

Spatial Query Tutorial - NYC Subway

Lee Hachadoorian

November 14, 2013

How many people in New York City live within a quarter mile of a subway station? What is their average income? How does that compare to the population that *don't* live near a subway station? These kinds of questions can be answered using a geographic information system, but an alternative to standard GIS is in-database analysis using spatial query. This workshop will introduce you to using spatial functions in SQL.

Prerequisites

Software

SpatiaLite GUI

QGIS

Data

- MTA Subway Entrances: <http://mta.info/developers/data/nyct/subway/StationEntrances.csv>
- MTA Subway Lines (CUNY Mapping Service at the Center for Urban Research): https://wfs.gc.cuny.edu/SRomalewski/MTA_GISdata/June2010_update/nyctsubwayroutes_100627.zip
- USCB NY Tract Shapefile: ftp://ftp2.census.gov/geo/tiger/TIGER2011/TRACT/tl_2011_36_tract.zip
- USCB NY geoheader: ftp://ftp2.census.gov/acs2011_5yr/summaryfile/2007-2011_ACSSF_By_State_By_Sequence_Table_Subset/NewYork/Tracts_Block_Groups_Only/g20115ny.csv
- USCB NY sequence 56 (Household Income): ftp://ftp2.census.gov/acs2011_5yr/summaryfile/2007-2011_ACSSF_By_State_By_Sequence_Table_Subset/NewYork/Tracts_Block_Groups_Only/20115ny0056000.zip

These instructions, with the tutorial data and software, may be downloaded from https://www.dropbox.com/sh/dikcur6ywt5f1yt/NguIn_v3n5.

Loading Data into SpatiaLite

SpatiaLite is a free and open source geodatabase built on top of SQLite, itself a FOSS database. As a lightweight, “server-less” database, SQLite is frequently used as an embedded database by other applications, including Firefox, Google Chrome, Skype, iTunes, Clementine, and Dropbox. SpatiaLite comes with a standalone executable GUI for Windows. This means you don’t actually have to install the software, just double click the executable to run it.

Geodata Formats

If you browse the geographic data at [NYC Open Data](#), you will find a lot of data in “ESRI shapefile” or “ESRI geodatabase” format. Additionally, many of the data are presented as web maps, and these data can all be exported in KML, KMZ, shapefile, or the “original” format (unspecified by the web interface, but in practice usually shapefile or ESRI geodatabase), as well as a variety of aspatial formats.

For storing and working with your own data, you have three broad choices: a file-based format (e.g., shapefile, KML, GeoJSON), a full RDBMS (e.g., PostGIS), or the in-between choice of a file-based database (e.g., SpatiaLite or the MS Access-based ESRI “Personal Geodatabase”). A full RDBMS is an excellent choice if you need access control, versioning, reproducibility, etc., but may be overkill for small projects, and makes sharing data a challenge. The shapefile format is widely supported, but this is about the only thing it has going for it. Shapefiles have, in my opinion, three strikes against them:

1. It is a multi-file format, and worse contains an inconsistent number of files, so one shapefile is actually 3 to 12 filesystem files with the same base name (although in practice most shapefiles will contain 5ish filesystem files).
2. Attribute field names are limited to ten characters because it uses DBF to store attribute data.
3. Support for the DBF format is declining. Microsoft Excel 2007 and later can no longer edit/save the DBF format, although you can still edit these formats with FOSS tools like LibreOffice.

File-based database formats such as SpatiaLite and ESRI personal geodatabase format have a major advantage, which is that they allow combining several data sources (like shapefiles) into one database file (SQLite or Microsoft Access file). This helps keep project folders tidy and makes it easy to transport or share

project data, since it is all contained in one file. An added advantage is that it is often faster and easier to edit or query attribute data without using a full GIS, using standard database tools or by writing a lightweight application. Finally, SpatiaLite has, like PostGIS, incorporated a large number of spatial functions which can be used for spatial analysis, obviating the need for a full GIS in many cases.

Before the workshop, copy the SpatiaLite exercise folder to your local machine.

Create a New Spatial Database

When you first open SpatiaLite GUI, you will see a mostly blank window. You can then open an existing database or create a new one. If you close SpatiaLite GUI with a database loaded (while “connected” in SpatiaLite parlance), the same database will load the next time you launch SpatiaLite GUI.


Choose **File -> Creating a New (empty) SQLite DB**. Call it `nyc.sqlite`¹, and save it to your working directory.²

You will immediately see several tables have appeared in the database. These are system tables necessary to spatially enable your data, but we have not yet imported any actual spatial data.

1. Click **Creating a New (empty) MEMORY-DB**.
2. Click **Saving the current MEMORY-DB** and set the desired file name.
3. Click **Disconnecting current SQLite DB**.
4. Click **Connecting an existing SQLite DB** and choose the newly created file.

