How do I do that in SpatiaLite/SQLite: Illustrating Classic GIS Tasks

Technical Report · February 2017

DOI: 10.13140/RG.2.2.34127.28324

Arthur J. Lembo
Salisbury University
46 PUBLICATIONS **838** CITATIONS
SEE PROFILE

How do I do that in SQLite and SpatiaLite:

Illustrating classic GIS tasks

Arthur J. Lembo, Jr.

Arthur J. Lembo, Jr.
2015

First Printing: February, 2017

Arthur J. Lembo, Jr.
1440 East Sandy Acres Drive
Salisbury, MD 21804

www.artlembo.wordpress.com

Preface

In 2004, I created a small publication with my students titled *How Do I Do that In ArcGIS/Manifold*. It was an enormously fun endeavor, which to my surprise really took off in the GIS community. There were tens of thousands of downloads, some lengthy debate, and finally some attempts to replicate the document for other GIS software products. Seeing the document take on a life of its own was really gratifying. This current document is a continuation of that theme, but with a focus on the use of spatial SQL in PostGIS to accomplish the tasks.

Once again, we will revisit the 1988 United States Geological Survey (USGS) classic document titled *The Process for Selecting Geographic Information Systems*[1] (Guptil, et. al., 1988). As you might recall from the previous *How Do I Do That* document, the USGS report provided an overview of the process for selecting geographic information systems, in addition to a checklist of functions that a GIS should include. The functions were broken into five separate categories: user interface, database management, database creation, data manipulation and analysis, and data display and presentation.

As the title indicates, I envision this book to act as a sort of reference to the question of *how do I do that..*, residing on the user's lap while attempting to implement GIS functionality with SQLite SQL. Also, I believe simply perusing through the pages will be convincing enough to cause users to consider the use of SQL as part of their daily GIS activities. As a teaching tool, one can see how many of the same SQL functions are just reused in a different fashion to complete a task, without the need to run some kind of special wizard or new tool. So, if you can write SQL, you can build all kinds of functionality in GIS.

You will also notice that this book is much shorter than the original *How do I do that in ArcGIS/Manifold*, and without illustrations. This was done on purpose to keep the cost of production low and to allow users to quickly get an answer to their spatial SQL questions. All of the examples were tested using Spatialite-GUI with an associated .sqlite database. A dump of that database is downloadable from my blog: artlembo.wordpress.com. Therefore, users can recreate the SQL in the book on the actual data. I encourage you to retype the queries. Simply copying and pasting the queries is not going to help you learn what is actually happening. So, while there is a little more effort on the part of the reader, I believe it will be the most effective way to learn how to write SQL. As you become more accustomed to writing SQL, you will find that you begin to *think in SQL*. For me, when presented with a GIS conundrum, I constantly find myself thinking about the SQL solution, rather than the classical GIS commands.

[1] Guptill, S., D. Cotter, R. Gibson, R. Liston, H. Tom, T. Trainer, H. VanWyhe. 1988. "A Process for Selecting Geographic Information Systems". Technology Working Group – Technical Report 1. USGS Open File Report 88-105.

SQL is very easy to understand query language, even for those who are unfamiliar with programming. When I began this book, I had no idea whether SQLite with the Spatialite extension could complete a majority of the tasks using only the SQL engine. I was quite pleased to see that many of the tasks listed in *How do I do that in ArcGIS/Manifold* could actually be accomplished with the SQL engine in SQLite. However, the reader should note that raster analysis is not supported in Spatialite – for that, you will need another product named Rasterlite. Therefore, the raster tasks are not included in this document.

I welcome you to participate in the discussion of this book on my blog: **artlembo.wordpress.com**.

Arthur J. Lembo, Jr.
February, 2017

Table of Contents

How to understand this guide

This guide follows the topic headings from the book *How do I do that in ArcGIS/Manifold*, as a way to illustrate the capabilities of Spatialite for accomplishing classic GIS tasks. You will notice that several gaps exist where the tasks cannot be completed using SQL. In some instances the gaps are logical as the specified task requires user interaction. In other cases, however, the gap exists because a function that one might assume would be a logical addition to the software simply was not built into the Spatialite SQL engine. A good example of this is the creation of area features from connected (spaghetti) lines. Another example are all of the raster functions. Spatialite does not currently support raster analysis.

All of the SQL queries were tested on a SQLite dump file you can download from artlembo.wordpress.com. Much of the SQL in this book can likely be used in other ANSI compliant databases that make use of the OGC specifications. All of the queries were written in the most straightforward way possible – however, as you learn SQL, you will discover other ways to accomplishing the tasks. In many cases, there may be more efficient ways to accomplish the task.

