maximum y value. While it's rare that such a geometry would be used to depict a real-world feature, envelopes are often used as inputs to other functions (e.g., selecting all features that are within a bounding box). Here is an example that produces an envelope surrounding Pennsylvania:

```
SELECT ST_MakeEnvelope(-80.52, 39.72, -74.70, 42.27, 4269);
```

This example simply demonstrates the syntax of ST_MakeEnvelope(). Note that the SRID of the envelope is provided as the last input to the function. We'll see a practical use for this envelope later in the lesson when we talk about the ST_Intersects() function.

Geometry Accessors

Geometry Accessors

Unlike the previous category of functions which were concerned with creating new geometries, this category involves functions used to retrieve information about geometries that already exist.

GeometryType()
This function returns the input geometry's type as a string (e.g., 'POINT', 'LINESTRING', or 'POLYGON'). It comes in particularly handy when dealing with a table of mixed geometries. Here we retrieve just the lines from the mixed nyc_poi table:

```
SELECT name FROM nyc_poi.mixed WHERE GeometryType(geom) = 'LINESTRING';
```

ST_X() and ST_Y()
These functions take a point as input and return its X or Y coordinate in numeric form. Similar functions exist for the M and Z coordinates as well. Here we get the coordinates of our nyc_poi pts data:

```
SELECT name, ST_X(geom), ST_Y(geom) FROM nyc_poi.pts;
```

ST_StartPoint() and ST_EndPoint()

These functions take a LINESTRING or POLYGON as input and return the first and last vertex of that geometry. Here is an example based on our nyc_poi lines table:

```
SELECT name, ST_AsText(ST_StartPoint(geom)), ST_AsText(ST_EndPoint(geom))
    FROM nyc_poi.lines;
```

Note that this example, and many others throughout this section, use the ST_AsText() function to output the returned geometry in a more human-friendly WKT.

ST_NPoints()
This function returns the number of points (vertices) that define the input geometry. Here we get the number of vertices from the states table:

```
SELECT name, ST_NPoints(geom) FROM usa.states ORDER BY name;
```

ST_Envelope()
This function accepts any type of geometry and returns that geometry's minimum bounding box. Here we get the bounding box for Pennsylvania:

```
SELECT ST_AsText(ST_Envelope(geom)) FROM usa.states WHERE name = 'Pennsylvania';
```

ST_ExteriorRing()
This function takes a polygon as input and returns its exterior ring as a LINESTRING. Example:

```
SELECT name, ST_AsText(ST_ExteriorRing(geom)) FROM nyc_poi.polys;
```

ST_NumInteriorRings()
This function takes a polygon as input and returns the number of interior rings it contains. Example:

```
SELECT name, ST_NumInteriorRings(geom) FROM nyc_poi.polys;
```

ST_InteriorRingN()
This function takes a polygon and interior ring number as inputs and returns that ring as a LINESTRING. Note that the rings are numbered beginning with 1. This may seem obvious, but in many programming contexts items are numbered beginning with 0. Here we retrieve the Central Park reservoir ring:

```
SELECT name, ST_AsText(ST_InteriorRingN(geom,1)) FROM nyc_poi.polys;
```

Geometry Editors

Geometry Editors

ST_SetSRID()
This function, as we've already seen, is used to set the SRID of a geometry whose SRID is undefined or defined incorrectly. The statement below would re-set the SRID of the geometries in the states table to 4269:

```
SELECT ST_SetSRID(geom, 4269) FROM usa.states;
```

Just as the Define Projection tool in ArcMap only changes metadata and not the coordinate values themselves, ST_SetSRID() also has no effect on coordinate values. Where ST_SetSRID() differs from the Define Projection tool is that it is applied on an individual geometry basis rather than on a table basis. To re-project geometries into different coordinate systems (changing both the data and metadata), use the ST_Transform() function.

ST_Transform()
We worked with this function in Lesson 3. If ST_SetSRID() is analogous to the Define Projection tool, then ST_Transform() is analogous to the Project tool. An important thing to remember about ST_Transform() is that it leaves the underlying geometry unchanged when used in a SELECT query. If you want to store the transformed version of the geometry in a table, you should use ST_Transform() in an UPDATE or INSERT query.

Geometry Outputs

Geometry Outputs

PostGIS offers a number of functions for converting geometries between different forms. We saw ST_AsText() in a few examples from earlier in the lesson. Here we'll look at a few others.