Load Shapefiles



Click on the Load Shapefile button . Navigate to your data folder and select `nyctsubwayroutes_100627.shp`. Set the table name to `geo_nyct_subway_lines_2010`³, GeomColumn name to `geom`⁴, and SRID to 4269. Check the box to load the geometry With Spatial Index. This will speed up spatial queries.

¹There is no “standard” SQLite file extension. `.db` and `.sqlite` are commonly used. SpatiaLite GUI expects `.sqlite`, so I have used that here.

²SpatiaLite GUI 1.6 on Linux will take several minutes to initialize a new spatial database, during which the hard drive will be crunching and the application will be unresponsive. This issue did not affect earlier versions on Linux. I do not know if it still affects 1.7 on Linux, but it does *not* affect version 1.7 on Windows. If your version is affected, the following process for initializing a new spatial database will be faster:

³Although not strictly necessary, I prefer to use the `geo_` prefix to distinguish spatial tables from aspatial (“attribute”) tables.

⁴This is a convention adopted from PostGIS 2.0+.

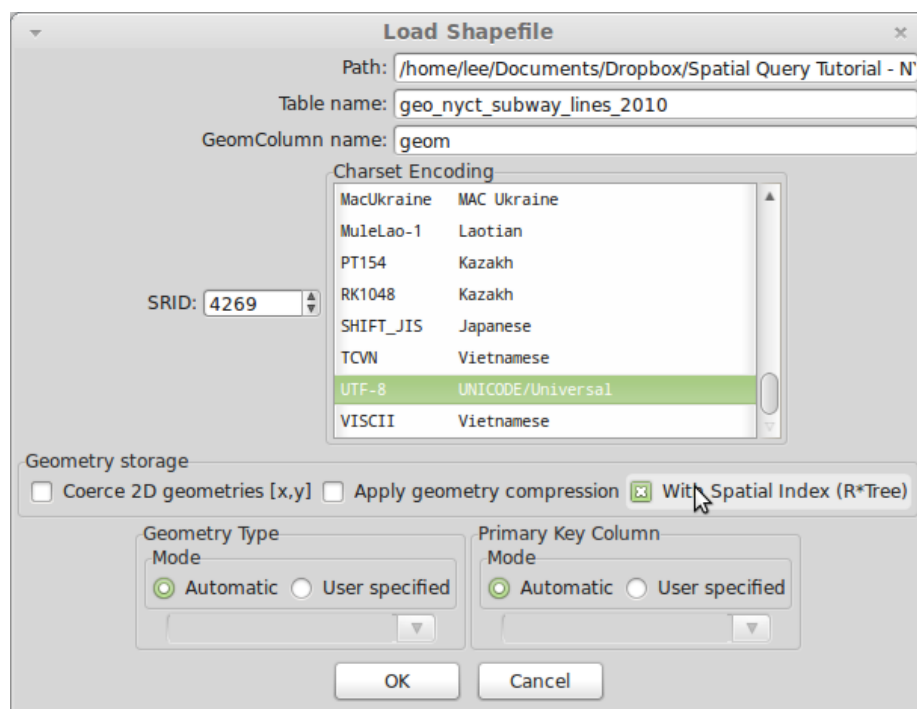


Figure 1: Load Shapefile Dialog

A statement will tell you how many rows were inserted. You will also see `geo_nyct_subway_lines_2010` appear as a new table. Right-clicking on the table name will allow you show the `CREATE TABLE` statement, or to dump the table to a PostGIS load format, among other things. If you expand the tree, you will see the columns. The `geom` column has a globe icon with a lightning bolt next to it, indicating it is a geometry column which has been indexed.



Right-click on `geom` and choose **Map Preview**. This will give you a visual display of the spatial data that you just loaded.

Repeat these steps to import the `tl_2011_36_tract.shp`. Set the name to `geo_tract_2011`, `GeomColumn` to `geom`, and `SRID` to `4269`. If you preview the geometry column, you will see the Census tracts for New York State. SpatiaLite has also added an entry to the `geometry_columns` table for each layer you imported. If you type:

```
SELECT * FROM geometry_columns
```

in the Query window and click the Execute SQL Statement icon to the right of the window, you'll see one record listed in the query result for each layer you imported. You'll see the geometry type, `SRID` etc.

Load Tabular Data (CSV)

We now need to load text data, including the demographic (income) data from the Census Bureau, and the subway station entrances. But, wait! Aren't the subway station entrances spatial data? Not yet. MTA provides the data with the latitude and longitude of the entrances. We need to load the coordinates, then create a geometry column from those coordinates. First we'll load the Census data, because it's relatively straightforward.