I have attempted to indent the queries as best as possible so that it is easier to read. However, there are some examples that have too long of names, or too much code to fit on a single line on the page. In these cases, there may be some minor *wrapping* of the text.

Database Management

Database management functions provide for tracking, retrieval, storage, update, protection, and archiving of stored data.

page 29. The Process for Selecting Geographic Information Systems

Adding a column to a table

Adding a column to a table is relatively straightforward in SQLite. The original *How do I do that in ArcGIS/Manifold* document did not include options like changing the name, data type, or size of the columns. Nonetheless, other books in the *How do I* series show how to ALTER columns. Unfortunately, SQLite has a very limited ALTER TABLE statement, and only allows users to add columns – you cannot remove or change the column type.

To add a new column (in this case, a column named **HomeAge** that stores Integers, the user simply writes:

Add a column

```
ALTER TABLE parcels
ADD COLUMN homeage INTEGER
```

Sorting tabular or graphical data

Using SQL, we can sort the data on any field, combination of fields, or even an on-the-fly calculated field. Sorting can be done in either ascending or descending order.

Sort in descending order

```
SELECT parcelkey, asmt FROM parcels
ORDER BY asmt DESC
```

Sort multiple fields

```
SELECT parcelkey, acres, asmt, addrstreet FROM parcels
ORDER BY asmt, acres
```

Sort using an on-the-fly calculation

```
SELECT parcelkey, (asmt - land) AS structurevalue
FROM parcels
ORDER BY structurevalue DESC
```

Calculating values for new fields using arithmetic or related tables
- making field calculations.

New values may be calculated using the `UPDATE` statement, or values can be calculated on-the-fly without changing the actual data in a table. You can calculate data as a result of an SQL `SELECT` query, or as a calculation into an existing field. Generally, calculating new values is performed using the `UPDATE` statement on a column for a table.

The following example updates a field named homevalue in our parcels table by subracting the value of the land from the entire assessment value of the property:

```
UPDATE parcels
SET homevalue =  parcels.asmt - parcels.land
```

Calculations on a related table

You can also use a table relation, and update the value in one table with values in another table that are related by a common field. In this example we are subtracting the land value in the parvalues table from the asmt value in the parcels table. To accomplish this, we are explicitly stating that we want to UPDATE parcels, but are referencing the parvalues table in the FROM directive:

```
UPDATE parcels
SET homevalue = (select parcels.asmt - parvalues.land
                 from parvalues
                 where parcels.parcelkey = parvalues.parcelkey)
WHERE EXISTS (select *
           from parvalues
           where parcels.parcelkey = parvalues.parcelkey)
```

Calculations without updating the table

The following performs a calculation on-the-fly to determine the tax amount for a property, based upon its assessed value as part of a `SELECT` query, without updating an existing field:

```
SELECT (parcels.asmt * .07) AS TaxValue
FROM parcels
```

Relating data files and fields

SQL relates can work on more than one table. This example illustrates how to relate multiple tables together based on a related item. While many complex relates are possible in SQL, simple tables are used here to illustrate the process. For example, we can join two tables (parcels and propclas) as:

```
SELECT parcels.parcelkey, parcels.propclass,propclas.description
FROM parcels, propvalues AS propclas
WHERE parcels.propclass = propclas.propvalue
```

Unfortunately, the previous query will return an error because the *value* field in the propclas table is represented as a text value, while the *propclass* field in the parcels table is an Integer value. However, SQL can create the relationship by simply changing the data type of the *value* field on-the-fly to an Integer using the CAST operator:

```
SELECT parcels.parcelkey,
parcels.propclass,propvalues.description
FROM parcels, propvalues
WHERE parcels.propclass = Cast(propvalues.propvalue as integer)
```

Database Creation

Database creation functions are those functions required to convert spatial data into a digital form that can be used by a GIS. This includes digitizing features found on printed maps or aerial photographs and transformation of existing digital data into the internal format of a given GIS.

Page 29, The Process for Selecting Geographic Information Systems

Digitizing

Ordinarily, digitizing is performed within the GUI of a GIS where the user points-and-clicks on the screen. However, sometimes SQL can be used if you have a table of coordinate values you want to enter in, or if you are receiving input from say an Internet based application. The following examples illustrate how to insert geometries into an existing layer.

