Spatial SQL

Michele Tobias Naomi Kalman

2024-03-20

Overview

Workshop Description

This workshop is intended to give participants an introduction to working with **spatial data** using SQL. We will work with a graphical user interface (GUI) and explore some examples of common analysis processes as well as present participants with resources for continued learning. This workshop will give participants a solid foundation on which to build further learning.

Learning Objectives

By the end of this workshop, participants will be able to

- Import data into an spatialite database
- Write queries to answer questions about spatial data
- Understand the difference between attribute queries and geometry queries
- View spatial tables and views in QGIS
- Use terminology related to spatial databases to facilitate future learning

Prerequisites

An introductory understanding of SQL is recommended, but not mandatory. For example, knowing how to compose a SELECT statement using SQL and the general concept of joining tables will serve learners well. For learners without a foundation in SQL, we recommend attending or reviewing the materials for DataLab's Introduction to Databases and Data Storage Technologies, which introduces the concept of databases, and Intro to SQL for Querying Databases, which teaches the basics of querying data using SQL.

Concepts

What is a relational database?

A relational database is a set of data in tables that are related to each other in some way. That's it. **A database is just a collection of related tables.**

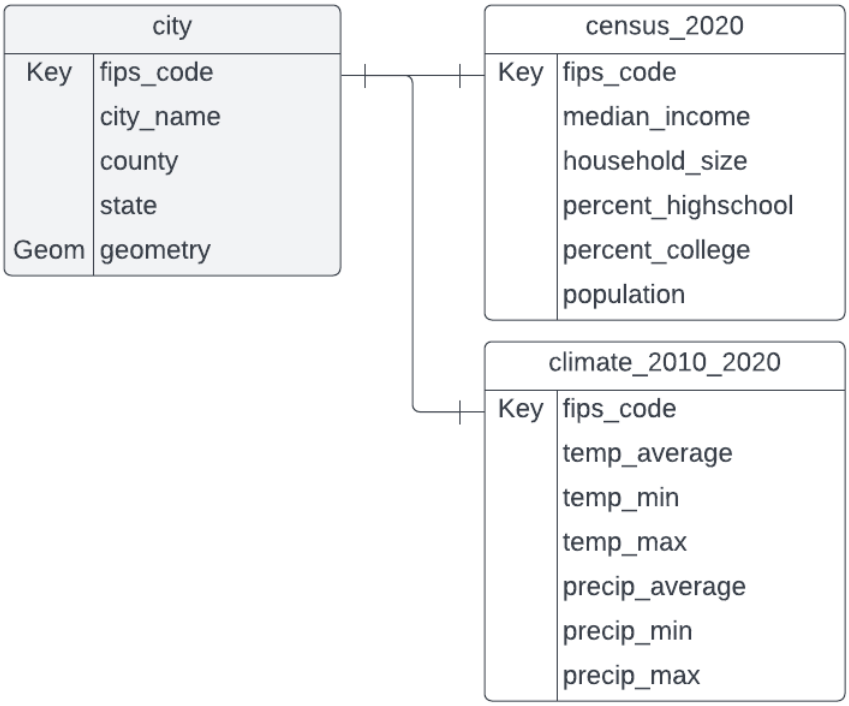
Generally each table can be connected to another table by a column that both tables have that stores the information to match up the rows. This column is called a **key**. For example, your student or employee ID number is a key commonly used on campus.

If you've done GIS, you already use a database. A shapefile is essentially just a fancy a table. And if you're ever joined a .csv table to a shapefile, you've performed a database process called a join. Using multiple spatial datasets in a GIS relates the data by location, if not by tabular data.

What is a Spatial Database?

A spatial database is a normal database (i.e. a set of related tables) but at least one of the tables has a column that holds spatial information commonly called the "geometry". The geometry information is stored as a Binary Large Object (BLOB). The geometry information allows us to relate the tables to each other based on their location and also to perform spatial analysis on our data.