Go to **Load CSV/TXT** and choose the file `g20115ny.csv`, a so-called geoheader file. Census raw data files do not have headers, so uncheck "First line contains column names", and set the column separator to Comma. Names of places in the Census data often contain special characters (like ñ), so set the Charset Encoding to "ISO-8859-1 Latin-1 Western European". Unfortunately without a header row, the tables are created with columns named "COL001", "COL002", etc. In order to create a table with the names that conform to the Census tables, open the file `Census data structure.sql`. These table defs are taken from my census-postgres project on GitHub⁵, which uses the Census data dictionary to build the statements programmatically. After the `CREATE TABLE`, an `INSERT` statement copies all the rows from the import table to the final table.⁶ You

⁵<https://github.com/leehach/census-postgres>

⁶In order to avoid having to type all the column names for the `INSERT` statements, I did the following:

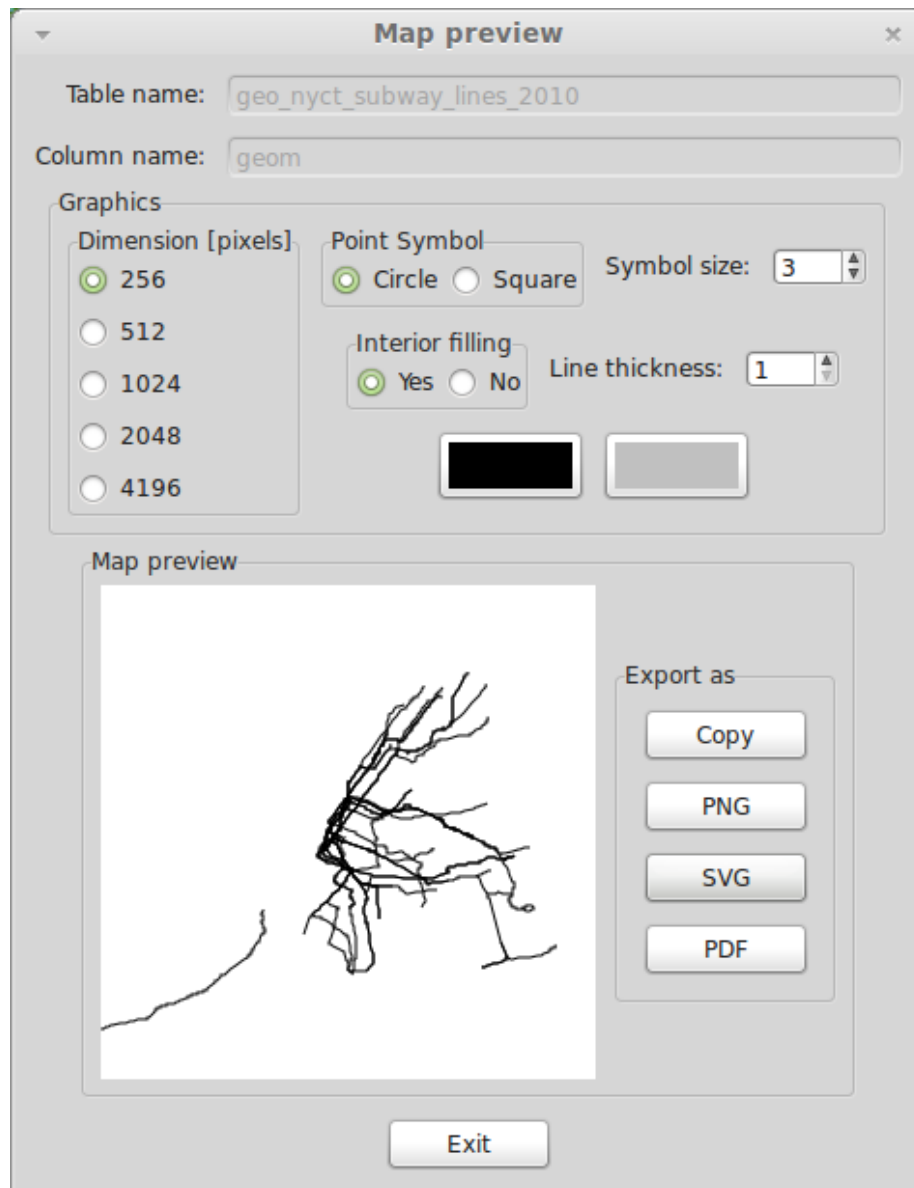


Figure 2: Map Preview Dialog

can copy the entire script to your SQL pane and run it. Then drop the original import tables.

We will discuss how to join the Census data to the spatial layer later, but now let's import the station data. First click the **Load CSV/TXT** button. Choose file **StationEntrances.csv**, set the table name to **nyct_station_entrance_2011**, and set the column separator to **Comma** (the other defaults should be OK). Load the table.