Points are created from two coordinate values using the ST_MakePoint function. Those points must then be assigned a coordinate system (SRID) using the ST_SetSRID function as:

```
SetSRID(ST_MakePoint(X,Y),SRID number))
```

In this example, we are inserting a point and an OID value into the “trees” table and assigning it to New York State Place Central Zone (2261):

```
INSERT into trees(geometry)
Values(SetSRID(MakePoint(833240,888478),2261));
```

If you had a table of X and Y values, you could insert all of them into a new layer as:

```
INSERT INTO trees (geometry)
SELECT SetSRID(MakePoint(easting, northing),2261)
FROM treetable
```

Lines are created from a series of point geometries. To add a Line, you could string together a series of ST_MakePoint functions. In the following example we are adding New York State Plane Central Zone (SRID 2261) points and converting them into a line using ST_MakeLine:

```
INSERT INTO hydro (geometry)
Values(SetSRID(MakeLine(MakePoint(808647,942424),
                            MakePoint(753828, 889564)),2261))
```

Areas are created as a series of coordinates. While you can string together a series of points, the following example uses the Well Known Text (WKT) format to create a linestring and then the ST_MakePolygon function to convert the linestring to an area:

```
INSERT INTO parks (name,geometry)
Values("artpark",SetSRID(ST_MakePolygon
 (ST_GeomFromText('LINESTRING(849268 886123,808647 893424,897647
                              893424, 849268 886123)')),2261)
       )         )
```

Assigning Topology - identifying intersection points

You can find intersection points for either line or area geometries in a single layer as:

```
ST_Intersection(geometry, geometry)
```

Intersections on a single layer

```
SELECT ST_Intersection(hydro.geometry,hydro.geometry)
FROM hydro
```

Intersections on multiple layers[2]

```
SELECT ST_Intersection(parks.geometry,hydro.geometry)
FROM parks, hydro
WHERE ST_Intersects(parks.geometry,hydro.geometry)
```

Also, you can insert the intersection points into a new layer using the `INSERT` command:

```
CREATE TABLE scratchlayer AS
SELECT ST_Intersection(parks.geometry,hydro.geometry) as
geometry
FROM parks, hydro
WHERE ST_Intersects(parks.geometry,hydro.geometry)
```

Creating a Polygon from Line Segments

Creating a polygon from multiple line segments (spaghetti) is not possible in Spatialite.

[2] Combining ST_Intersection with ST_Intersects accomplishes two things: it speeds up the query because intersection points are only created for those lines that intersect. Also, it reduces the number of *null* values returned because if you ask PostGIS to return the ST_Intersection of two lines that don't intersect, it will obey your wish: that is, it will return a null value.

Creating a distance buffer from line segments

Buffers can be created on any type of geometry, either points, lines, or areas - using the Buffer statement as:

```
ST_Buffer(geometry,distance)
```

The units can be virtually any unit such as meters, feet, miles, kilometers, etc.

Buffer with a constant value

```
SELECT ST_buffer(geometry,50)
FROM hydro
```

Buffer with an attribute assigned value

```
SELECT ST_buffer(geometry,hydro.length)
FROM hydro
```

Creating ringed buffers around a geometry[3]:

```
SELECT name, ST_Buffer(geometry, 50) AS g
FROM parks

UNION ALL
SELECT name, ST_Buffer(geometry, 30) AS g
FROM parks

UNION ALL
SELECT name, ST_Buffer(geometry, 20) AS g
FROM parks
```

Another approach to a ringed buffer is to use the VALUES statement to return a virtual table of the three distances. In this case, SQLite returns an empty column (“”), and that column must be renamed in the query.:

```
SELECT parks.name, "" AS distbuf, ST_Buffer(geometry,"") as g
```

[3] UNION ALL assumes that the table structure for each query is the same - therefore, you must have the same fields and field types.

```
FROM parks,  (VALUES (10),(20),(30))
```

Correcting topological errors - eliminating overlaps, undershoots, and dangles.

PostGIS does not natively eliminate overlays, undershoots, and dangles.

Import and export - importing database tables, raster data, and vector data

Database Tables

Spatialite does not allow for the import of export of external data through the SQL window. You must import the data using the GUI.

Raster Data

Spatialite does not support raster functions.

Data Manipulation and Analysis

Data manipulation and analysis functions provide the capability to selectively retrieve, transform, restructure, and analyze data.

