

Introduction to the NYC Geodatabase (nyc_gdb) Open Source Version

Frank Donnelly, Geospatial Data Librarian, Baruch College CUNY

Aug 10, 2015

Abstract

This tutorial provides an introduction to the NYC Geodatabase (nyc_gdb), a resource for mapping and analyzing city-level features and data in GIS. The database comes in two formats: a Spatialite geodatabase built on SQLite that can be used in open source software like QGIS and the Spatialite GUI, and a personal geodatabase built on MS Access that can be used in ArcGIS. This document explains their content and structure (which are identical for both formats) and demonstrates how you can use them to explore and map data. Experience with using GIS software is presumed. For detailed metadata for all objects in the database and information about updates, see the document "NYC Geodatabase (nyc_gdb) Data Dictionary". The databases and associated documentation are available at <https://www.baruch.cuny.edu/confluence/display/geoportal/NYC+Geodatabase>.

This document contains a general overview of the database format and structure, specific instructions for using the open source SQLite / Spatialite database with QGIS and the Spatialite GUI, and an introductory tutorial for Spatialite.


Contents

1	Rights	1
2	Purpose	2
3	The Databases	3
3.1	Formats	3
3.2	Structure	3
4	Spatialite and QGIS	6
4.1	Adding Spatialite Data	7
4.2	Example: Mapping Spatialite Data	8
4.3	Spatial Queries in QGIS	10
5	Working With the Spatialite GUI	10
5.1	Basic Tasks	10
5.2	Spatial SQL	15
5.3	Extending the DB	21
5.4	Reference & Links	27

1 Rights

Disclaimer: Every effort was made to insure that the data, which was compiled from public sources, was processed accurately for inclusion in the NYC Geodatabase. The creator, Baruch College, and CUNY disclaim any liability for

errors, inaccuracies, or omissions that may be contained therein or for any damages that may arise from the foregoing. Users should independently verify the accuracy of the data for their purposes.

The database and associated documentation are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike license CC BY-NC-SA <http://creativecommons.org/licenses/by-nc-sa/4.0/> . You are free to share and to adapt the work as long as you cite the source, do not use it for commercial purposes, and release adaptations under the same license.

2 Purpose

The goals of the NYC Geodatabase project are:

- To provide new users with a resource for learning GIS and experimenting with data
- To provide intermediate users with a resource for expanding their GIS skills, into the realms of spatial SQL and database management
- To provide NYC users with a foundational dataset that they can build on for their specific research
- To provide all users with an example of an open source Spatialite database, to demonstrate its capabilities
- To build a foundation for creating additional GIS instructional programs at Baruch College, CUNY
- To create a basic dataset for in-house work at Baruch College, CUNY

The NYC Geodatabase (`nyc_gdb`) is a resource designed for basic geographic analysis and thematic mapping within the five boroughs of New York City. It contains geographic features and data compiled from several public sources. All of the features were transformed to share a common coordinate reference system (CRS) that is appropriate for the area: NAD 83 NY State Plane Long Island (feet); EPSG code 2263. Subsets of large features like water, greenspace, and public facilities were created and Census geographies like tracts, ZCTAs, and PUMAs were geoprocesed to create land-based boundaries. Census data from the 2010 Census, American Community Survey (ACS), and ZIP Code Business Patterns are stored in tables that can be easily related to geographic features. Transit and public facility point data were gathered from several city agencies and transformed into spatial data that can be used for reference or analysis for measuring distance, drawing buffers, or counting features within areas.

The database contains many foundational map layers and data that can be readily used, but was also constructed so users could build on that foundation and extend it for their own projects. All of the boundaries are based on the 2010 Census, which allows users to easily add additional layers from the Census TIGER files or to extend the study area beyond the five boroughs. The database also serves as an educational tool for introducing spatial databases and SQL.

The dataset is appropriate for thematic and reference mapping at a city and borough level and for thematic mapping at a sub-borough level. While it can be used for creating detailed reference maps at a sub-borough level, it is not the ideal choice for this purpose, given the degree of generalization in the TIGER Line files (in terms of the detail of the line work for the coast line and the number of water and landmark features selected for inclusion). Users will have to judge for themselves based on their intended purpose.

The database will be updated bi-annually: Census American Community Survey (ACS) data in the winter, Census ZIP Business Patterns and NYC transit data in the summer, and NYC public facilities in both winter and summer. Features created from the Census Bureau TIGER shapefiles (statistical areas and landmarks) and 2010 Census data are stable and won't be updated until after the 2020 Census.

The databases and documentation are available at <https://www.baruch.cuny.edu/confluence/display/geoportal/NYC+Geodatabase>.

3 The Databases

3.1 Formats

A geodatabase (or spatial database) is a relational database that has been enhanced to hold spatial objects or geographic features. The NYC Geodatabase (`nyc_gdb`) comes in two formats for use with different GIS software. The Spatialite version (`.sqlite`) is an open source format built on the SQLite database, and can be used in open source GIS software like QGIS. Spatialite also has its own command line and windows-based tools (the Spatialite GUI) which allow users to manipulate the data in a relational database environment. The personal geodatabase (`.mdb`) is a proprietary format created by ESRI; it is built on the Microsoft Access database and can be used in an ArcGIS environment. Even though each format is suited for a specific platform, this distinction is fading. ESRI began supporting the Spatialite format beginning with ArcGIS version 10.2, and personal geodatabases can at least be accessed with later 2.x versions of QGIS.

Both formats give users a better way to organize and structure their data relative to shapefiles and individual data tables, as multiple features and attributes can be stored in a single database file and can be easily related to each other. Both formats are simple, file-based databases that can be created, copied, and distributed easily. Unlike enterprise-level databases (i.e. ArcGIS enterprise geodatabases, PostGIS), file sizes are functionally limited to a few gigabytes, and individual user permissions cannot be specified. Desktop databases are not an ideal choice for data that must be accessed simultaneously from many computers over a network filesystem.

The open source Spatialite database has the added advantage of allowing users to perform spatial queries in addition to regular SQL queries. Examples include calculating areas and distance and evaluating geographic relationships like adjacency and overlap. Thus, Spatialite is able to extend the geographic selection and analysis capabilities of open source GIS, which is still developing these capabilities. In contrast, the capabilities of the MS Access personal geodatabase are limited; in the proprietary world the ArcGIS software does most of the heavy lifting via the ArcToolbox, and the database serves as a simple container for organizing and storing data.

Both geodatabases can also be accessed and manipulated using regular relational database tools, like MS Access or the SQLite Manager (available as a Firefox plugin), but these tools can only be used for traditional SQL queries and nothing spatial. When a geodatabase is created each respective program (Spatialite GUI or ArcGIS) populates the new database with tables and relations that manage and support any geographic features that are inserted. When working with the database care must be taken to not alter or remove these tables.

3.2 Structure

Objects in the database are categorized and named with a prefix to differentiate different types of features. A brief list of the database objects is provided below; for full details consult the `nyc_gdb` Data Dictionary. In most instances normal form for relational databases is violated in order to provide a resource that is readily usable for mapping and analysis; names and codes are repeated in some tables to facilitate identification and selection, and percent values are pre-calculated and provided with totals. This is particularly valuable for the American Community Survey data, where calculating margins of error for derived values like percentages is a difficult task that's cumbersome to perform within a relational database.

3.2.1 A Tables

Objects that begin with the prefix "a" are geographic features that represent points, lines, and areas. The census statistical areas are designed so that they can be joined with "b" tables that contain census data, and all census area features are generalized so that they represent land areas. Public facility point features contain identifying information like names and addresses as well as one variable, like capacity or ridership, that can be measured or mapped; features from the NYC Dept of City Planning's Facilities dataset are taken "as is" and are not verified for accuracy or omissions.

Each table contains a unique identifier.

All "a" tables have a column called "bcode" that indicates which borough the feature is in, to facilitate select queries. The bcode is the US Census ANSI / FIPS code for the county.