Right click on the table in the database tree and choose **Edit table rows**. You will immediately notice that each station has several rows of data. This is because this is a table of *entrances*. We will create a table of *stations*, with one row per station. In order to do this we will use the **DISTINCT** keyword. When you choose **Edit table rows**, SpatiaLite actually constructs a SQL statement and shows it in the query pane. **ROWID** is a hidden internal identifier (you don't see it in the table definition). The columns up to **Route_11** seem to apply to the *station*, and after that they apply to the *entrance*. For example, if you look at DeKalb Av (PK_UID = 56 to 59), you will see that the **Entrance_Type** for one entrance is "Elevator" and for three others it is "Stairs". Similarly, the last two columns, **Latitude** and **Longitude**, are unique to the entrance. We will leave all of these columns out of our select statement:

```
SELECT DISTINCT
  Division, Line, Station_Name, Station_Latitude, Station_Longitude,
  Route_1, Route_2, Route_3, Route_4, Route_5, Route_6, Route_7,
  Route_8, Route_9, Route_10, Route_11
FROM nyct_station_entrance_2011
```

We now see 466 rows instead of 1862.

Georeferencing XY Coordinates

Now that we know the latitude and longitude of the station, how do we turn that into a geometry column? We use the function **ST_Point(<x>, <y>)**.⁷ **ST_Point()** returns a geometry with the default SRID of 0, so we have to wrap the result in **SetSRID(<geometry>, <srid>)**.⁸ The SRID for this table is 4326,

1. Right-click on the table name that you want to SELECT from and choose **Edit table rows**.
2. A SELECT statement now appears in the SQL pane. Delete the **ROWID** and **PK_UID** columns, as well as the **ORDER BY** clause.
3. Add the **INSERT** line at the top.

⁷SpatiaLite has a similar function **MakePoint()**, but **MakePoint()** does not conform to the OGC standard, and also doesn't match the non-OGC-compliant PostGIS function **ST_MakePoint()**. Since I would rather remember one thing than three, I use **ST_Point()**. I have begun a [Spatial SQL Concordance](#) on my blog.

⁸Annoyingly, **SetSRID** does not conform to the PostGIS/OGC-compliant function **ST_SetSRID()**.

otherwise known as the WGS 84 datum which is used by GPS satellites. These days, if someone gives you a set of latitude and longitude coordinates and doesn't specify the SRID, you can usually assume that it is 4326. We're also going to take this query and turn it into a CREATE TABLE statement.

```
CREATE TABLE geo_nyct_station_2011 AS
SELECT DISTINCT
    Division, Line, Station_Name, Station_Latitude, Station_Longitude,
    Route_1, Route_2, Route_3, Route_4, Route_5, Route_6, Route_7,
    Route_8, Route_9, Route_10, Route_11,
    SetSRID(ST_Point(Station_Longitude, Station_Latitude), 4326) AS geom
FROM nyct_station_entrance_2011
```

After running the query, right-click anywhere in the database tree and choose **Refresh**, and you will see the new table `geo_nyct_station_2011`. But you may notice there is no globe icon on the table, and when you expand the tree you will see that `geom` is an ordinary column. The reason is because this table is not registered in the `geometry_columns` table. We can add it to that table by right-clicking `geom` and choosing **Recover geometry column**. Set the SRID = 4326, Dims = "XY", and Geometry Type = "POINT".

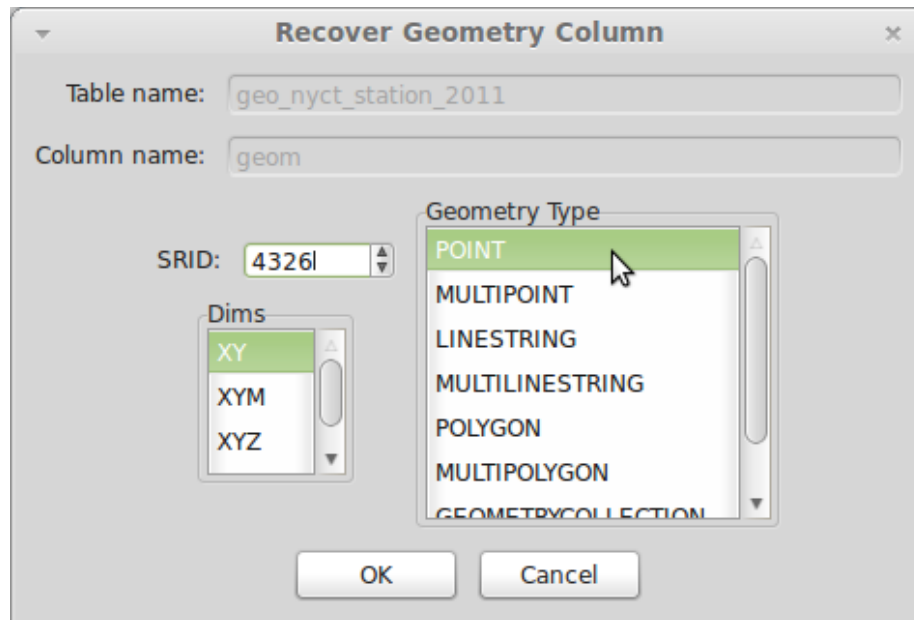


Figure 3: Recover Geometry Column

Your queries will run faster if your spatial columns are indexed, so refresh the database tree, right-click on `geom`, and choose **Build Spatial Index**.