Below is an example of a database entity relationship diagram (ERD) for tables in an example (imaginary, yet plausible) spatial database. At least one column in each table relates to a column in another table (indicated in this diagram by a line drawn between the two columns). Documenting a database with a diagram like this is common practice. It provides a quick visual reference to the data contained in each table and how the tables relate to each other. The `city` table is a spatial table. The `geometry` column provides the spatial information that allows the city polygons to be mapped. The other tables are tabular data that can be joined to the spatial table using the key, in this case, the Federal Information Processing System (FIPS) Code, a system that assigns unique numeric codes to geographic entities such as cities, counties, and states. FIPS codes help us have a standardized key that prevents mismatches from variations in names (such as "Davis" vs. "City of Davis").



What is Spatial SQL?

SQL stands for "structured query language" and it's a language that allows you to ask questions of a database. Spatial SQL is regular SQL but with some additional functions that perform spatial analysis. Spatial SQL functions typically work on the geometry column.

If you've ever written an attribute query in ArcGIS or QGIS, you've worked with SQL. Example: Hey GIS program, please highlight all the records in my attribute table that have "Yolo" in the "county" column! In SQL, we would write `SELECT * FROM city WHERE county = 'Yolo';` It's actually quicker to write that query than to fill out the interface in the GIS.

A spatial join is an example of a spatial operation that you may have performed in a GIS that

can also be performed using SQL. For example, if we had a dataset of business locations, we might query which businesses are inside of city boundaries. `SELECT * from businesses, cities WHERE ST_Intersects(businesses.geometry, city.geometry);` It's much quicker to write the query than fill out the spatial join interface.

Why should you learn to work with spatial databases and spatial SQL?

- It's a good way to work with large amounts of data
- Typically faster to run a process in a spatial database than in a desktop GIS program
- Store lots of data (compare with shapefile's 70m row limit)
- One database file stores many, many tables –> easier data management
- Write a query instead of making a new file (no exporting of intermediate results to shapefile necessary!)

What makes this challenging?

If you're a GIS user, you're probably used to a graphical user interface (GUI) where you can see your data, have tools with guided interfaces, and can see the results of your processing immediately. These aren't things you get with a typical database manager tool, however, we can connect our database to QGIS so we can see our results and, with practice, you will get used to the typical workflow and seeing everything won't be so necessary.

Frequently Asked Question: Can I use spatial databases with R or Python? Yes! Look for libraries that can read data from your database file or can query your database to create data tables to use in your code.

Databases that support Spatial SQL:

- DB Manager in QGIS
- Oracle
- MySQL
- SQL Server
- SpatiaLite
- PostGRES with the PostGIS extension
- Others too!

We'll be working with the DB (database) Manager in QGIS, because it works on all the common computer operating systems and is fairly easy to install. QGIS also allows us to seamlessly view our results on a map. Once you learn the basics, you can choose the database program that best fits your needs.

Data

The data we'll be working with is a set of hydrology-related data for the San Francisco Bay Watershed.

You'll need to download the following data from the data folder in this repository or from this Box folder (click the download button in the upper right corner to download all the data in one .zip file):

- Watershed Boundaries (Polygons)
- Watershed Centroids (Points)
- Rivers (Lines)

If you downloaded a .zip file, be sure to unzip it.