- 36005 - Bronx County (Bronx)
- 36047 - Kings County (Brooklyn)
- 36061 - New York County (Manhattan)
- 36081 - Queens County (Queens)
- 36085 - Richmond County (Staten Island)

a_boroughs : The five boroughs of NYC, from the census counties file

a_colleges : From the NYC Dept City Planning Facilities database

a_facilities : Selection of airports and other large public facilities from the census landmarks file

a_greenspace : Selection of large parks, wildlife areas, and cemeteries from the census landmarks file

a_hospitals : From the NYC Dept City Planning Facilities database

a_libraries : Public libraries from the NYC Dept City Planning Facilities database

a_metro_counties : Counties in the NYC Metropolitan CSA, from the census counties file

a_path_stations : NYC PATH Stations from NJ Transit with ridership data from PANYNJ

a_pumas2010 : Public Use Microdata Areas; census statistical areas designed to have approx 100k residents. Boundaries from the 2010 Census were used for the first time in the 2012 ACS

a_roads : All roads in NYC from the census roads file

a_schools_private : From the NYC Dept City Planning Facilities database

a_schools_public : From the NYC Dept City Planning Facilities database

a_subway_complexes : Single or multiple stations with shared entrances and passages where riders can freely transfer, and for which the MTA publishes ridership statistics

a_subway_complexes_srvnotes : Notes on service disruptions that impact the ridership statistics in a_subway_complexes

a_subway_stations : Individual stations represented by distinct platforms for specific trains

a_tract_popcenters : Population centers / centroids for census tracts based on the 2010 Census. Represents the center of the population's distribution within each tract

a_tracts : 2010 census tracts; census statistical areas designed to have an ideal size of 4,000 residents, with a range of 1,200 to 8,000. Tracts can be aggregated to Neighborhood Tabulation Areas (NTAs) created by the City

a_train_stations : LIRR and Metro North stations in NYC, from the MTA

a_water_coastal : Selection of major coastal water from the census water file, used to create land boundaries for the census layers

a_water_lakes : Selection of major lakes from the census water file

a_zctas : 2010 ZIP Code Tabulation Areas; census statistical areas created by aggregating census blocks based on postal addresses, to create geographic approximations of USPS ZIP Codes

3.2.2 B Tables

Objects that begin with the prefix "b" are non-spatial data tables from the US Census, with data reported as values and percentages. These tables can be joined to geographic "a" features so that quantities can be mapped and evaluated spatially. The unique identifier field for the 2010 Census and ACS "b" tables is "GEOID2", which is the census FIPS code for that area. The unique ID for the Business Patterns tables is "ZCTA5", the five-digit Census ZCTA number. Tables are named based on their geography, dataset, and year. Column names are codes that uniquely identify each variable. For each dataset there is an index table named with the suffix "lookup", that relates column codes to variable names.

The American Community Survey (ACS) is an ongoing sample survey of the population that's tabulated annually for 1, 3, and 5-year periods. The values are published as estimates with a 90% confidence interval and margins of error (+/-). In this database, data from the ACS are from the 5-year series; the year indicates the year of release and final year of the estimate range (i.e. 2012 represents 2008-2012 5-year data). There are two data tables for each geography that represent a subset of the four demographic profiles (tables DP02 through DP05). Each individual variable is named based on its subject and consists of four adjacent columns in this order: the estimate itself (identified by the suffix "E"), a margin of error for the estimate (suffix "M"), a percent total (suffix "PC"), and a margin of error for the percent total (suffix "PM"). The lookup table correlates the column heading with the variable name. This data is updated annually.

- b_YEARacs_lookup
- b_pumas_YEARacs1
- b_tracts_YEARacs1
- b_zctas_YEARacs1
- b_pumas_YEARacs2
- b_tracts_YEARacs2
- b_zctas_YEARacs2

The decennial census is a 100% count of the population taken on April 1st. Data from the 2010 Census represents all the data in the demographic profile (table DP01). Each variable has two values: the actual count (named with the prefix HD01) and a percent total (named with the prefix HD02). The percent totals are stored in a separate table with the suffix "pct". The lookup table correlates the column headings (created by the Census) with the variable names, while the footnotes table contains footnotes referenced for certain variables in the index. This data will not be updated until the 2020 Census. Decennial census data is not tabulated at the PUMA level.

- b_2010census_lookup
- b_2010census_footnotes
- b_tracts_2010census
- b_tracts_2010census_pct
- b_zctas_2010census
- b_zctas_2010census_pct

The Census Bureau compiles the ZIP Code Business Patterns (ZBP) data from the Business Register, which contains a record for each business establishment with paid employees in the US; an establishment is a single physical location at which business is conducted or services or industrial operations are performed. ZBP data is stored in two tables: the "emp" table provides the total number of establishments, employees, and payroll (for the first quarter and annually in \$1,000s of dollars) and the "ind" table provides a count of establishments based on type of business, as classified by the North American Industrial Classification System (NAICS). The names that are correlated with the two-digit NAICS sector codes are in the "indcodes" table. The records in these tables represent US Census ZCTAs and *not* USPS ZIP Codes. The data was aggregated from ZIP Codes (as published in the ZBP) to ZCTAs. A table that cross-walks ZIP Codes to ZCTAs is included for user reference. This data is updated annually.

- b_zctas_YEARbiz_emp
- b_zctas_YEARbiz_ind
- b_zctas_YEARbiz_indcodes
- b_zips_to_zcta

3.2.3 C Tables

Objects that begin with the prefix "c" are geographic features that represent the actual boundaries for census statistical areas. Other than transforming the projection to match the other database features, these features and their attributes have not been altered in any way from the original TIGER shapefiles from the Census. They are included in case the user wishes to depict the actual boundaries (that encompass land and water) for reference. They should not be used for mapping census data.

- c_bndy_boroughs
- c_bndy_metro_counties
- c_bndy_pumas2010
- c_bndy_tracts

3.2.4 X Tables

Objects that begin with the prefix "x" are "extra" geographic features that represent the original source data for some of the "a" features. The "x" features are included in case the user wants to add additional features that are not part of the generalized "a" layer.

x_landmarks : this layer was used to create the facilities and greenspace layer and includes all 2010 Census landmarks in NYC

x_nad83_boroughs : this borough layer does not share the same coordinate reference system as the other layers in the database; it is in simple NAD 83. It is included to provide a frame of reference for users who need to plot latitude and longitude data

x_water : this layer was used to create the coastal and lakes water layers and includes all the 2010 Census water layers in the greater metro area

3.2.5 Other Tables

"d_ntas_2010census" is not a table, but a view that is included for the sake of example. It joins the a_tracts layer to the b_tracts_2010census table and groups basic population and housing data by Neighborhood Tabulation Areas defined by the City.

"z_metadata" is a table that describes the name and source of all of the tables in the database, along with the year that the feature or table was last updated.

All other objects are core parts of the geodatabase designed to manage and support geographic features. The table names will differ between the MS Access and SQLite versions. These tables should not be removed or altered, otherwise the database could be rendered unusable.

4 Spatialite and QGIS

QGIS is free, open source, cross-platform GIS software, available for download from <http://www.qgis.org/>. For an introduction to QGIS try the GIS Practicum at <https://www.baruch.cuny.edu/confluence/display/geoportal/GIS+Practicum>.

With QGIS (2.x) you can use a Spatialite database to:

- Add, view, and symbolize geographic features and tables

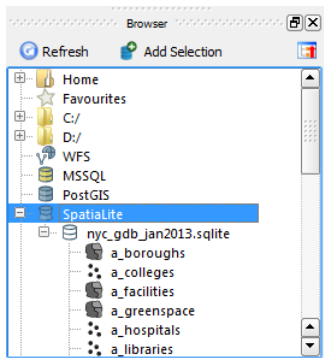
- Layer Spatialite features with other spatial files (like shapefiles)
- Use QGIS selection, analysis, and geoprocessing tools
- Create new shapefiles or tables from Spatialite layers
- Edit Spatialite layers by modifying or adding features and attributes
- Create basic SQL and spatial SQL queries
- Create basic spatial views stored within the project (selections with criteria, table joins)
- Create new Spatialite databases (using Browser)
- Export Single vector file out as new Spatialite database (using Save As)
- Import and export vector layers to an existing Spatialite database (using the DB Manager)


There are also 3rd party plugins specifically designed for working with Spatialite databases within QGIS.

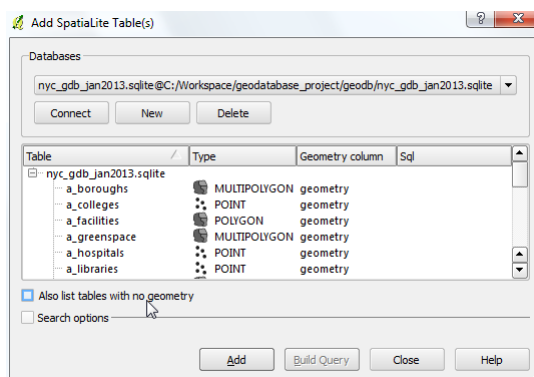
4.1 Adding Spatialite Data



There are a few different methods for adding Spatialite data to QGIS:

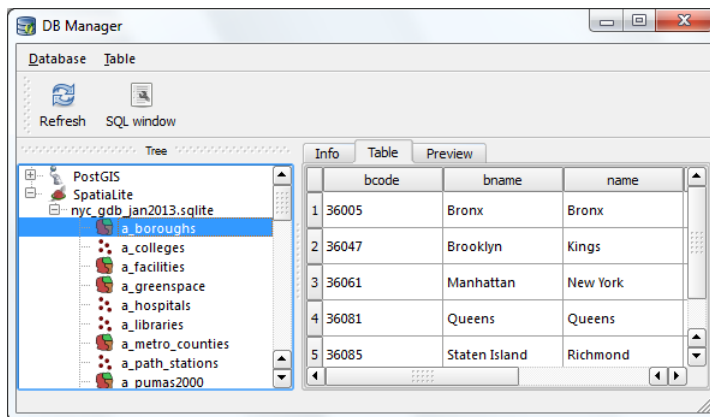
Browser : The data browser allows you to view your entire file system within QGIS so you can easily drag and drop features into a project. The browser allows you to connect to databases by selecting the database type and browsing to its location to establish a connection. You can see and add Spatialite geographic features, but not tabular ones. If the browser is not visible, you can activate it by right-clicking on an empty area of the toolbars and checking Browser.