ST_AsBinary()
This function outputs the geometry in Well-Known Binary (WKB) format, as laid out in the OGC specification. Outputting geometries in this format is sometimes necessary to interoperate with third-party applications. Here we output the NYC points in WKB format:

```
SELECT ST_AsBinary(geom) FROM nyc_poi.pts;
```

ST_AsEWKB()
One of the shortcomings of ST_AsBinary() is that it doesn't provide the geometry's SRID as part of its output. That's where the ST_AsEWKB() function comes in (the "E" in "EWKB" stands for extended). Note that the SRID is binary-encoded just like the coordinates, so you shouldn't expect to be able to read the SRID in the output. Here we use ST_AsEWKB() on the same NYC points:

```
SELECT ST_AsEWKB(geom) FROM nyc_poi.pts;
```

ST_AsText()
As we've seen in a number of examples, this function can be used to output geometries in a human-readable format.

ST_AsEWKT()
Just as the ST_AsBinary() function does not include SRID, the same is true of ST_AsText(). ST_AsEWKT() outputs the same text as ST_AsText(), but it also includes the SRID. For example:

```
SELECT ST_AsEWKT(geom) FROM nyc_poi.pts;
```

Other output functions
Other formats supported by PostGIS include GeoJSON (ST_AsGeoJSON), Geography Markup Language (ST_AsGML), Keyhole Markup Language (ST_AsKML) and scalable vector graphics (ST_AsSVG). Consult the PostGIS documentation [3] for details on using these functions.

Spatial Relationship and Measurement Functions

Spatial Relationship and Measurement Functions

The functions in this category are the big ones in terms of providing the true power of a GIS. (So pay attention!)

A. Spatial relationship functions

ST_Contains()
This function takes two geometries as input and determines whether or not the first geometry contains the other. The example below selects each city in the state of New York and checks to see if it is contained by a bounding box, the box representing the bounds of Pennsylvania which we created earlier using MakeEnvelope. This query should return True values for the border cities of Binghamton, Elmira and Jamestown; and False for all other cities.

```
SELECT name, ST_Contains(ST_MakeEnvelope(-80.52, 39.72, -74.70, 42.27, 4269),geom)
    FROM usa.cities
    WHERE stateabb = 'US-NY';
```

ST_Within()
The converse of the ST_Contains() function is ST_Within(), which determines whether or not the first geometry is within the other. Thus, you could obtain the same results returned by ST_Contains() by reversing the geometries:

```
SELECT name, ST_Within(geom,ST_MakeEnvelope(-80.52, 39.72, -74.70, 42.27, 4269))
    FROM usa.cities
    WHERE stateabb = 'US-NY';
```

ST_Covers()
This function will return the same results as ST_Contains() in most cases. To illustrate the difference between the two functions, imagine a road segment that is exactly coincident with a county boundary (i.e., the road forms the boundary between two counties). If the road segment and county geometries were fed to the ST_Contains() function, it would return False. The ST_Covers() function, on the other hand, would return True.

ST_CoveredBy()
This function is to ST_Covers() as ST_Within() is to ST_Contains().

ST_Intersects()
This function determines whether or not two geometries share the same space in any way. Unlike ST_Contains(), which tests whether or not one geometry is fully within another, ST_Intersects() looks for intersection between any parts of the geometries. Returning to the road/county example, a road segment that is partially within a county and partially outside of it would return False using ST_Contains(), but True using ST_Intersects().

ST_Disjoint()
This function is the converse of ST_Intersects(). It returns True if the two geometries share no space and False if they intersect.

ST_Overlaps()
This function is quite similar to ST_Intersects with a couple of exceptions: a. the geometries must be of the same dimension (i.e., two lines or two polygons), and b. one geometry cannot completely contain the other.

ST_Touches()
This function returns True if the two geometries are tangent to one another but do not share any interior space. If the geometries are disjoint or overlapping, the function returns False. Two neighboring land parcels would return True when fed to ST_Touches(); a county and its parent state would yield a return value of False.

ST_DWithin()
This function performs "within a distance of" logic, accepting two geometries and a distance as inputs. It returns True if the geometries are within the specified distance of one another and False if they are not. The example below reports on whether or not features in the NYC pts table are within a distance of 2 miles (5280 feet x 2) of the Empire State Building.

```
SELECT ptsA.name, ptsB.name,
    ST_DWithin(ST_Transform(ptsA.geom,2260),ST_Transform(ptsB.geom,2260),5280*2)
    FROM pts AS ptsA, pts AS ptsB
    WHERE ptsA.name = 'Empire State Building';
```

Some important aspects of this query are:

1. The geometries (stored in the NAD83 latitude/longitude coordinates) are transformed to the New York East State Plane coordinate system before being passed to ST_DWithin(). This avoids measuring distance in decimal degrees.
2. A cross join is used to join the pts table to itself. As we saw in Lesson 1, a cross join produces the cross product of two tables; i.e., it joins every row in the first table to every row in the second table.
3. The WHERE clause restricts the query to showing just the Empire State Building records; if that clause were omitted, the query would output every combination of features from the pts table.