You may have noticed when we recovered the geometry column that there were other geometry types, including MULTIPOINT, MULTILINESTRING, and MULTIPOLYGON. MULTIs mean that geometrically distinct entities are combined into one database row. For example, the state of Hawaii has several islands, but we don't want to have it appear in a database table multiple times, so we use a MULTIPOLYGON to represent its geometry. Multi-polygonal geometries are so common that many polygon datasets that do not strictly need to use MULTIPOLYGON will still be created that way.

Under what circumstances would you use MULTIPOINT? Remember that our subway stations each have several entrances. You might want to store the locations of those entrances in the database, but still only have one record per station, so you could combine those entrances into a MULTIPOINT (instead of using one "average" station location, like we did). I won't discuss it further, but for those of you interested in playing with it, the SQL to create such a table is:

```
CREATE TABLE geo_nyct_station_multi_2011 AS
SELECT
    Division, Line, Station_Name, Station_Latitude, Station_Longitude,
    Route_1, Route_2, Route_3, Route_4, Route_5, Route_6, Route_7,
    Route_8, Route_9, Route_10, Route_11,
    ST_Multi(ST_Collect(SetSRID(ST_Point(Longitude, Latitude), 4326))) AS geom
FROM nyct_station_entrance_2011
GROUP BY
    Division, Line, Station_Name, Station_Latitude, Station_Longitude,
    Route_1, Route_2, Route_3, Route_4, Route_5, Route_6, Route_7,
    Route_8, Route_9, Route_10, Route_11
```

Basic Data Manipulations

Creating a View

A view is a presentation of data that acts like a table, but is not actually a table. As a presentation of data, a view may only include certain table columns (the SELECT list), only include certain table rows (the WHERE clause, join related data from multiple tables, and operate on the data to create new values. Views can also alias column names to make them easier to work with, which will be particularly useful in dealing with the awkward column names of the Census data.

First, for some stupid reason, the Census has two-digit state codes (`stusab`) stored in upper case in the geoheader but in lower case in all the sequence files. In order to save ourselves having to constantly adjust between them (or even remember which is which), let's update the sequence table to use the more natural upper case code.

```
UPDATE acs2011_5yr_seq0056
SET stusab = UPPER(stusab);
```

Now we can go ahead and create a view which aliases the columns to something more intelligible.

```
CREATE VIEW vw_tract_household_income AS
SELECT
  substr(geoid, 8) AS geoid, b19001001 AS hh_total,
  b19001002 AS hh_under_10k, b19001003 AS hh_10k_to_15k,
  b19001004 AS hh_15k_to_20k, b19001005 AS hh_20k_to_25k,
  b19001006 AS hh_25k_to_30k, b19001007 AS hh_30k_to_35k,
  b19001008 AS hh_35k_to_40k, b19001009 AS hh_40k_to_45k,
  b19001010 AS hh_45k_to_50k, b19001011 AS hh_50k_to_60k,
  b19001012 AS hh_60k_to_75k, b19001013 AS hh_75k_to_100k,
  b19001014 AS hh_100k_to_125k, b19001015 AS hh_125k_to_150k,
  b19001016 AS hh_150k_to_200k, b19001017 AS hh_over_200k,
  b19013001 AS median_hh_income, b19025001 AS aggregate_hh_income
FROM acs2011_5yr_seq0056 JOIN acs2011_5yr_geoheader USING (stusab, logrecno)
WHERE sumlevel = 140;
```

In addition to aliasing the columns, this statement is doing a number of other things:

1. We are using `substr(<text>, <start>, <end>)` to take a substring from field `geoid` beginning at position 8 and (since the third parameter is omitted) going to the end of the string. We are doing this because the `geo_tract_2011` table also has a `geoid` field with a slightly different format. By dropping the first 7 characters, the fields will be joinable.
2. We are joining the `geoheader` and `sequence` tables on their primary key fields, `stusab` and `logrecno`. We are doing this with the `USING` keyword. `USING` can only be used when the join fields have the same name, and it allows us to avoid using the more verbose construction `ON (acs2011_5yr_seq0056.stusab = acs2011_5yr_geoheader.stusab AND acs2011_5yr_seq0056.logrecno = acs2011_5yr_geoheader.logrecno)`.
3. We are using the `WHERE` clause to restrict the resultset to summary level 140, which means Census tracts.

Refresh the database tree and you will see the view.

Query the data

Our data is finally starting to come into shape. Now let's try to ask it some questions.

```

SELECT *
FROM vw_tract_household_income
WHERE median_hh_income > 50000

```

How many census tracts have a median income of more than \$50,000?