Add Spatialite Layer Button : This button  is on the Layers toolbar. Press it and browse through your file system to connect to the database. Once connected you can add geographic and tabular features to your project.



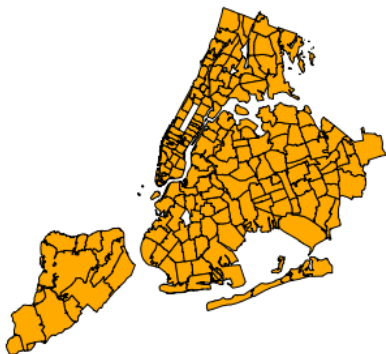
DB Manager : This plugin allows you to access a geodatabase and interact with it in a separate window - you can preview tables and geometry and drag and drop both into your project. You can also run SQL and spatial SQL queries in the query window, and create temporary spatial queries or views that you can view in the project window (such as selecting a subset of records that meet a certain criteria and viewing the features as a new layer). To activate the plugin go to Plugins > Manage Plugins and select the DB Manager. This will add a button  to the toolbar and a Database entry on the file menu. You must connect to your database using the browser or the  toolbar button before accessing it via the manager.




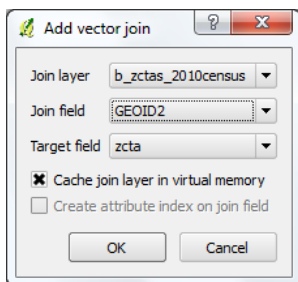
4.2 Example: Mapping Spatialite Data

The following example illustrates how to add Spatialite features, join data tables to features, and map data in QGIS. These instructions were written using QGIS 2.4 in an MS Windows environment, and will vary when using other operating systems (i.e. Mac laptop users should substitute "right-click" with a two-fingered single click) or other versions of QGIS.

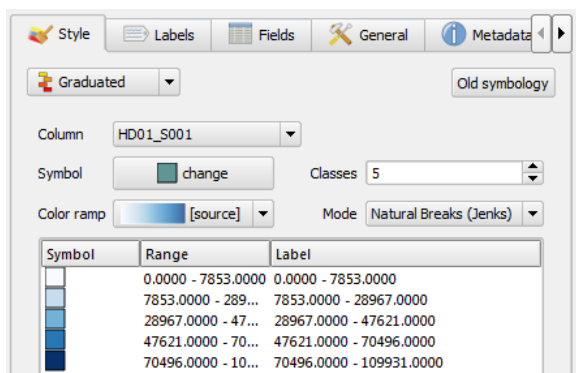
1. Launch QGIS. In the browser select SpatialLite, right click, and Choose new connection. Browse to the folder where you've placed the database, select it, and hit OK. Back in the browser, you can now see the database and can hit the plus symbol to expand the database listing. You should see a list of all the geographic features in the database.
2. Select the a_zctas layer and drag it into the project window. This adds the ZCTAs to the project.



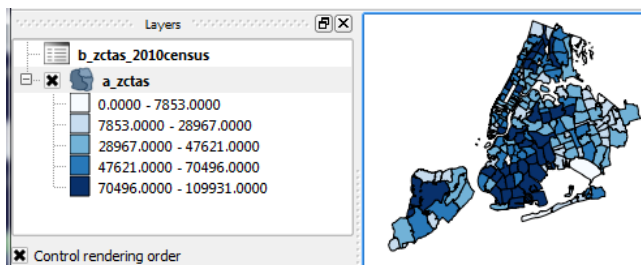
- Hit the Database Manager  button. Expand the Spatialite DB tree by hitting the plus sign, then expand the nyc_gdb database to see its contents. Scroll through the list to the tables and select b_zctas_2010census. Add it to the project by selecting the table, right click, and choose the option that says Add to QGIS canvas. Close the DB Manager.
- Doubleclick on the a_zctas layer in the layers list to open the Layer Properties menu (make sure you're in the project layers list and not the browser - you can tab between each one). Select the Joins tab. Hit the green plus symbol. In the Add vector join menu choose b_zctas_2010census as the join layer, GEOID2 as the join field, and zcta as the Target field. Hit OK to save the join. GEOID2 and zcta are the unique identifiers in the table and zcta features respectively; they both contain the 5 digit ZCTA number (you could verify this by viewing the table and attribute table of the features).



- In the Layer Properties menu switch to the Style tab. Change the symbol dropdown from Single Symbol to Graduated. The default Column is HD01_S001, which is the total 2010 population. In the dropdown menu on the right change the Mode to Natural Breaks. Choose a color ramp. Hit Classify, then hit OK.




- Expand the a_zctas layer in the layers menu to see the classification categories. Select the layer name, right click, and check Show Feature Count to display the number of features in each class.



7. If you save the project QGIS will remember to add these objects and join them each time you open the project. We know that HD01_S001 is total population; we can verify this and see what the other columns are by viewing the b_2010census_lookup table within the DB Manager. Instead of mapping totals we could map percentages stored in the b_zctas_2010census_pct table. To fill in the white space and cover non-residential areas we could add the a_facilities and a_greenpace features and layer them over top of the ZCTAs (selecting and dragging the items in the layer list changes their drawing order).

NOTE: When joining tabular data to geographic features in this manner, the join is not permanent and will only be saved within the project. This is fine for simply visualizing and mapping data. You can permanently fuse tabular data to geography by creating a new feature. You can do this by creating a new table / feature in Spatialite and populating it with a query (demonstrated in the next chapter), or you can use QGIS to save the joined geography and data as a new shapefile.

4.3 Spatial Queries in QGIS

You can use the SQL Window in the Database Manager  to perform many of the SQL and Spatial SQL queries that are demonstrated in the next chapter on the Spatialite GUI. The benefit of using QGIS is that you can see the visual result of your queries, and you can do your work within one application. However, the benefit of the Spatialite GUI is that it's designed specifically for working with databases, and it's better for creating well-designed databases that follow standard rules, where you can specify keys and data types for new features and tables. It's also a small, light-weight application that makes a nice addition to your GIS toolkit - so it's worth learning.

5 Working With the Spatialite GUI

The Spatialite GUI is free, open source, cross-platform software that allows you to work with a geodatabase in a relational database environment and to conduct spatial SQL queries. For MS Windows users the simplest way to get it up and running is to download a pre-compiled binary from the home page at <http://www.gaia-gis.it/gaia-sins/>. Follow the link for the stable version (32 or 64 bit), and then select the latest gui version to download. As it's a binary there's nothing to install; the application is ready to use. For Linux users, Spatialite is included in the repositories of many distros, so take a look in your package handler and see. Alternatively, Windows, Linux, and Mac users can download and build the application from source, which is a little more involved but not too painful: https://www.gaia-gis.it/fossil/spatialite_gui/index.

This tutorial covers Spatialite 4.1.1 using the Spatialite GUI 1.7.1. The nyc_gdb transitioned to this version in July 2014. Most (*but not all*) of the features and functions demonstrated here will also work with the older version of the software used in previous versions of the nyc_gdb (2.4.0 RC-4 using GUI 1.4.0).

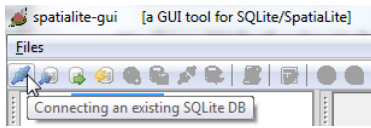
This section demonstrates basic SQL and spatial SQL that you can use when working with the database. For a fuller treatment of SQL with Spatialite visit the Spatialite tutorials, cookbook, and function reference guide at <https://www.gaia-gis.it/fossil/libspatialite/index>. The extensive Spatialite Cookbook was written for an earlier version of the software, but most of the examples still apply and it continues to be the premier Spatialite tutorial. There are a number of good tutorials on non-Spatial SQL with SQLite, such as <http://zetcode.com/databases/sqlitetutorial/>, as well as a few good books - *Using SQLite* by Jay Kreibich provides a clear intro for new database users as well as a thorough reference. The complete reference for SQLite syntax is available at: <https://www.sqlite.org/lang.html>.