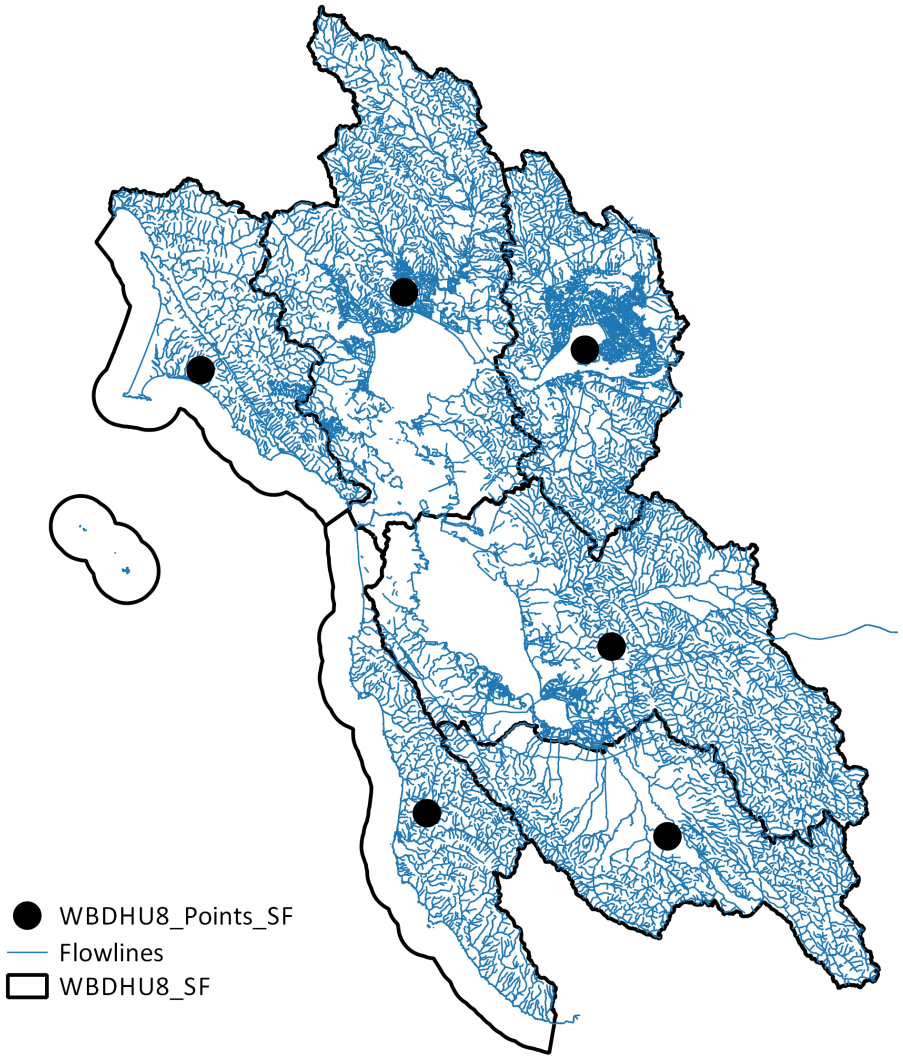


Figure 1: alt text

Creating a Database

Start QGIS

Open QGIS. Start a new project by either clicking on the white rectangle icon (with the corner turned down) or selecting "new" under the Project menu. You now should have a blank map canvas ready to go.

Starting the DB Manager Plugin

We'll be using the DB Manager plugin, so let's make sure that is installed:

1. Click the Database menu at the top of the QGIS window. Do you have the DB Manager listed as an option in this menu? If so, click the icon to open the tool and skip Step 2 below. If not, move on to Step 2 below.
2. If the DB Manager is not in the Database menu, we need to enable the plug in:
    i. Click the Plugins menu at the top of the window and select "Manage and install plugins". This will open the Plugins Manager.
    ii. Make sure you're in the "All" tab on the left side of the tool, and then in the search box, start typing "DB Manager" to narrow down the options.
    iii. Select "DB Manager" from the list to view the details about this plugin.
    iv. Click the *Install Plugin* button in the lower right corner to install it. If this option isn't available, the tool is probably already installed and you just need to make sure the box next to the tool in the list is checked so that is available. Now go back to Step 1 above to start the tool.

Make a Database

You can think of a database as a folder in which you keep tables that are related to each other. You don't want to put data in this database that isn't related to the other data (you could, but that's not the point of a database).

We'll need to make a new database to keep our spatial tables in:

1. Locate your Browser Panel. By default, it usually sits above or below the Layers Panel on the left side of the Map Canvas. If you've closed it (like your instructor tends to do), you can get it back with *View* Menu –> *Panels* –> *Browser Panel*
2. Right click on the SpatiaLite item (it has the feather icon) [alt text](./images/QGIS_Spatialite_Icon.png "the spatial lite icon looks like a feather) and select *Create Database*
3. Navigate to where you would like to keep your database, perhaps in the folder where you downloaded the data for this workshop. Name your database *sfbay.sqlite* because we'll be working with San Francisco Bay data. Yes, it should be all lowercase.