ST_DFullyWithin()
This function is similar to ST_DWithin(), with the difference being that ST_DFullyWithin() requires each point that makes up the two geometries to be within the search distance, whereas ST_DWithin() is satisfied if any of the points comprising the geometries are within the search distance. The example below demonstrates the difference by performing a cross join between the NYC pts and polys.

```
SELECT pts.name, polys.name,
    ST_DWithin(ST_Transform(pts.geom,2260),ST_Transform(polys.geom,2260),5280*2),
    ST_DFullyWithin(ST_Transform(pts.geom,2260),ST_Transform(polys.geom,2260),5280*2)
    FROM pts CROSS JOIN polys
    WHERE pts.name = 'Empire State Building';
```

ST_DWithin() reports that the Empire State Building and Central Park are within 2 miles of each other, whereas ST_DFullyWithin() reports that they are not (because part of the Central Park polygon is greater than 2 miles away). Note that this query shows an alternative syntax for specifying a cross join in Postgres.

B. Measurement functions

ST_Area()
The key point to remember with this function is to use it on a geometry that is suitable for measuring areas. As we saw in Lesson 3, the ST_Transform() function can be used to re-project data on the fly if it is not stored in an appropriate projection.

ST_Area() can be used on both geometry and geography data types. Though geography objects are in latitude/longitude coordinates by definition, ST_Area() is programmed to return area values in square meters when a geography object is passed to it. By default, the area will be calculated using the WGS84 spheroid. This can be costly in terms of performance, so the function has an optional *use_spheroid* parameter. Setting that parameter to false causes the function to use a much simpler, but less accurate, sphere.

ST_Centroid()
See Lesson 3 for example usages of this function.

ST_Distance()
This function calculates the 2D (Cartesian) distance between two geometries. It should only be used at a local or regional scale when the curvature of the earth's surface is not a significant factor. The example below again uses a cross join between the NYC pts table and itself to compute the distance in miles between the Empire State Building and the other features in the table:

```
SELECT ptsA.name, ptsB.name,
    ST_Distance(ST_Transform(ptsA.geom,2260),ST_Transform(ptsB.geom,2260))/5280
    FROM pts AS ptsA CROSS JOIN pts AS ptsB
    WHERE ptsA.name = 'Empire State Building';
```

The ST_Distance() function can also be used to calculate distances between geography data types. If only geography objects are supplied in the call to the function, the distance will be calculated based on a simple sphere. For a more accurate calculation, an optional *use_spheroid* argument can be set to True, as we saw with ST_Area().

ST_DistanceSpheroid() and ST_DistanceSphere()
These functions exist to provide for high-accuracy distance measurement when the data are stored using the geometry data type (rather than geography) and the distance is large enough for the earth's curvature to have an impact. They essentially eliminate the need to transform lat/long data stored as geometries prior to using ST_Distance(). The example below illustrates the use of both functions to calculate the distance between Los Angeles and New York.

```
SELECT cityA.name, cityB.name,
    ST_DistanceSphere(cityA.geom,cityB.geom)/1000 AS dist_sphere,
    ST_DistanceSpheroid(cityA.geom,cityB.geom,'SPHEROID["GRS 1980",6378137,298.257222101]')/1000 AS dist_spheroid
    FROM cities AS cityA CROSS JOIN cities AS cityB
    WHERE cityA.name = 'Los Angeles' AND cityB.name = 'New York';
```

Note that the Spheroid function requires specification of a spheroid. In this case, the GRS80 spheroid is used because it is associated with the NAD83 GCS. Other spheroid specifications can be found in the spatial_ref_sys

table in the public schema. You can query that table like so:

```
SELECT srtext FROM spatial_ref_sys WHERE srid = 4326;
```

The query above returns the description of the WGS84 GCS, including its spheroid parameters. These parameters could be copied for use in the ST_DistanceSpheroid() function as in the example above.

ST_Length()
This function returns the length of a linestring. The length of polygon outlines is provided by ST_Perimeter(); see below. As with measuring distance, be sure to use an appropriate spatial reference. Here we get the length of the features in our NYC lines table in feet:

```
SELECT name, ST_Length(ST_Transform(geom,2260)) FROM lines;
```

As with the ST_Distance() function, ST_Length() accepts the geography data type as an input and can calculate length using either a sphere or spheroid.

ST_3DLength()
This function is used to measure the lengths of linestrings that have a Z dimension.