Retrieval options provide the ability to retrieve either graphic features or feature attributes in a variety of ways. Transformation includes both coordinate/projection transformations and coordinate adjustments. Data restructuring includes the ability to convert vector data to raster data, merge data, compress data, reclassify or rescale data, and contour, triangulate, or grid random or uniformly spaced z-value data sets

Analysis functions differ somewhat depending on whether the internal data structure is raster or vector based. Analysis functions provide the capability to create new maps and related descriptive statistics by reclassifying and combining existing data categories in a variety of ways. Analysis functions also support: replacement of cell values with neighboring cell characteristics (neighborhood analysis); defining distance buffers around points, lines and areas (proximity analysis); optimum path or route selection (network analysis); and generating slope, aspect and profile maps (terrain analysis).

Page 29, The Process for Selecting Geographic Information Systems.

Vector Retrieval - Select by Window

Although *select by window* normally assumes an interactive session with the GUI, you can use SQL to select by geometric shape. In this example, we are entering a WKT expression for a triangle. All vector features intersecting the box are selected. Instead of intersecting, the query could use contains or touches[4]:

```
SELECT parcels.*
FROM parcels,
  (VALUES (SetSRID(MakePolygon(ST_GeomFromText
  ('LINESTRING(849268 886123,808647 893424,897647 893424,
            849268 886123)')),2261))
  ) AS g
WHERE ST_Intersects(geometry,"")
```

In the above example, the polygon is simply a triangle. You could easily add more points to create a more detailed polygon, or you could create a rectangle.

Raster Retrieval - select data by area masks

Spatialite does not support raster functions.

Data restructuring - convert from raster to vector; and vector to raster

Raster to Vector

Spatialite does not support raster functions.

Modify raster cell size by resampling

Spatialite does not support raster functions.

[4] The VALUES statement returns a table, with a column named Column1.

Reducing unnecessary coordinates - weeding

You can simplify a geometry using the Douglas-Peucker algorithm using the ST_Simplify functions as:

```
SELECT ST_Simplify(geometry,30) FROM hydro
```

Smoothing data to recover sinuosity

Smoothing data in Spatialite is currently not supported.

Data restructuring - changing raster values by selected area or feature

Spatialite does not support raster functions.

Generate a TIN from point data

Using a set of points, we can generate Triangulated Irregular Networks (TIN) using the DelaunayTriangulation function. In order to do that, all of the point geometries need to be joined together with the ST_Union function. An example is:

```
SELECT DelaunayTriangulation(ST_Union(geometry)) AS geometry
FROM sixcities
```

Kriging from point data

Kriging is not performed in Spatialite.

Generate contour data from points

Spatialite does not create contours from points.

Generate contour data from raster

Spatialite does not create contours from a raster file.

Data Transformation - mathematical transformation of raster data

Spatialite does not support raster functions.

Projection definition and coordinate transformation

Geometries may be projected and defined on-the-fly with SQL in PostGIS. The following query projects the geometry from the parcels layer stored as *State Plane, NY Central* to the *UTM Zone 18* coordinate system:

Change Projection

```
SELECT ST_Transform(geometry,3450)
FROM parcels;
```

Define Projection

If the geometry is in the correct numerical format, but does not have a coordinate system assigned, you can assign a coordinate system. For example, assume that the parcel layer does not have a coordinate system defined, but should actually be UTM 18N. The query would be:

```
SELECT UpdateGeometrySRID('elevpts','geometry',3450)
```

Vector overlay - polygon in polygon overlay; point in polygon overlay; line in polygon overlay

Overlaying and finding geometry features contained within polygons are the same whether using points, lines, or polygons, and would take the form of:

```
SELECT parcels.*
     FROM parcels, parks
WHERE ST_Intersects(parks.geometry,parcels.geometry)
```

in this case, it does not matter whether the parcels are points, lines, or polygons. In addition to `ST_Intersects`, other operations may be used and include `ST_Touches`, `ST_Contains`, or `ST_Overlaps`.

Raster processing - mathematical operations on one raster (sine, cosine, exponent)

Spatialite does not support raster functions.

Raster processing - mathematical operations on two rasters - adding, subtracting, minimum, maximum

Spatialite does not support raster functions.

Neighborhood functions - average, minimum, maximum, most frequent

Spatialite does not support raster functions.