5.1 Basic Tasks

5.1.1 Connect to database

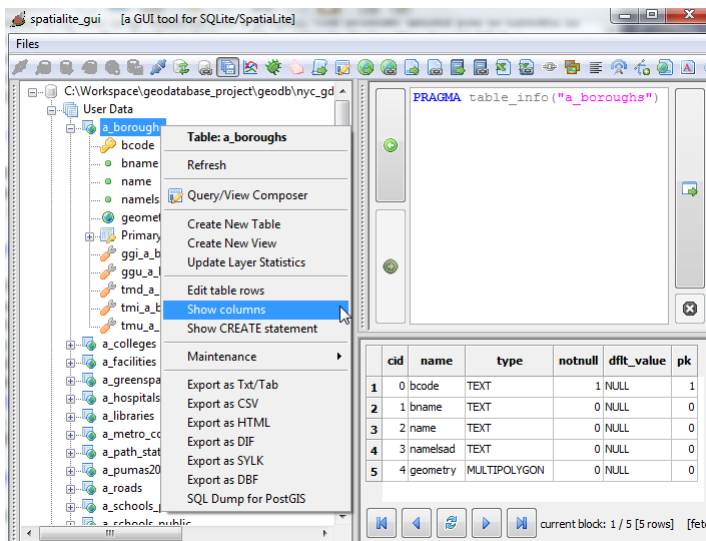
Launch the Spatialite GUI. Hit the Connect to database button, browse through your file system to where the database is stored, and select it. Whenever you subsequently launch Spatialite it will automatically connect to this database,

unless you explicitly hit disconnect on the toolbar.



5.1.2 View table metadata

The core features of the nyc_gdb are grouped under the "User Data" heading, while the internal components of the spatial database are grouped under subsequent headings. In the menu tree on the left hit the plus symbol beside User Data and then a_boroughs to expand and display its attribute list. The primary key that uniquely identifies each record looks like a key, and the column where geometry is stored looks like a globe. Click on a_boroughs and right click to open a menu of options. Choose the Show Columns option. This lists each column, its data type, and any constraints (i.e. whether null values are permitted, what the primary key is, etc).



You can also use this menu to edit individual rows in a table, add new columns, change the name of a table, and delete tables. You cannot delete columns or change the name or data type of columns; this is something that SQLite does not support (more on this later). These menu options are shortcuts that save you the step of having to enter SQL code; you could type the code yourself in the SQL window. For example, the code for viewing the table metadata for the boroughs feature:

```
1 PRAGMA table_info (a_boroughs)
```

5.1.3 View table records

To view the actual records in a table you have to execute a SELECT query. To view the records in the boroughs table type the following into the SQL box, then execute the code by hitting the large Execute SQL Statement button to the right of the SQL window.

```
2 SELECT *  
FROM a_boroughs
```

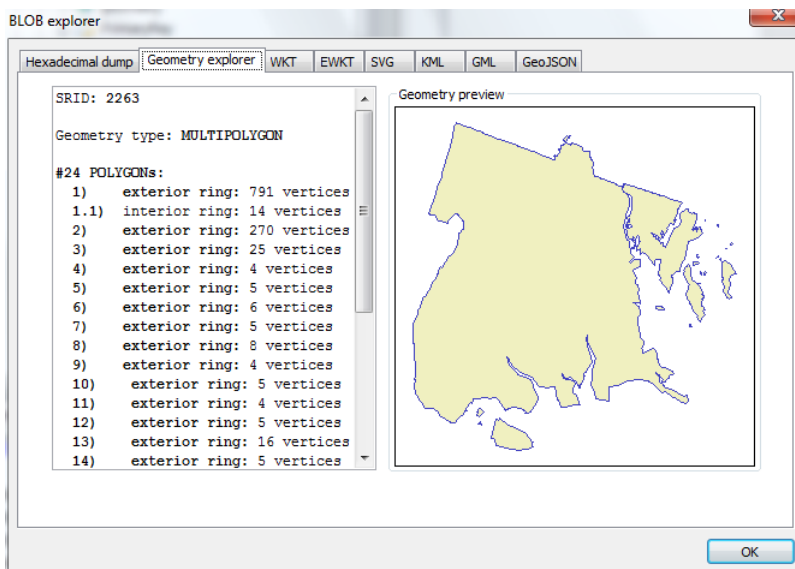
	bcode	bname	name	namelsad	geometry
1	36005	Bronx	Bronx	Bronx County	BLOB sz=21452 GEOMETRY
2	36047	Brooklyn	Kings	Kings County	BLOB sz=32270 GEOMETRY
3	36061	Manhattan	New York	New York County	BLOB sz=17402 GEOMETRY
4	36081	Queens	Queens	Queens County	BLOB sz=42097 GEOMETRY
5	36085	Staten Island	Richmond	Richmond County	BLOB sz=17544 GEOMETRY

Navigation icons: Home, Previous, Refresh, Next, End. Status: current block: 1 / 5 [5 rows] [fetched in 00:00:00.035]

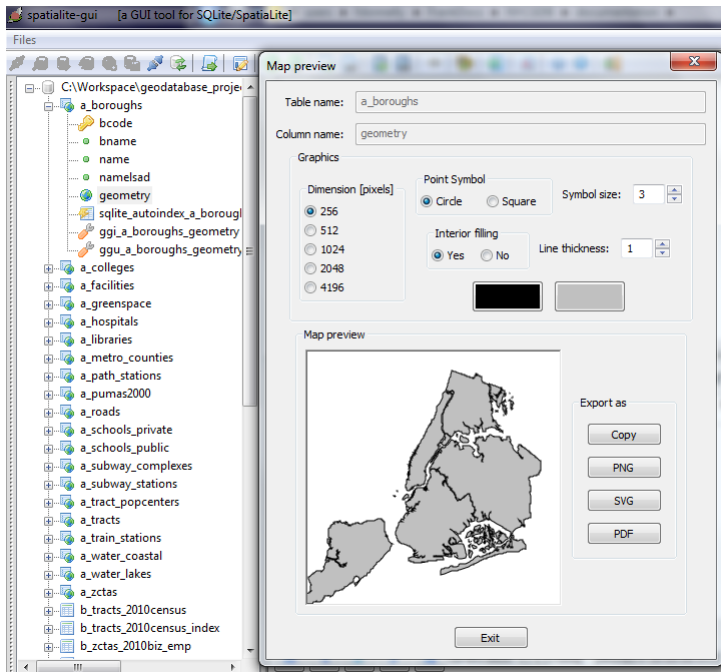
NOTE - executing the SELECT * statement on tables with a large number of columns or records (for example, census tract data tables) may take time to process. If you simply want to preview records in a large table, you can add the LIMIT clause to the end of your statement and specify the maximum number of records to view (i.e. LIMIT 50 or LIMIT 100).

5.1.4 Preview table geometry

In the table results window (underneath the SQL window), in the geometry column for the Bronx, click on the actual data value that says BLOB sz=21452 GEOMETRY to activate it, then right click on it and choose BLOB Explore. In the BLOB Explorer menu that appears the default is the Hexadecimal tab - click on the Geometry Explorer tab to see a preview of the feature and details about the geometry. You can select other tabs to preview the data in a number of standard text-based formats for encoding vector data. Hit OK to close the window.



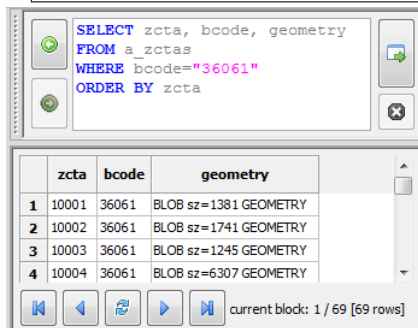
You can also preview an entire layer. In the object tree on the left, expand the columns for the boroughs. Select the geometry column, right click, and chose the Map Preview option The preview gives you the option to export a picture of the layer out as an image. Hit Exit to close the window.



5.1.5 Basic select query

Let's perform a basic, non-spatial SQL query. Select all ZIP Code Tabulation Areas in Manhattan and order them by ZCTA number. We'll use the bcode column, which contains the census ANSI / FIPS code for each borough. Type the code into the SQL box and hit execute to see the results.

```
3 SELECT zcta, bcode, geometry
   FROM a_zctas
   WHERE bcode="36061"
   ORDER BY zcta
```



You can also do aggregate queries. This will count the number of ZCTAs by borough:

```
4 SELECT bcode, COUNT(zcta) AS zctas
   FROM a_zctas
   GROUP BY bcode
```

	bcode	zctas
1	36005	25
2	36047	38
3	36061	69
4	36081	69
5	36085	13

5.1.6 Identify census variables

In order to work with the census data in the "b" tables, we need to identify the variable that we want and find the code that identifies that column. Let's say we are interested in total population by ZCTA for the 2010 census. First, we need to look in the lookup table for the 2010 Census:

```
5 SELECT *
   FROM b_2010census_lookup
```

	count_id	count_name
1	GEOID	Id
2	GEOID2	Id2
3	GEOLABEL	Geography
4	HD01_S001	Number; SEX AND AGE - Total population
5	HD01_S002	Number; SEX AND AGE - Total population - Under 5 years
6	HD01_S003	Number; SEX AND AGE - Total population - 5 to 9 years

The lookup table tells us that the total population for all sexes and ages is identified by the code HD01_S001. If we scroll to the right of the table we'll see the column IDs for the percent totals. For the 2010 Census data, codes that begin with the prefix HD01 indicate that the value is a number. Percent totals, which begin with the prefix HD02, are stored in a separate "pct" table. The suffix, S001, indicates the variable number. HD01_S001 and HD02_S001 are the total population and percent total respectively, HD01_S002 and HD02_S002 are the total population under 5 years of age and the percent total under five years of age, etc.