You may have noticed that there are many database format options available to you besides SpatiaLite. We're working with a SpatiaLite file today, but you can match your future databases to the needs of your project. Different database formats have different functions available and can be better for certain types of analysis. For example, PostGIS is great for geocoding and also for working with very large datasets, among other things.

Let's connect to our database:

1. Open or locate the DB Manager that we found earier.
2. Right click on the *SpatiaLite* option and select *New Connection*.
3. Navigate to where you saved your *sfbay.sqlite* file, slect the file, and click *Open*.
4. Expand the list (click on the > symbol) in the *Providers Panel* on the left to see that your database is now available.

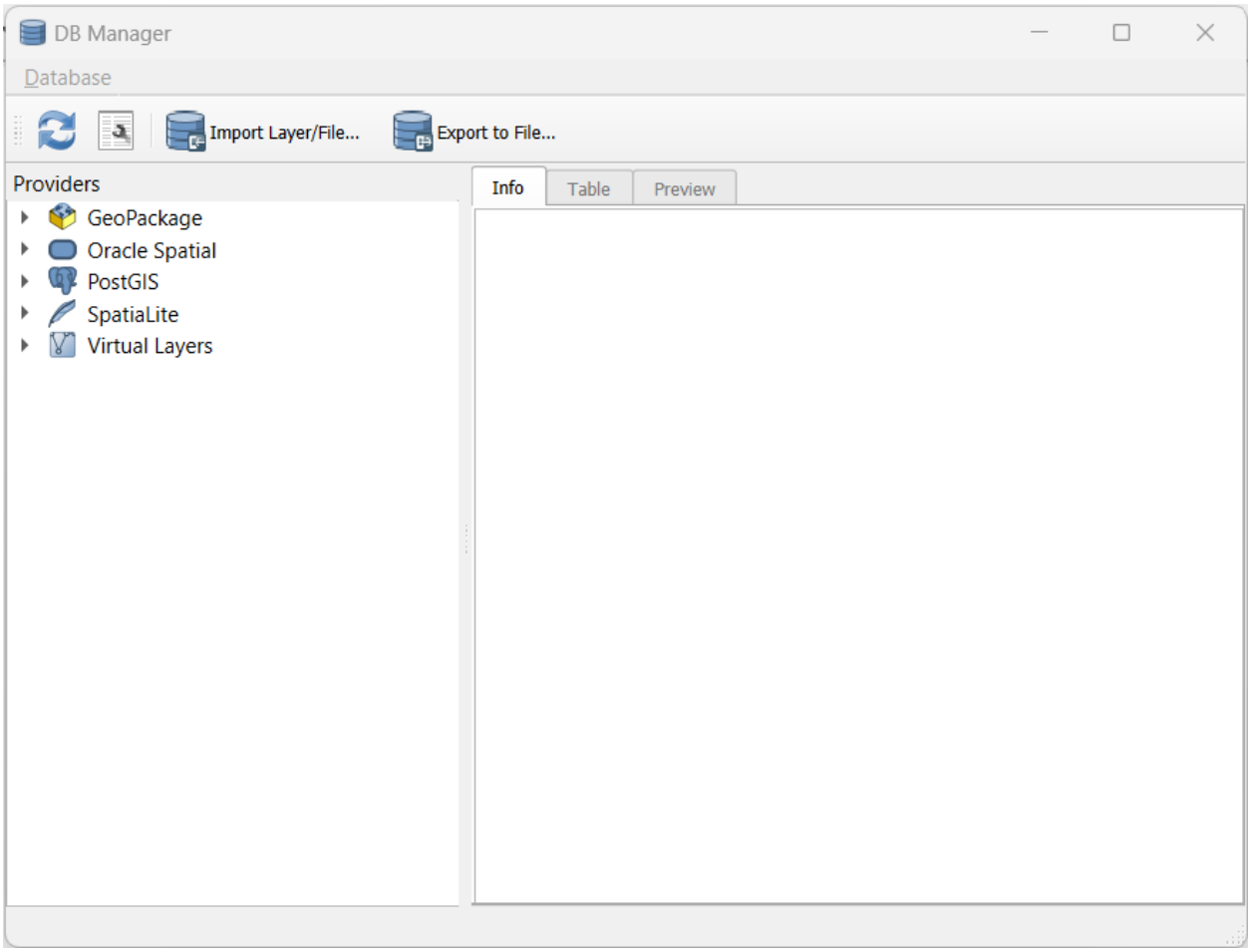


Figure 2: alt text

5. If you expand the list next to the *sfbay.sqlite* database, you'll see if has some tables, but none of these contain data yet. We'll add some data next.