ST_LengthSpheroid()
Like the ST_DistanceSpheroid() function, this function is intended for measuring the lengths of lat/long geometries without having to transform to a different spatial reference. It can be used on 2D or 3D geometries.

ST_Perimeter()
This function is used to measure the length of the exterior ring of a polygon. Here we obtain the perimeter of Central Park:

```
SELECT name, ST_Perimeter(ST_Transform(geom,2260)) FROM polys;
```

ST_3DPerimeter()
This function is used to measure the perimeter of polygons whose boundaries include a Z dimension.

PostGIS Spatial Function Practice Exercises

PostGIS Spatial Function Practice Exercises

Before moving on to Project 4, try your hand at the following practice exercises. Solutions can be found at the bottom of the page.

1. Add Times Square to the nyc_poi pts table (located at 40.757685,-73.985727).
2. Report the number of points in the geometries of states with a 2010 population over 10 million.
3. Perform an on-the-fly re-projection of your nyc_poi pts data into the New York East NAD27 state plane coordinate system.
4. Output the U.S. cities coordinates in human-readable format, including the SRID.
5. Select the names and centroids of states where the 2010 male population is greater than the female population.
6. List the states that contain a city named Springfield (based on points in the cities table).
7. List the cities that are found in ‘Soda’ states. Sort the cities first by state name, then city name.
8. Select features from the nyc_poi lines table that are within 1 mile of Madison Square Garden.
9. Calculate the distance in kilometers (based on a sphere) between all of the state capitals.
10. Find the SRID of the Pennsylvania North NAD83 state plane feet coordinate system.
11. Using data in the 'usa' schema select states containing cities that have a 'popclass' value of 4 or 5. Pretend that the stateabb column in the cities table doesn't exist. Don't fret if your results include duplicate states. How to handle duplicate results is demonstrated in the solution.

Solutions [4]

PostGIS Spatial Function Practice Exercise Solutions

PostGIS Spatial Function Practice Exercise Solutions

1. Add Times Square to the nyc_poi pts table (located at 40.757685,-73.985727).

```
INSERT INTO nyc_poi.pts (name, geom)
    VALUES ('Times Square', ST_SetSRID(ST_MakePoint(-73.985727,40.757685),4269));
```

OR

```
INSERT INTO nyc_poi.pts (name, geom)
   VALUES ('Times Square', ST_GeomFromText('POINT(-73.985727 40.757685)',4269));
```

2. Report the number of points in the boundaries of states with a 2010 population over 10 million.

```
SELECT name, ST_NPoints(geom)
    FROM states INNER JOIN census2010
    ON states.name = census2010.state
    WHERE census2010.total > 10000000;
```

3. Perform an on-the-fly re-projection of your nyc_poi pts data into the New York East NAD27 state plane coordinate system (SRID 32105).

```
SELECT name, ST_Transform(geom,32015) FROM nyc_poi.pts;
```

OR for human-readable geom:

```
SELECT name, ST_AsText(ST_Transform(geom,32015)) FROM nyc_poi.pts;
```

4. Output the U.S. cities coordinates in human-readable format, including the SRID.

```
SELECT name, ST_AsEWKT(geom) FROM usa.cities;
```

5. Select the names and centroids of states where the 2010 male population is greater than the female population.

```
SELECT name, ST_AsText(ST_Centroid(geom))
    FROM states INNER JOIN census2010 ON states.name = census2010.state
    WHERE male > female;
```

6. List the states that contain a city named Springfield (based on points in the cities table).

```
SELECT states.name
    FROM states CROSS JOIN cities
    WHERE ST_Contains(states.geom, cities.geom) AND cities.name = 'Springfield';
```

OR

```
SELECT states.name
    FROM states CROSS JOIN cities
    WHERE ST_Covers(states.geom, cities.geom) AND cities.name = 'Springfield';
```

7. List the cities that are found in ‘Soda’ states. Sort the cities first by state name, then city name.

```
SELECT cities.name, cities.stateabb
    FROM states CROSS JOIN cities
    WHERE ST_Within(cities.geom, states.geom) AND states.sub_region='Soda'
    ORDER BY cities.stateabb, cities.name;
```

OR

```
SELECT cities.name, cities.stateabb
    FROM states CROSS JOIN cities
    WHERE ST_Intersects(cities.geom, states.geom) AND states.sub_region='Soda'
    ORDER BY cities.stateabb, cities.name;
```

8. Select features from the nyc_poi lines table that are within 1 mile of Madison Square Garden.

```
SELECT lines.name
    FROM lines CROSS JOIN pts
    WHERE pts.name = 'Madison Square Garden' AND
    ST_DWithin(ST_Transform(lines.geom,2260),ST_Transform(pts.geom,2260),5280*1);
```