Now that we know that HD01_S001 holds total population, we can query the ZCTA 2010 Census table for that value and sort the results from most to least populated. Using AS allows us to create an alias for cryptic column names. The first three columns in the table are standard geographic identifiers; GEOID2 is always the unique identifier in the 2010 Census and ACS "b" tables.

```
6 SELECT geoid, geoid2, geolabel, HD01_S001 AS pop2010
   FROM b_zctas_2010census
  ORDER BY pop2010 DESC
```

	GEOID	GEOID2	GEOLABEL	pop2010
1	8600000US11368	11368	ZCTA5 11368	109931
2	8600000US11226	11226	ZCTA5 11226	101572
3	8600000US11373	11373	ZCTA5 11373	100820

5.1.7 Table join

Let's query the ZCTAs in Manhattan that have a population that's greater than 200 people, to eliminate ZCTAs that are non-residential (encompassing large office buildings and public facilities). To do that we'll need to join our ZCTA census table to our ZCTA geographic features, which have the borough code field. There are two different ways to

write a JOIN statement; you can be implicit:

```
7 SELECT zcta, bcode, HD01_S001 AS pop2010
   FROM a_zctas, b_zctas_2010census
   WHERE zcta=geoid2 AND bcode="36061" AND pop2010 > 200
   ORDER BY pop2010
```

Or explicit:

```
8 SELECT zcta, bcode, HD01_S001 AS pop2010
   FROM a_zctas AS Z
   JOIN b_zctas_2010census AS C
   ON (Z.zcta = C.geoid2)
   WHERE bcode="36061" AND pop2010 > 200
   ORDER BY pop2010
```

The result is the same:

	zcta	bcode	pop2010
1	10162	36061	1685
2	10006	36061	3011
3	10004	36061	3089
4	10282	36061	4783

current block: 1 / 45 [45 rows]

In this example our two tables did not share any column names in common. If they did, we would have to explicitly say which columns the tables were in using the convention tablename.column, and we would want to use aliases to keep things brief:

```
9 SELECT a_zctas.zcta AS zcta, a_zctas.bcode AS bcode,
   b_zctas_2010census.HD01_S001 AS pop2010
   FROM a_zctas, b_zctas_2010census
   WHERE zcta=b_zctas_2010census.geoid2 AND bcode="36061" AND pop2010 > 200
   ORDER BY pop2010
```

5.2 Spatial SQL

5.2.1 Retrieve Coordinates

We can use the geometry column in the subway features to find the northernmost station in the subway system:

```
10 SELECT stop_name, trains, bcode
   FROM a_subway_stations
   WHERE MbrMaxY(geometry) IN (
   SELECT Max(MbrMaxY(geometry))
   FROM a_subway_stations)
```

	stop_name	trains	bcode
1	Wakefield-241 St	2	36005

We can alter this statement to get the nth most point for each cardinal direction:

- MbrMaxY and Max for northernmost
- MbrMinY and Min for southernmost
- MbrMaxX and Max for easternmost
- MbrMinX and Min for westernmost

How did that work? Spatial SQL utilizes the geometry of spatial features to perform a variety of geographic operations. The geometry is stored in the geometry column as a binary object, or BLOB in database-speak. If we convert that object into human-readable form, we would recognize the data as a series of coordinates. We can extract the coordinates of subway stations from the geometry column:

```
11 SELECT stop_name, trains, bcode, ST_AsText(geometry) AS coordinates
    FROM a_subway_stations
```

	stop_name	trains	bcode	coordinates
1	Van Cortlandt Park-242 St	1	36005	POINT(1012291.155985 263271.208046)
2	238 St	1	36005	POINT(1011660.704076 261601.441833)
3	231 St	1	36005	POINT(1010566.908282 259483.047364)
4	Wakefield-241 St	2	36005	POINT(1025543.987933 268346.145574)
5	Nereid Av	2 5	36005	POINT(1024508.570441 266615.235549)

Since our features are projected in NY State Plane, the coordinates that are returned are from that system, and the XY units are represented in feet. If we wanted something more familiar, like longitude and latitude, we can transform the coordinates to a general coordinate system like NAD 83:

```
12 SELECT stop_name, trains, bcode,
    ST_AsText(Transform(geometry,4269)) AS coordinates
    FROM a_subway_stations
```

	stop_name	trains	bcode	coordinates
1	Van Cortlandt Park-242 St	1	36005	POINT(-73.898583 40.889248)
2	238 St	1	36005	POINT(-73.90087 40.884667)
3	231 St	1	36005	POINT(-73.904834 40.878856)
4	Wakefield-241 St	2	36005	POINT(-73.85062 40.903125)
5	Nereid Av	2 5	36005	POINT(-73.854376 40.898379)

4269 is the EPSG code that uniquely identifies the NAD 83 coordinate system. The database's library of coordinate systems is stored in the table spatial_ref_sys, which is automatically created when the database is created. The geometry in each table is tied to a specific coordinate system stored in the database. If you wanted to look up the definition there's a button on the toolbar that allows you to search the spatial reference table by name or EPSG code; alternatively you could write the statement yourself:

```
13 SELECT *
    FROM spatial_ref_sys
    WHERE srid = 4269
```

	srid	auth_name	auth_srid	ref_sys_name	proj4text
1	4269	epsg	4269	NAD83	+proj=longlat +ellps=GRS80 +datum=NAD83 +no_defs

We can also determine which coordinate systems each of our layers are in by querying the geometry_columns table (EPSG 2263 is NY State Plane Long Island feet):


```
14 SELECT *
    FROM geometry_columns
```

	f_table_name	f_geometry_column	geometry_type	coord_dimension	srid	spatial_index_enabled
1	a_boroughs	geometry	6	2	2263	0
2	a_metro_counties	geometry	6	2	2263	0
3	a_facilities	geometry	3	2	2263	0
4	a_greenpace	geometry	6	2	2263	0

Looking up the coordinates for point features is pretty straightforward. We could do the same for lines and polygons, but the output will be much longer as the coordinates for every node will be listed; if the feature is very large or complex the database won't display the coordinates at all. If we wanted a frame of reference for polygon features, we could return bounding box coordinates instead. Bounding boxes, also known as minimum bounding rectangles (MBRs), are drawn around features to enclose them entirely and represent their minimum extent; the four sets of coordinates represent the corners of the box. This will give us the bounding box of each borough in longitude and latitude:

```
15 SELECT bname, bcode, ST_AsText(Transform(ST_Envelope(geometry),4269)) AS bbox
    FROM a_boroughs
```

	bname	bcode	bbox
1	Bronx	36005	POLYGON((-73.933453 40.785787, -73.764908 40.785567, -73.76445 40.915076, -73.933324 40.915297, -73.933453 40.785787))
2	Brooklyn	36047	POLYGON((-74.042457 40.570097, -73.833564 40.569985, -73.833142 40.73895, -74.042565 40.739063, -74.042457 40.570097))
3	Manhattan	36061	POLYGON((-74.04728 40.683912, -73.907264 40.683884, -73.906992 40.879275, -74.047419 40.879302, -74.04728 40.683912))
4	Queens	36081	POLYGON((-73.962808 40.549663, -73.701124 40.54928, -73.699996 40.800638, -73.962668 40.801023, -73.962808 40.549663))
5	Staten Island	36085	POLYGON((-74.25585 40.495909, -74.051333 40.496182, -74.051451 40.649483, -74.256438 40.649209, -74.25585 40.495909))

As a shorthand reference for linear features you can retrieve the start and end point of a line using ST_StartPoint and ST_EndPoint on the geometry column. Unlike the previous examples, the coordinates would be retrieved in separate fields, one set for the beginning and one for the end.