We can also aggregate the data in our query using the `GROUP BY` clause. Let's say we want to find the percent of households that earns less than \$50,000 annually by borough. (Each borough is a county.) We need to include the grouping criteria in the column list. Each additional column must be part of an aggregate function, which is a function applied to the values in all database rows within each group. For example:

```

SELECT
  substr(geoid, 1, 5) AS county_fips, sum(hh_total) AS hh_total,
  sum(hh_under_10k + hh_10k_to_15k + hh_15k_to_20k + hh_20k_to_25k
    + hh_25k_to_30k + hh_30k_to_35k + hh_35k_to_40k + hh_40k_to_45k
    + hh_45k_to_50k) AS hh_under_40k,
  100.0 * sum(hh_under_10k + hh_10k_to_15k + hh_15k_to_20k + hh_20k_to_25k
    + hh_25k_to_30k + hh_30k_to_35k + hh_35k_to_40k + hh_40k_to_45k
    + hh_45k_to_50k) / sum(hh_total) AS pct_hh_under_40k
FROM vw_tract_household_income
WHERE substr(geoid10, 1, 5) IN ('36005', '36047', '36061', '36081', '36085')
GROUP BY county_fips;

```

This tells the SQL parser to group on the `county_fips` columns. The first two digits of the result are the state code, the next three digits are the county code, which are only unique within a state, which is why we use a five-digit code for each county. Then within each county, get the total number of households `hh_total` of each tract and add them all together, i.e. take the `sum()`. Note that we are also aliasing the column with the `AS` keyword. The alias could be anything we choose, but here we have simply reused the name `hh_total`. We do the same thing for the number of households in the 9 income buckets below \$50,000 (note that we add them to each other within each row, and then we take the sum across all rows), and in the final column we divide these two numbers and multiply by 100 to get the percent of households with earnings below \$50,000 within each county.

Using spatial functions

None of this is too interesting. The reason we use SpatiaLite is because of the spatial capabilities. So, the same way you can select census tracts with certain attributes, you can also select census tracts satisfying a certain spatial relationship with another layer. The reference for these spatial functions is available at <http://www.gaia-gis.it/gaia-sins/spatialite-sql-4.1.0.html>. As a simple example, what is the result of the following query?

```
SELECT GeometryType(geom) FROM geo_nyct_station_2011;
```

Now try:

```
SELECT statefp, countyfp, tractce, ST_Area(geom)
FROM geo_tract_2011;
```

The results are a little nonsensical because the unit of measurement for SRID 4269 is decimal degrees. Since degrees vary in size at different points on the globe, the area returned by the above query is not really meaningful. We need to transform the data, and in order to make it easier to work with the data we will save the transformed geometry, along with income data from the view we created previously, as a new spatial table. We will transform the geometry to SRID 2263, New York State Plane Long Island Zone. State Plane CRSes cover small areas (this zone only covers NYC and Long Island) and allow extremely precise measurement and calculation, but are unsuitable for displaying or operating on data from outside the zone.

```
CREATE TABLE geo_nyc_tract_income AS
SELECT
  PK_UID, STATEFP || COUNTYFP AS county_fips, TRACTCE AS tract, NAME AS name,
  NAMELSAD AS long_name, i.*, ST_Transform(geom, 2263) AS geom
FROM geo_tract_2011 JOIN vw_tract_household_income i USING (geoid)
WHERE STATEFP || COUNTYFP IN ('36005', '36047', '36061', '36081', '36085');
```

Now run the area calculation on this view:

```
SELECT geoid, ST_Area(geom) FROM geo_nyc_tract_income;
```

You will notice that the value in the Area column is much larger. That is because it is now being measured in the units of SRID 2263, which is feet.

Let's also add the data to QGIS. In QGIS go to **Layer→Add SpatiaLite Layer...** In the dialog box click New and select `nyc.sqlite` in the file chooser. Then click Connect, and you will see a list of spatial tables and views to add. Hold down the Ctrl key and select each of the following layers to add:

- geo_nyc_tract_income
- geo_nyct_station_2011
- geo_nyct_subway_lines_2010

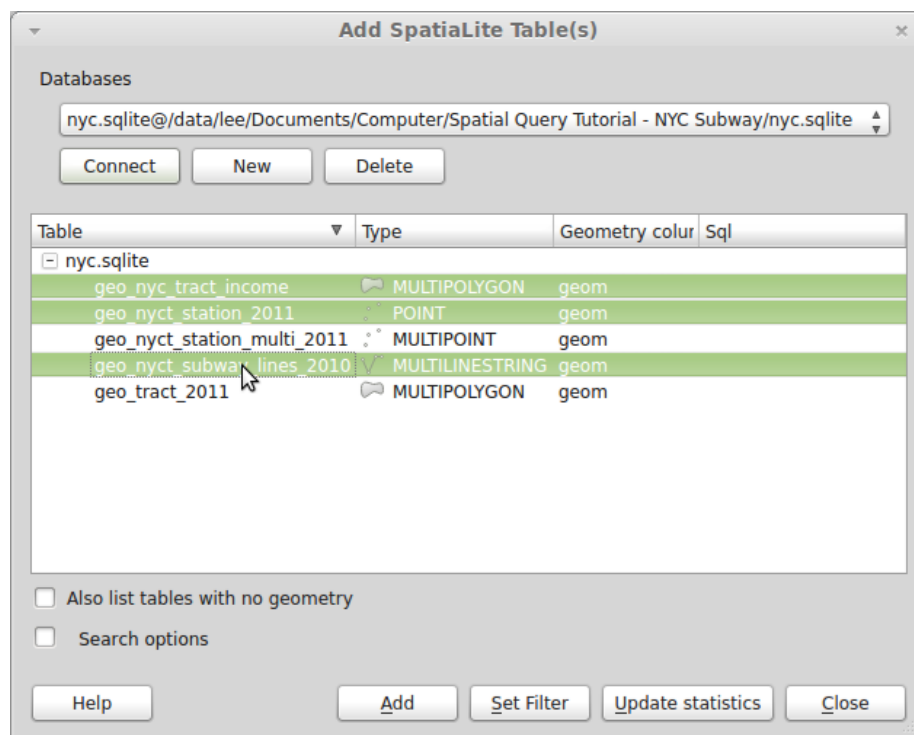


Figure 4: Add SpatiaLite Table(s)

Creating a Spatial View

We're almost at the point where we can perform some queries based on a spatial relationship between the Census tracts and the subway stations. But in order for the spatial functions to work correctly, the geometries need to be in the same coordinate reference system. So we will create a spatial view which will transform the geometries to the same 2263 SRID that we used for `geo_nyc_tract_income`.

```
CREATE VIEW vw_geo_nyct_station_2011 AS
SELECT
    ROWID, Division, Line, Station_Name, Station_Latitude, Station_Longitude,
    Route_1, Route_2, Route_3, Route_4, Route_5, Route_6, Route_7, Route_8,
    Route_9, Route_10, Route_11, ST_Transform(geom, 2263) AS geom
FROM geo_nyct_station_2011;
```

The view will not be recognized as a spatial view until it is registered in `views_geometry_columns`. This can be done manually or automatically. Manually, the following SQL will add the necessary entry:⁹

```
INSERT INTO views_geometry_columns
VALUES ('vw_geo_nyct_station_2011', 'geom', 'rowid', 'geo_nyct_station_2011', 'geom', 1);
```

Automatically, the view can be created and registered at the same time using QSpatialite or SpatiaLite GUI. Unfortunately, I find the SpatiaLite GUI Query/View Composer somewhat clunky, and it does not let you manually edit the SQL. QSpatialite allows you to enter arbitrary SQL (like the SpatiaLite GUI query pane) and then direct that result to QGIS, or create a spatial table or view. So open QSpatialite (a QGIS plugin) and paste *the SELECT statement only* (i.e., leave off the first line) into the query pane. Set the name of the table or (in this case) view to output, and the name of the geometry column (`geom`) in the input query.

This view can be treated as a table in your queries, without having to duplicate the underlying data. (One of the things I find annoying about working with desktop GIS is how many procedures require you to create a new shapefile in the process.)

More Complex Operations

Spatial Join

Now let's find the median income of the Census tract around each subway stations:

⁹Unfortunately, unlike other SQL parsers, SpatiaLite silently ignores any SQL query after the first one, so these two steps must be run separately.