Load Data

Let's add some data to our database:

1. Click on *sfbay.sqlite* in the *Providers Panel* to select it in the list.
2. Click on the *Table* menu, and select *Import Layer/File* to open the Import Vector Layer dialog.
3. For the *Input*, click the button with the three dots (ellipsis mark) to browse your files. Select the *Flowlines.shp* file and click *Open*. If you're familiar with shapefiles, you might wonder, "Do I need to select all of the files that comprise a shapefile?" No, you just need to select the .shp and QGIS will know to look for the rest.
4. In the *Table* box, type *flowlines* for the name of our new table - all lowercase
5. In the *Options* section, check the *Source SRID* box. Use the "Select SRID" button to open the projection dialog. In the filter box at the top, type 3310 to narrow down the options. Select "NAD83 / California Albers EPSG:3310" for the Coordinate Reference System because the projection for this data is California Albers (NAD38) which has the SRID of 3310.
6. NOTE: There are many other options here that you may find useful later. In particular, the Encoding option allows you to tell QGIS which character encoding you are using. Did your characters show up as rectangles or other unreadable characters? Reimport your data and specify your character encoding.

You should see a pop-up that says "Import was successful." Click OK.

You can repeat this process to add the other two datasets. Call the *WBDHU8_Points_SF* table *centroids* and *WBDHU8_SF* table *watersheds*.

Congratulations! You now have a database with files related to the San Francisco Bay's watersheds!

SQL Window

Now we're just about ready to do some analysis with our database. Click on the *SQL Window* Icon.  You may need to expand the window by dragging the lower right corner of the window out so you can see everything.

A query is a request for information from the database. You will type your queries into the big blank box at the top of the window (next to the 1 line number).

You'll run the query by clicking the *Execute* button (or hover over the button and it will pop up a tool tip telling you what the keyboard keystrokes are on your system).

The results of the query will appear in the box below. Sometimes the results will be a table; sometimes it will be a message.

A query has a structure. The most common one you'll see today is a "select statement". These start with the SELECT command, followed by the information you want to know, then the name of the table you want the information from, and finally (and optionally) other parameters that limit the results or provide some important caveats. Queries end with a semicolon.

Review Non-spatial Queries

Non-spatial queries are queries that don't involve the geometry column (the spatial information) of our table.

We'll start by investigating our *flowlines* data. The *flowlines* are linear features that carry water from one place to another. Some are nature features like rivers or streams, others are man-made like canals.

Let's look at the whole table:

```
SELECT * FROM flowlines;
```

The asterisk (*) means "everything" or "give me all the columns". You could read the query as "Select everything from the *flowlines* table."

The result should look very much like the attribute table you explored earlier, but with a couple of additional columns. The import process added an *id* field and a *geom* field. The *geom* field contains information that allows the database tool to know where that particular object should be located in space, but unfortunately, it doesn't look like anything we understand as humans. We'll learn to deal with this column more in a little while.

Add a WHERE clause:

```
SELECT * FROM flowlines WHERE ftype = 460;
```

This query limits our results to just the rows where the number in the *ftype* column is 460, which corresponds to the natural rivers and streams (not canals). "Where" in this case does **NOT** indicate location, but rather a condition of the data.

Add a function:

```
SELECT COUNT(id) FROM flowlines WHERE ftype = 460;
```

Here we've added the function COUNT(). So we've asked the database tool to count all of the IDs but only if they have an FYTYPE of 460.

Summarize Data

What if we wanted to know how many lines there were of each *ftype*?

```
SELECT ftype, COUNT(id) FROM flowlines GROUP BY ftype;
```

Here, I've asked for a table with the *FTYPE* and the count of each *id*, and finally that it should summarize (group by) the *FTYPE*.