5.2.2 Calculate distances

We can select all the subways that are within 1/2 mile of ZCTA 10010:

```
16 SELECT stop_name, trains,
    ST_DISTANCE(a_zctas.geometry, a_subway_stations.geometry) AS dist
    FROM a_zctas, a_subway_stations
    WHERE zcta = "10010"
    AND dist <= 2640
    ORDER BY dist
```

	stop_name	trains	distance
1	23 St	6	0.000000
2	23 St	FM	0.000000
3	23 St	NR	0.000000
4	28 St	6	150.952836
5	28 St	NR	329.946622
6	23 St	1	899.626723

Distance represents the minimum distance, and for polygons distances are measured from the edge to the nearest point; stations located inside the polygon are selected and assigned a distance of zero. Since the coordinate reference system for our features is in feet, our input and output units are in feet (.5 miles = 2,640 feet). If we want to measure

from the center of a polygon rather than the edge, we just calculate the centroid first:

```
17 SELECT stop_name, trains,  
ST_Distance(ST_Centroid(a_zctas.geometry), a_subway_stations.geometry) AS dist  
FROM a_zctas, a_subway_stations  
WHERE zcta = "10010" AND dist <= 2640  
ORDER BY dist
```

	stop_name	trains	dist
1	23 St	6	1246.529939
2	28 St	6	1570.913475
3	23 St	NR	2135.758565
4	3 Av	L	2501.604854
5	33 St	6	2563.851270
6	14 St-Union Sq	N Q R	2605.844989

We can also round the result by wrapping the ROUND statement around our calculation: the ROUND statement goes at the beginning and the number of decimal places goes at the end:

```
18 SELECT stop_name, trains,  
ROUND(ST_Distance(ST_Centroid(a_zctas.geometry), a_subway_stations.geometry),1) AS  
dist  
FROM a_zctas, a_subway_stations  
WHERE zcta = "10010" AND dist <= 2640  
ORDER BY dist
```

5.2.3 Evaluate geographic relationships

Spatialite allows you to evaluate several geographic relationships between features: Equals, Disjoint, Touches, Within, Overlaps, Crosses, Intersects, Contains, and Relate. (for a good explanation of these see http://workshops.opengeo.org/postgis-intro/spatial_relationships.html). For example, instead of measuring the distances of subway stations from ZCTAs, we may want to know which stations are within a ZCTA:

```
19 SELECT stop_name, trains  
FROM a_zctas, a_subway_stations  
WHERE zcta = "10010" AND ST_Within (a_subway_stations.geometry, a_zctas.geometry)
```

	stop_name	trains
1	23 St	6
2	23 St	FM
3	23 St	NR

Or which ZCTA every station is in:

```
20 SELECT zcta, stop_name, trains  
FROM a_zctas, a_subway_stations  
WHERE ST_Within (a_subway_stations.geometry, a_zctas.geometry)  
ORDER BY zcta, stop_name
```

	zcta	stop_name	trains
1	10001	28 St	1
2	10001	28 St	NR
3	10001	34 St-Herald Sq	B D F M
4	10001	34 St-Herald Sq	N Q R
5	10001	34 St-Penn Station	1 2 3
6	10001	34 St-Penn Station	A C E

Or how many stations are in each ZCTA (have to count the stations using their unique ID, as many stations have duplicate names):

```
21 SELECT zcta, COUNT(stop_id) AS stations
    FROM a_zctas, a_subway_stations
    WHERE ST_Within (a_subway_stations.geometry, a_zctas.geometry)
    GROUP BY zcta
    ORDER BY stations DESC, zcta
```

	zcta	stations
1	11207	13
2	11101	12
3	11201	12
4	11206	9

These examples evaluate point layers within other polygon layers, but you can relate any type of layer to any other, and for some operations you can evaluate features within a single layer. Let's say we want to list every ZCTA and its neighboring ZCTAs - since we're referencing features in the layer more than once we have to give each instance a distinct alias:

```
22 SELECT zcta1.zcta AS zcta, zcta2.zcta AS neighbor
    FROM a_zctas AS zcta1, a_zctas AS zcta2
    WHERE ST_Touches (zcta1.geometry, zcta2.geometry)
    ORDER BY zcta, neighbor
```

	zcta	neighbor
1	10001	10010
2	10001	10011
3	10001	10016
4	10001	10018
5	10001	10119
6	10001	10199

5.2.4 Spatial Index

When determining adjacency, overlap, or any other relationship the database evaluates each feature one at a time. Feature number 1 is checked against every single feature to determine their relationship, then feature number 2 is checked against every single feature, etc. The operation in our previous example didn't take too long as there are only about 200 ZCTAs, but it is inefficient. We would NOT want to run this process on our census tracts; there are over 2,000 of those and we would be waiting a few minutes.

Each of the three census geography "a" features (a_pumas2010, a_tracts, and a_zctas) have a spatial index that makes this process more efficient, but utilizing the index requires a different set of SQL statements. The spatial index

is a virtual table that acts as a bounding box index; rectangles are drawn around each feature to encompass it, and these rectangles become the search frame. Whenever a feature is evaluated it is compared to neighbors in the spatial index within its bounding box, rather than to every single feature in the layer.

So, if we want to generate a neighbor list for all the ZCTAs in NYC, we can use the spatial index to speed up the process:

```
23 SELECT zcta1.zcta AS zcta, zcta2.zcta AS neighbor
    FROM a_zctas AS zcta1, a_zctas AS zcta2
    WHERE ST_Touches(zcta1.geometry, zcta2.geometry)
    AND zcta1.rowid IN (
        SELECT rowid FROM SpatialIndex
        WHERE f_table_name='a_zctas' AND search_frame=zcta2.geometry)
    ORDER BY zcta, neighbor
```

NOTE: The rowid is a unique integer ID field that SQLite automatically assigns to every record in a table. rowid is "invisible" in the sense that you don't see it in the table and it's not listed in the metadata as a column, but it exists and you can call it in a SQL statement. The spatial index relies on the rowid, so we have to call it for this operation.

If you want to create spatial indexes for other layers, you can do so with this simple command:

```
24 SELECT CreateSpatialIndex("layer_name", "geometry")
```

5.2.5 Calculate area and density

We can calculate areas for all the ZCTAs; the default will be in square feet since our CRS is in feet, but we can do the extra work to get square miles:

```
25 SELECT zcta, ST_Area(geometry) AS sqft,
    (ST_Area(geometry))*0.0000000358700643 AS sqmi
    FROM a_zctas
    ORDER BY sqmi DESC
```

	zcta	sqft	sqmi
1	10314	386100312.702513	13.849443
2	10312	216029373.971924	7.748988
3	11234	210358174.577227	7.545561
4	10306	208988895.334777	7.496445

If we join the ZCTAs to our census table we can calculate population density; we'll modify our query to select just residential ZCTAs based on population:

```
26 SELECT zcta, HD01_S001 AS pop2010,
    HD01_S001 / ((ST_Area(geometry))*0.0000000358700643) AS density
    FROM a_zctas, b_zctas_2010census
    WHERE zcta=geoid2 AND pop2010 > 200
    ORDER BY density DESC
```

	zcta	bcode	pop2010	density
1	10162	36061	1685	148232.400385
2	10028	36061	45141	143685.635309
3	10075	36061	26121	141548.203054
4	10128	36061	60453	128442.218728

In addition to area you can also calculate perimeter (ST_Perimeter) for polygons and length (ST_Length) for linear features.

5.3 Extending the DB

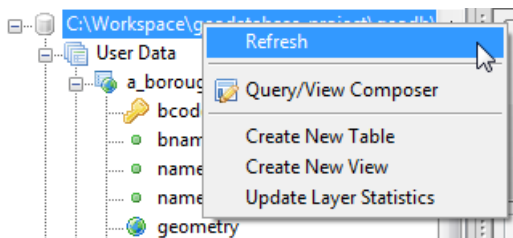
5.3.1 Create new view

As these queries can get quite involved you may want to save some of them to re-use in the future. When you're in Spatialite you can use the arrow keys to the left of the SQL statement box to cycle through queries you've created, but once you close the software your statements will not be saved. To save them, you have to create a view:

```
27 CREATE VIEW d_zctas_2010pop AS
SELECT zcta, bcode, HD01_S001 AS pop2010
FROM a_zctas, b_zctas_2010census
WHERE zcta=geoid2 AND pop2010 > 200
ORDER BY pop2010
```

	Message
Message	SQL query returned an empty ResultSet This is not an error

Whenever you run a CREATE, INSERT, or ALTER statement it's normal to get the Empty Results Set message (as you haven't done any selection). After you've run the statement, you have to refresh the database in order to see the view. Select the name of the database at the top of your tree in the menu on the left, right click, and choose refresh.



Scroll down and you can see your view listed in the menu; we've named it with the prefix "d" so that it doesn't get mixed up with the other objects in our database. A view saves your SQL statement - not the data that is shown as a result of executing it. So whenever you open a view you're executing a saved SQL statement.