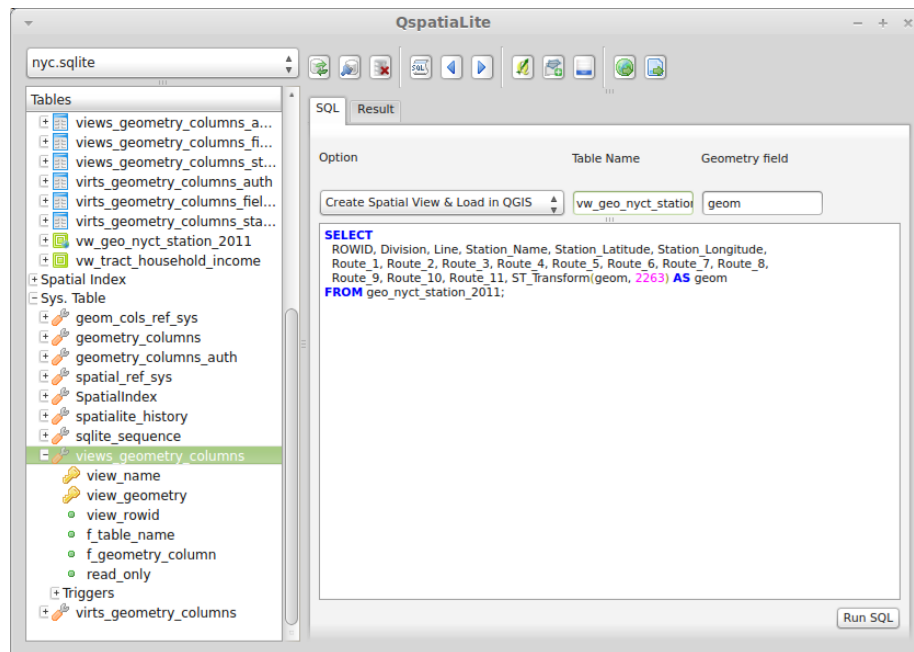


Figure 5: Create Spatial View with QSpatialLite

```
SELECT station_name, route_1, route_2, long_name, median_hh_income
FROM vw_geo_nyct_station_2011 s JOIN geo_nyc_tract_income t
ON (ST_Within(s.geom, t.geom))
```

I have aliased the table names so as to avoid having to retype them when I refer to them elsewhere in the query. By writing `geo_nyc_tract_income t`, I can then use `t` instead of `geo_nyc_tract_income` when I refer to the geometry column as `t.geom`. (This latter is necessary because both tables have a column named `geom`, so I have to specify which one is which, and for this function, we want to specify the subway stations layer first. If we asked for tracts within subway stations, the query would fail because a point can't contain a polygon.)

If we wanted to map this, we would have to add the table's geometry column. As we create more complex queries, display speed will slow down unless we materialize them in the database. So we could select **Create Spatial Table & Load in QGIS**.¹⁰ Set the (output) table name to `geo_nyct_station_median_hh_income`.

```
SELECT station_name, route_1, route_2, long_name, median_hh_income, s.geom AS geom
FROM vw_geo_nyct_station_2011 s JOIN geo_nyc_tract_income t
ON (ST_Within(s.geom, t.geom))
```

¹⁰In practice, I would probably add the `median_hh_income` column to the existing table and run this as an UPDATE query

When the layer loads, I will show you how to change the symbology so that each station is colored based on its median_hh_income value.

Buffering and Intersection

Let's assume we don't just want to know the median income of the tract the subway station falls in, but an average income of all households with a quarter-mile. What we need to do is construct a buffer around each station. What does this buffer look like? Run the following query in QSpatialite. Choose Load in QGIS as Spatial Layer.

```
SELECT rowid, station_name, ST_Buffer(geom, 1320) AS geom
FROM geo_nyct_station_median_hh_income;
```

The function `ST_Buffer()` accepts a geometry (point, line, or polygon) and a distance expressed in the SRID's measurements, so we used 1320 feet = 1/4 mile.

There are two ways to run this function. One is to create the buffer, and then intersect the buffer with Census tracts layer, to select those tracts which overlap the buffer. The other, more efficient way, is to use the `PtDistWithin()` function to select those tracts that fall within the specified distance. We then need to GROUP BY the station_name and construct some an aggregate expression to calculate the average income over all the tracts that satisfy the criteria.

```
SELECT station_name, sum(aggregate_hh_income) / sum(hh_total) AS avg_hh_income
FROM geo_nyc_tract_income t JOIN geo_nyct_station_median_hh_income s
  ON (PtDistWithin(t.geom, s.geom, 1320))
GROUP BY station_name;
```

As with the case of median income, we could output a new spatial layer with the average income, or update the existing layer, so that we could map the result.

Going Further

There was an extensive amount of documentation and tutorials related to earlier versions (2.x) of Spatialite, which has not really materialized for more recent versions. The old tutorials are still a good guide. See the Spatialite Cookbook here: <http://www.gaia-gis.it/spatialite-2.4.0-4/spatialite-cookbook/index.html>

Frank Donnelly, Geospatial Librarian at Baruch College, maintains the NYC Geodatabase, a Spatialite database stuffed full of New York oriented data. He also has a tutorial which introduces you to both the data set and the

kind of spatial query we've done in this workshop. Check it out here: http://www.baruch.cuny.edu/geoportal/nyc_gdb/

More MTA data is available here: <http://www.mta.info/developers/download.html>

For those wanting to learn more about QGIS, you can again use a practicum developed at Baruch, or a series of videos at Harvard. The Baruch practicum is almost a mini-textbook in GIS, with a lot of supporting text that explains the what and why of what you are doing in the software. The Harvard one has less explanation, but the video demonstration may be appealing to some.

- <http://www.baruch.cuny.edu/geoportal/practicum/>
- <http://maps.cga.harvard.edu/qgis/>

Also, keep an eye on my Open Geospatial Technologies blog at <http://geospatial.commonsgc.cuny.edu/>. Topics I expect to cover with some frequency include QGIS, PostGIS, R spatial, and geovisualization.

Credits

This exercise was adapted for New York City by [Lee Hachadoorian](#) from a lab exercise developed by Kurt Menke for the [GTCM](#)-aligned *Introduction to Open Source GIS*.

This work is licensed under a [Creative Commons Attribution-ShareAlike 3.0 United States License](#).