If I don't like the column name that it automatically generates - `COUNT(id)` - I can give it an alias with the AS command:

```
SELECT ftype, COUNT(id) AS NumberOfLines FROM flowlines GROUP BY ftype;
```

This is especially handy if you're making a table for people unfamiliar with your data or SQL or if you need the column name to be something specific.

Spatial Queries

Basic Spatial Query Examples:

View Geometry

Let's start understanding spatial queries by looking at the geometries column:

```
SELECT ST_AsText(geom) FROM flowlines;
```

ST_AsText() lets us see the geometry string in human-readable form. This isn't very useful most of the time, but perhaps it's comforting to know it's there. You can make columns in the results tables larger by placing your mouse cursor over the edge of the column and dragging it out once the expander handle appears (it looks like two arrows pointing different directions).

Length

Let's do an analysis that you might come across. Let's get the lengths of each of the *flowlines*: `SELECT id, ST_Length(geom) FROM flowlines;`

What are the units of the length query? The units are meters because the units for the projection (California Albers; SRID 3310) are meters.

We just found the length of the individual *flowlines*. That was not a very informative query. It would be more useful to know what the total length of the lines are summed by their fcode.

```
SELECT fcode, SUM(ST_Length(geom)) FROM flowlines GROUP BY fcode;
```

Area

Let's look at an example to get the area of the *watershed* polygons:

```
SELECT NAME, ST_AREA(geom) FROM watersheds;
```

Remember that if you don't like the column headings, you can alias them with *as*.

Projections:

Because we are working with spatial data, we need to know how to handle projections.

Set the Projection

When you import your data into the DB Manager, it asked you what the SRID (EPSG Code) was for your data. It's easy to forget to do this or to put in the wrong one if you're in a hurry. If you discover that you've made a mistake, you don't need to re-import the table; you can set the SRID to the correct projection with an update command. For our data it would look like this:

```
UPDATE flowlines SET geom = SetSRID(geom, 3310);
```

This query replaces the contents of the *geom* column with the results of the SetSRID command. In our case, it doesn't really do anything new since we had our projection set correctly, but you should know how to do this, so we did.

Reproject

To change the projection of a dataset, you need to use the *Transform* or *ST_Transform* command.

Let's transform our watershed data into UTM Zone 10 North, the zone that San Francisco falls into.

First, we'll start with a query that results in a returning information (but doesn't make a new table):

```
SELECT id, HUC8, NAME, geom FROM watersheds;
```

We have a table with a subset of the columns from the original watersheds table. Now let's work on transforming the *geom* column. We'll add a function around the *geom* column to reproject the data. 26910 is the SRID for UTM Zone 10 N.

```
SELECT id, HUC8, NAME, ST_Transform(geom, 26910) FROM watersheds;
```

It may look like nothing happened, but the column heading on the *geom* column should have changed. Finally, we'll want do something to keep this data. Remember that a SELECT statement just returns information, it doesn't save it. We have two options. (1) We could overwrite the *geom* column of the *watersheds* table, but that will mean the projection won't match the other data we have. (2) The other option is to make a new table with a different projection. We can do this by adding a CREATE TABLE statement to the query we already have:

```
CREATE TABLE watershedsUTM AS SELECT id, HUC8, NAME, ST_Transform(geom, 26910) as geom
FROM watersheds;
```

To see the new table in the list, we'll need to refresh our database. From the Database menu at the top of the DB Manager window, select *Refresh*. Now we can see the table, but it's just a table. The database doesn't seem to know the table is actually polygons.

`SELECT * FROM watershedsUTM;` Shows that all the columns we asked for, including the *geom* column are there. What's going on? We need to recover the geometry column so the database will recognize the table as a spatial table.

```
SELECT RecoverGeometryColumn('watershedsUTM', 'geom', 26910, 'MULTIPOLYGON', 'XY');
```

This query will return a single column and row. Now we need to vacuum the database (yes, that sounds a little odd). From the *Database* menu, select *Run Vacuum*. Now the icon next to the *watershedsUTM* table should look like a set of polygons.

Now we could add this to our map canvas to see the polygons. Right click on the *watershedsUTM* table in the tree, and select *add to map canvas*. Note that we just reprojected the data, so it won't look too much different.