5.3.2 Create spatial view

If you save a query as a spatial view you'll be able to view the results and map the data in QGIS. This requires two steps. First you write the CREATE VIEW statement. In this statement you must include the geometry column and the invisible rowid unique integer column (automatically created for every SQLite table; it's hidden but call-able) for the geographic feature.

```

28 CREATE VIEW d_zctas_spatial AS
    SELECT a_zctas.rowid, zcta, bcode, HD01_S001 AS pop2010, geometry
    FROM a_zctas, b_zctas_2010census
    WHERE zcta=geoid2 AND pop2010 > 200
    ORDER BY pop2010

```

Second, you need to register the view in a special table in the database that contains geometry for the views:

```

29 INSERT INTO views_geometry_columns
    (view_name, view_geometry, view_rowid, f_table_name, f_geometry_column,
    read_only)
    VALUES ("d_zctas_spatial", "geometry", "ROWID", "a_zctas", "geometry", 1)

```

Refresh the database and you should see the view in the tree. Now if you launch QGIS and connect to the database, `d_zctas_spatial` will be listed as a geographic feature. You can add it to a project and symbolize the data just like you would any other layer.

If you'd rather not go through all the steps of creating views, spatial or non-spatial, Spatialite has a menu driven view builder that you can use instead. If you select the database at the top of the tree, then right click, you can launch the Query/View composer.

LIMITATIONS WITH SPATIAL VIEWS: You cannot create spatial views that have calculated fields, like area or density from our earlier example. In the view, calculated fields do not have a data type associated with them, so QGIS is not able to read the columns and evaluate what they are. Also, you cannot create spatial views using aggregated queries that use the `GROUP BY` clause. If you create calculated fields or aggregates that you wish to map, you will have to create a brand new table with that data.

5.3.3 Create table from query

In many implementations of SQL it's possible to write a `CREATE TABLE AS` statement, where a `SELECT` statement with a query follows the `CREATE` statement and the result is written to a new table. While you can do this in SQLite it is NOT always recommended, because you will not be able to specify a primary key or data types for columns (thus, any calculated fields would not have a specified type). Generally, it's considered bad practice to have un-typed columns and tables without keys.

If you have a query or view that you would like to turn into a table, or you have a table where you want to drop or rename columns or change data types (as noted earlier, SQLite does not allow this as part of the `ALTER TABLE` statement), the following are the standard SQLite steps that you need to take:

1. Create a new, blank table where you specify your column names, their data types, a primary key, and any constraints that you wish to add.
2. Add a geometry column for spatial tables.
3. Insert the data from a query, table, or view into the new table.
4. Drop the old table or view (only if it's redundant or no longer needed).

Let's say that we want population density for ZCTAs. We need to turn the query into a permanent table so we can map the data in QGIS. First, we'll create the blank table that will hold our data; for data types the `zcta` is `TEXT` (since it's an identifier that may begin with a zero, and not a quantity), `pop2010` is an `INTEGER` (whole numbers) and `density` is a `REAL` (number with decimals):

```

30 CREATE TABLE d_zctas_density (
    zcta TEXT NOT NULL PRIMARY KEY,
    bcode TEXT,
    pop2010 INTEGER,
    density REAL)

```

Then we add a geometry column. We need to specify the type of geographic features as well as the coordinate system. POINT, MULTIPOINT, LINESTRING, MULTILINESTRING, POLYGON, and MULTIPOLYGON are the six primary options. Since one ZCTA feature may consist of several separate polygons, it's declared as a MULTIPOLYGON. The code for our CRS is 2263 (NAD 83 New York Long Island State Plane feet).

```

31 SELECT AddGeometryColumn ( "d_zctas_density", "geometry", 2263, "MULTIPOLYGON", "XY")

```

Refresh the database to see the table in the tree. Now we can insert the data into our table; we'll nest a subquery into our INSERT query to select the data we need. This makes our statement look a little busy; if we were inserting pre-existing data from a table or view it would look more straightforward. The order of the columns matters - data in the columns listed in the SELECT statement are going to be inserted into the columns in the INSERT statement in the specified order.

```

32 INSERT INTO d_zctas_density (zcta, bcode, pop2010, density, geometry)
    SELECT zcta, bcode, HD01_S001, HD01_S001/((ST_Area(geometry))*0.000000358700643),
    geometry
    FROM a_zctas, b_zctas_2010census
    WHERE zcta=geoid2 AND HD01_S001 > 200

```

Refresh the database to see the new spatially-enabled table, d_zctas_density in the list. View all the data:

```

33 SELECT *
    FROM d_zctas_density

```

	zcta	bcode	pop2010	density	geometry
1	10001	36061	21102	33959.490640	BLOB sz=1381 GEOMETRY
2	10002	36061	81410	92573.511829	BLOB sz=1741 GEOMETRY
3	10003	36061	56024	97188.806436	BLOB sz=1245 GEOMETRY
4	10004	36061	3089	5518.808120	BLOB sz=6307 GEOMETRY

And view the metadata:

```

34 PRAGMA table_info (d_zctas_density)

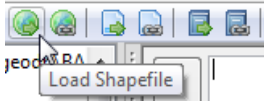
```

	cid	name	type	notnull	dflt_value	pk
1	0	zcta	TEXT	1	NULL	1
2	1	bcode	TEXT	0	NULL	0
3	2	pop2010	INTEGER	0	NULL	0
4	3	density	REAL	0	NULL	0
5	4	geometry	MULTIPOLYGON	0	NULL	0

This feature can be added to a QGIS project and symbolized like any other layer.

5.3.4 Import Data

You can load shapefiles, delimited text files, Excel spreadsheets (97-2003 files with the .xls extension), and dbf files into the database via the Spatialite GUI. For each format you have a choice of loading the file or creating a virtual table; the latter creates a link to the external file without actually loading it into the database.



When you load a shapefile you must specify what coordinate system it uses in the SRID input box; change the default from -1 to 2263, the EPSG code for NAD83 New York Long Island (ftUS). Before you load a shapefile you must prep it by transforming its CRS to that projection, so it will match the database layers (you can use the Transform function covered in the final section, or you can use any GIS software to reproject files). You should keep the default name of the geometry column as "geometry". You will have the option to specify a primary key and to create a spatial index.

When you load delimited text files you'll be asked to specify the delimiter (tab, comma, etc.) and whether or not text fields are surrounded by quotes (a common convention for preserving text values). If you have numeric values that need to be saved as text (like ZIP Codes or FIPS codes) you will need to prep your text data so that these columns are enclosed in quotes. Microsoft Excel is poor for working with text or csv files; it automatically saves anything that looks like a number as a number and doesn't allow you to quote text fields when saving a file as delimited text. Consider using either a text editor or Calc, the spreadsheet that's part of the LibreOffice suite <http://www.libreoffice.org/>. Calc does a better job with text files and also supports the dbf format (which Excel does not).

If you load an Excel file it must be in the older .xls format and not the .xlsx format. Current versions of MS Excel allow you to save in the older format by going to file, save as, and choosing the older 97-2003 xls file. LibreOffice Calc also supports the .xls format. Before import it's best to specify the data formats for each of the columns within the spreadsheet (including the number of decimal places for numbers), so Spatialite can interpret them properly on import. In Spatialite you'll be prompted to choose which worksheet you want to import from the Excel workbook.

When you import data Spatialite does not give you the option to include or exclude columns, rename them, specify data types, designate primary or foreign keys (except for shapefiles and dbf files), or enforce constraints. It imports all columns as is and makes assumptions about how to cast the data. If you want to insure that you have a well formed database you'll have to follow the steps in the previous section: import the data, create an empty table to hold the values you want, load the values into that table, and drop the imported data table. After you import your data, these are the generic steps:

```
CREATE TABLE newtable (  
newid TEXT NOT NULL PRIMARY KEY,  
otherid TEXT,  
value1 INTEGER,  
value2 REAL)  
  
35 SELECT AddGeometryColumn ( "newtable", "geometry", 2263, "TYPE OF GEOMETRY", "XY")  
  
INSERT INTO newtable (newid, otherid, value1, value2, geometry)  
SELECT shapeid, label, popvar, housevar, geometry  
FROM imported_file  
  
DROP imported_file
```