9. Calculate the distance in kilometers (based on a sphere) between all of the state capitals.

```
SELECT cityA.name, cityB.name, ST_DistanceSphere(cityA.geom, cityB.geom) / 1000 AS dist_km
    FROM cities AS cityA CROSS JOIN cities AS cityB
    WHERE (cityA.capital = 1 AND cityB.capital = 1) AND (cityA.name != cityB.name);
```

10. Find the SRID of the Pennsylvania North NAD83 state plane feet coordinate system.

```
SELECT * FROM spatial_ref_sys WHERE srtext LIKE '%Pennsylvania North%';
```

The correct spatial reference has an SRID of 2271.

11. Select states containing cities with a 'popclass' value of 4 or 5.

```
SELECT DISTINCT states.name
    FROM states CROSS JOIN cities
    WHERE ST_Contains(states.geom, cities.geom) AND cities.popclass >= 4;
```

If you completed this exercise on your own, you probably had Texas appear in your results 3 times. Remember that a cross join creates a result set that combines all the rows from table A with all the rows from table B. The WHERE clause narrows down the cities in the output to those 9 having a popclass of 4 or 5, but 3 of those 9 cities are in Texas. That explains why the query returns Texas 3 times. The answer in a case like this is to insert the DISTINCT keyword after SELECT. This ensures that none of the output rows will be duplicated.

Project 4: Jen and Barry's Site Selection in PostGIS

A. Scenario Refresher

If you took GEOG 483, the first project had you finding the best locations for "Jen and Barry" to open an ice cream business. We're going to revisit that scenario for this project. Your task is to import the project shapefiles into a PostGIS schema and then write a series of SQL statements that automate the site selection process. If you need a new copy of the data, you can download and unzip the Project 4 Data [2] file. Here's a reminder of Jen and Barry's original selection criteria:

- Greater than 500 farms for milk production
- A labor pool of at least 25,000 individuals between the ages of 18 and 64 years
- A low crime index (less than or equal to 0.02)
- A population of less than 150 individuals per square mile
- Located near a university or college
- At least one recreation area within 10 miles
- Interstate within 20 miles

You should narrow the cities down to 9, based on the county- and city-level criteria. After evaluating the interstate and recreation area criteria (which are a bit more difficult), that should get you down to 4 cities.

B. Tips for completing Project 4

- While it may be possible to meet the project requirements using one really long query loaded with embedded subqueries, I suggest you avoid attempting to go that route.
- The most logical workflow I can think of is as follows:

1. Write a query that selects the suitable counties based on the county-level criteria and save this as a view.
2. Write a query that selects cities in the suitable counties that also meet the city-level criteria and save this as a view. This would meet 80% of the project requirements (with incorporating the interstate-recreation area criteria and the quality of your write-up accounting for 10% each).
3. Select from cities in the view saved in #2 above that meet the "near an interstate" criterion and save this as a view.
4. Then select from cities in the view saved in #3 above that meet the "near a recreation area" criterion.

- It should not be necessary to create any new tables to hold intermediate results.
- This is a scenario that calls for the use of cross joins.
- If you look at the .prj files for the project shapefiles, you'll see that they are in geographic coordinates, NAD27 (SRID = 4267). Be sure to re-project the data into a spatial reference that is appropriate for distance measurement when dealing with the interstates and recreation areas. Pennsylvania State Plane North (NAD27 or NAD83), units = feet, would be a logical choice for that purpose.

C. Deliverables

This project is one week in length. Please refer to the Canvas Calendar for the due date.

1. Submit a write-up that summarizes your approach to this project. Most importantly include the SQL code that you developed. Don't forget to include the code built into your views if you create any. You are not required to create a map of the candidate cities in QGIS, though that would leave a good impression with the instructor. :-)

   SQL code that selects the correct 9 cities based on the county- and city-level criteria: 80 of 100 points
   SQL code that incorporates the interstate and recarea criteria: 10 of 100 points
   Quality of write-up: 10 of 100 points
2. Complete the Lesson 4 quiz.

Source URL: https://www.e-education.psu.edu/spatialdb/l4.html

Links
[1] http://postgis.net/docs/manual-3.0/reference.html
[2] https://www.e-education.psu.edu/spatialdb/sites/www.e-education.psu.edu.spatialdb/files/Project4.zip
[3] http://postgis.net/docs/manual-3.0/reference.html#Geometry_Outputs
[4] https://www.e-education.psu.edu/spatialdb/L4_practice_solutions.html