Spatial Join

Let's go back to the DB Browswer window and look at some more complex queries.

Spatial joins allow us to combine information from one table with another based on the location associated with each record. Let's see if we can figure out which watershed each of our flowlines is in:

```
SELECT flowlines.*, watersheds.NAME as Watershed_Name
FROM flowlines, watersheds
WHERE ST_Contains(watersheds.geom, flowlines.geom);
```

Your table should look just like your *flowlines* table, but we've added the *NAME* column from our *watersheds* table (but called it "Watershed_Name" because this will make more sense if we needed to use the this data later and didn't remember where this information came from).

ST_Contains tells us if a line is completely within a particular watersheds polygon. How would you change this query to identify which watershed each line *intersects* rather than is *contained by*? Hint: SpatiaLite Function Reference List

Viewing a Query to QGIS

You don't have to save a query as a table to view it in QGIS. Sometimes you just want to see the results, but don't need to keep a table that you might not need later. Let's see the spatial join we just did in the QGIS map canvas:

1. After you run the code to do the spatial join above, you will see a check box below the table that says *Load as new layer*. Click in the box to check it off. New options will appear.
2. Check the box next to *Column with unique values* and select *id* from the list.
3. Check the box next to *Geometry column* and select *geom* from the list.
4. Fill in the *Layer name (prefix)* with "Flowlines in Watersheds".
5. Click the *Load* button. It may take it a few minutes to load because the *flowlines* layer is large.

Now you have access to the *Layer Properties* and all the other tools you might use with any other vector layer in QGIS. For fun, let's style this layer:

1. Open up the *Layer Properties* (right click on the layer in the *Layers Panel* and select *Properties*).
2. Click on the *Symbology* tab.
3. Instead of *Single Symbol*, let's pick *Categorized* from the top drop-down menu.
4. For the *Column*, pick *Watershed_Name*
5. Click the *Classify* button near the bottom.
6. Finally, click *OK* to close the dialog and apply our changes.

Spatial Analysis

Not surprisingly, you can use a spatial database to do more than just get lengths and areas of existing geometries, or change projections. Let's go back to the DB Manager.

Distance

Let's find out which watershed is closest to the city of San Francisco. We could go about this a number of ways, but let's find the distance from the city's center point to the centroid of each watershed:

```
SELECT ST_Distance(MakePoint(37.7749, -122.4194), centroids.geom) FROM centroids;
```

Here we used *MakePoint()* to turn a set of latitude/longitude coordinates into a format that the database tool understands, then put the results into the *Distance()* function.

How could you make this table more informative? Could you add or rename some columns?

Buffer & Nesting Functions

One interesting thing about SQL is that you can nest functions to do a series of functions in one query like you just saw above, but it can get more complex. For example, maybe I want to find out the area (in square kilometers) within 1 kilometer of all the *flowlines*.

```
SELECT sum(ST_Area(ST_Buffer(geom, 1000)))/1000000 FROM flowlines;
```

Here, we take the sum of the area of the buffer of 1000 meters, then divide the whole thing by 1,000,000 to convert square meters to square kilometers. Wow. That's pretty complicated. But I didn't have to make a bunch of intermediate files and add columns to an attribute table, then save a CSV, then sum it all up in Excel. Now which option sounds more complicated? Perhaps you're starting to see some of the power of spatial SQL.

Conclusion

Today we've gotten an introduction to Spatial Databases and Spatial SQL. This is certainly not all you can do with these tools; we've barely scratched the surface! To encourage you to keep learning, I've provided more resources below. Additionally, a well-crafted Google search can return helpful posts and tutorials for learning new skills.

Additional Resources:

More about Databases with DataLab

Introduction to Databases and Data Storage Technologies

Intro to SQL for Querying Databases

General SQL Resources:

W3 Schools' SQL Reference

Spatial SQL Resources:

QGIS Training Manual on DB Manager

SpatiaLite Function Reference List

SpatiaLite Cookbook

General Slides & Tutorials:

Todd Barr's Slides

Mike Miller's Spatialite Tutorial

Assessment