5-3-5 Importing Coordinate Data and Transforming Projections

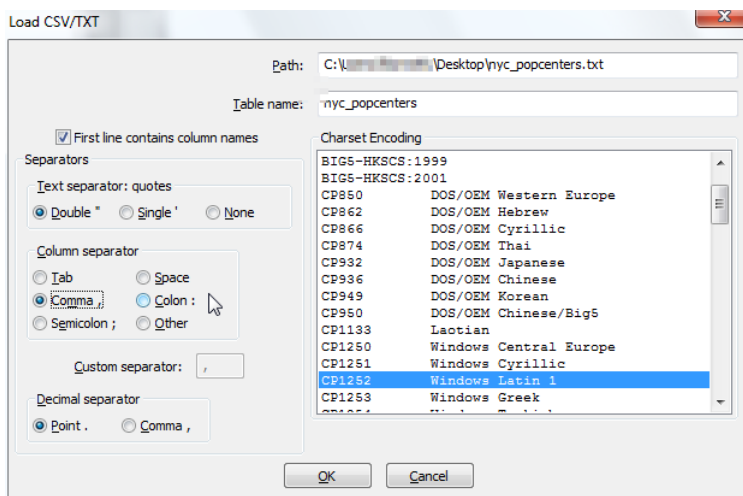
In this section we'll cover several operations: importing a data table, converting data stored as XY coordinates into geometry, transforming the coordinate system of a layer, adding columns, and updating data. We'll import the population centers of New York's counties into the database. Open your computer's text editor (i.e. Notepad on MS Windows), copy the data below and paste it into the editor. **Make sure to hit the enter key at the end of the last record (Richmond), so it will be recognized as a record. Save the file as nyc_popcenters.txt.

```
STATEFP,COUNTYFP,COUNAME,STNAME,POPULATION,LATITUDE,LONGITUDE
```

```
"36","005",Bronx,New York,1385108,+40.849354,-073.882363  
"36","047",Kings,New York,2504700,+40.650523,-073.954811  
"36","061",New York,New York,1585873,+40.777958,-073.966303  
"36","081",Queens,New York,2230722,+40.721294,-073.833451  
"36","085",Richmond,New York,468730,+40.588523,-074.137304
```

(Given the small amount of data, we could have created a blank table and used a SQL INSERT statement to populate it. But since it's more likely that you'll be importing data rather than creating it manually, we'll use this example).

First we'll load the text file into the database - hit the load CSV / TXT file button on the toolbar. Browse to the location where you saved the file and select it. In the menu check the box that says First line contains column names, select the Comma option under Column separator, and verify that the Text separator: quotes option is set to Double. Hit OK.



Query the data to make sure it loaded properly.

```
36 SELECT * FROM nyc_popcenters
```

	PK_UID	STATEFP	COUNTYFP	COUNAME	STNAME	POPULATION	LATITUDE	LONGITUDE
1	0	36	005	Bronx	New York	1385108	40.849354	-73.882363
2	1	36	047	Kings	New York	2504700	40.650523	-73.954811
3	2	36	061	New York	New York	1585873	40.777958	-73.966303
4	3	36	081	Queens	New York	2230722	40.721294	-73.833451
5	4	36	085	Richmond	New York	468730	40.588523	-74.137304

Next we'll create the table that will hold our data. We'll have just one column to hold our id code and we'll drop the state name as it's unnecessary.

```
37 CREATE TABLE a_borough_popcenters (
    bcode TEXT NOT NULL PRIMARY KEY,
    cname TEXT,
    pop2010 INTEGER,
    latitude REAL,
    longitude REAL)
```

Next we create a geometry column for the table.

```
38 SELECT AddGeometryColumn ("a_borough_popcenters", "geometry", 2263, "POINT", "XY")
```

Then we'll insert the data from the original table into our new one.

```
39 INSERT INTO a_borough_popcenters (bcode, cname, pop2010, latitude, longitude,
    geometry)
    SELECT (statefp || countyfp), couname, population, latitude, longitude,
    Transform(MakePoint(longitude, latitude, 4269),2263)
    FROM nyc_popcenters
```

There are a few things going on here. First, we're using || to concatenate the state and county codes into a single code. Second, we're taking the longitude and latitude coordinates (longitude is always X, latitude is Y) and using MakePoint to create geometry out of them; we have to define the geometry as NAD 83 (EPSG code 4269) because that's the format the coordinates are in (most lat / long data from the US government is in NAD 83). Lastly, we're taking that new geometry and transforming it to NY Long Island feet (EPSG 2263) because that's the coordinate system that the rest of our data is in, and we want to be able to layer this data over our other layers.

In this case we nested the MakePoint command inside the Transform command. Ordinarily, if we were reprojecting layers that have geometry, we'd create a new table, add a geometry column with the system we want, then copy the geometry from the existing table into the new one while transforming it: Transform (geometry, EPSG code).

Refresh the database, then query the data to make sure it looks OK.

```
40 SELECT * FROM a_borough_popcenters
```

	bcode	cname	pop2010	latitude	longitude	geometry
1	36005	Bronx	1385108	40.849354	-73.882363	BLOB sz=60 GEOMETRY
2	36047	Kings	2504700	40.650523	-73.954811	BLOB sz=60 GEOMETRY
3	36061	New York	1585873	40.777958	-73.966303	BLOB sz=60 GEOMETRY
4	36081	Queens	2230722	40.721294	-73.833451	BLOB sz=60 GEOMETRY
5	36085	Richmond	468730	40.588523	-74.137304	BLOB sz=60 GEOMETRY

Delete the original table since we don't need it any longer, then refresh the database.

```
41 DROP TABLE nyc_popcenters
```

Our new layer has county names but no borough names. We could have created a new column in the MAKE TABLE statement to hold the names, but we can add additional columns after the fact:

```
42 ALTER TABLE a_borough_popcenters ADD COLUMN bname TEXT
```

Now we can add the borough names. We can simply copy the names for the Bronx and Queens from the county column, since the names are identical; for the other three boroughs we have to change them manually. You could use the GUI interface to modify the tables, or use SQL:

```
43 UPDATE a_borough_popcenters  
SET bname=cname WHERE bcode IN ("36005","36081")
```

```
44 UPDATE a_borough_popcenters  
SET bname="Brooklyn" WHERE bcode="36047"
```

```
45 UPDATE a_borough_popcenters  
SET bname="Manhattan" WHERE bcode="36061"
```

```
46 UPDATE a_borough_popcenters  
SET bname="Staten Island" WHERE bcode="36085"
```

```
47 SELECT * FROM a_borough_popcenters
```

	bcode	cname	pop2010	latitude	longitude	geometry	bname
1	36005	Bronx	1385108	40.849354	-73.882363	BLOB sz=60 GEOMETRY	Bronx
2	36047	Kings	2504700	40.650523	-73.954811	BLOB sz=60 GEOMETRY	Brooklyn
3	36061	New York	1585873	40.777958	-73.966303	BLOB sz=60 GEOMETRY	Manhattan
4	36081	Queens	2230722	40.721294	-73.833451	BLOB sz=60 GEOMETRY	Queens
5	36085	Richmond	468730	40.588523	-74.137304	BLOB sz=60 GEOMETRY	Staten Island

You can add this layer to QGIS and overlay it with the boroughs to see where the population centers are; they happen to be close to where the geographic centers are, indicating that population is rather evenly distributed in each borough. Staten Island is an exception; the population is clustered a little more to the north of the island.

5.4 Reference & Links

5.4.1 SQLite

- Official reference for all SQLite functions: <http://www.sqlite.org/lang.html>
- Good SQLite tutorial: <http://zetcode.com/db/sqlite/>
- Good SQLite & Python tutorial: <http://zetcode.com/db/sqlitepythontutorial/>
- Good book: *Using SQLite* by Jay Kreibich, O'Reilly Media 2010.

5.4.2 Spatialite Concepts and Reference

- Spatialite 4.1 SQL functions reference: <http://www.gaia-gis.it/gaia-sins/spatialite-sql-4.1.0.html>

- Spatialite Cookbook: <http://www.gaia-gis.it/gaia-sins/spatialite-cookbook/index.html>
- Other Spatialite tutorials and docs: <https://www.gaia-gis.it/fossil/libspatialite/wiki?name=misc-docs>
- Scratching Surfaces (blog with good examples): <http://www.surfaces.co.il/?cat=12>
- Spatial Relationships (for PostGIS but largely applies): http://workshops.opengeo.org/postgis-intro/spatial_relationships.html
- Spatial Geometries (for PostGIS but largely applies): <http://workshops.opengeo.org/postgis-intro/geometries.html>

5.4.3 Forums for Help

- GIS Stack Exchange: <http://gis.stackexchange.com/>
- stackoverflow (for SQLite, SQL, or general database questions): <http://stackoverflow.com/>

5.4.4 Common CRS for NYC Area

- EPSG 2263 (NAD83 / New York Long Island (ft US)): used in the nyc_gdb and by most city agencies
- EPSG 26918 (NAD83 / UTM Zone 18 N): another projected coordinate system that's suitable for the region (units are in meters)
- EPSG 4269 (NAD83): basic longitude/ latitude system used by US federal government agencies like the Census Bureau
- EPSG 4267 (NAD27): older longitude/ latitude system commonly used by US federal government agencies up to the mid 1990s
- EPSG 4326 (WGS84): basic longitude/ latitude system common throughout the world; almost identical to NAD83