Statistical functions - calculating areas, perimeters and lengths

Descriptive statistics on geometries can be calculated to include area, length, or perimeter. In the example below, "parks" are an area feature so to obtain the length, we are extracting the boundary from the area.

```
SELECT ST_Area(geometry) AS area,
       ST_Length(ST_Boundary(geometry)) AS perimeter
FROM parks
```

Cross tabulation of two data categories

Spatialite does not support the creation of cross tabulation matrices. However, the following query will create an n x 3 table that summarizes the data. The reader will still have to convert the n x 3 table into an n x n table with unique names in one of the columns acting as column headings.

```
SELECT  name , zone,
     Sum(ST_Area(ST_intersection
                           (parks.geometry,firm.geometry))/43560)
                          AS sumArea
    FROM parks, firm
    WHERE ST_Intersects(parks.geometry,firm.geometry)
    GROUP BY name,zone
    ORDER BY 1,2
```

General - specify distance buffers; polygons within a distance of selected buffers; find nearest features

The ability to specify distance buffers was already addressed in a previous section. However, finding polygons within a specified distance and finding nearest features are calculated as:

<u>Polygons within a distance</u>

```
SELECT parcels.*
FROM parcels, parks
WHERE ST_Distance(parcels.geometry,parks.geometry) < 500
```

<u>Nearest Features</u>

To find the nearest feature, you must first calculate the distance between all the features in both layers. After that calculation (shown as `mindist` in the subquery below), you must create another virtual table of all the distances between the layers and perform a `LEFT JOIN` on the second layer where the calculated distance between the two layers is the same as the minimum distance for the first layer. This query finds the closest park from each tree - however, to speed the query up, a distance threshold of 200 ft. is used as a cutoff value.

```
SELECT site_id, park_no, minium_distance
FROM
  (SELECT min(dist) AS minium_distance,site_id
   FROM (SELECT parks.park_no, trees.site_id,
         st_distance(parks.geometry, trees.geometry) AS dist
         FROM parks, trees
         WHERE st_distance(parks.geometry,trees.geometry) < 200
         ORDER BY dist, park_no
         ) AS mindist
  GROUP BY site_id
  ) AS T
LEFT JOIN (SELECT park_no, site_id s_id,
           st_distance(parks.geometry, trees.geometry) AS dd
           FROM parks, trees) AS T2
ON
   site_id = s_id AND dd = minium_distance
```

3D analysis - generating slope and aspect

Spatialite does not support raster functions.

Identifying watersheds

Spatialite does not support the creation of watersheds.

Network functions - choosing the optimal path through a network

Creating a routable network using Spatialite is beyond the scope of this document. However, the Spatialite GUI does provide a relatively easy approach to create a routable network. The following discussion provides a quick overview of preparing a node table and routing between two nodes:

Create a node table

```
CREATE TABLE nodes AS
SELECT startpoint(geometry) AS geometry, fnode_ AS node
FROM streets
UNION ALL
SELECT endpoint(geometry) as geometry, tnode_ AS node
FROM streets
```

Route the network

```
CREATE TABLE routing AS
SELECT *
FROM streets_net
WHERE NodeFrom = 3
  AND NodeTo = 382;
```

Defining a drive-time zone

Spatialite does not support creation of drive time zones.

Geocoding addresses

Spatialite does not support geocoding.

Topological intersection

SQL in PostGIS supports all of the classical topological intersection methods (intersect, union, identity) using variations of the ST_Intersection and st_Difference methods. These capabilities are more sophisticated than most SQL presented in this book. Nonetheless, the command builds upon the more foundational SQL that you have already seen. The parcel and parks layers are used as illustrative examples:

Intersect

The basic principle for intersection is to clip two area layers using the ClipIntersect function, and then joining the subsequent data tables to the clipped features.

```
DROP TABLE bobo;
CREATE TABLE bobo as
SELECT parcels.*, B.*
FROM parcels, (
     SELECT parks.*, a.*, g
     FROM parks, (
               SELECT parcels.ogc_fid as paroid, parks.ogc_fid as
poid, st_intersection(parks.geometry, parcels.geometry) as g
               FROM parcels, parks
               WHERE
st_intersects(parcels.geometry,parks.geometry)
               ) AS a
      WHERE parks.ogc_fid = a.poid) AS B
WHERE parcels.ogc_fid = paroid
```

Union

Union is slightly more complicated and requires the joining of three separate queries: clipping the two layers together with ST_Intersection, obtaining the parts of layer 1 that are not in layer 2 using ST_Difference, and obtaining the parts of layer 2 that are not in layer 1. After the geometries are assembled, the attribute tables are joined in. In this example we will overlay the firm layer and the parks layer:

```
SELECT * FROM
(
  SELECT
  ST_Intersection(firm.geometry,parks.geometry) g,
                  firm."OID"  AS cid,
                  parks."OID" AS rid
  FROM firm, parks
  WHERE ST_Intersects(firm.geometry,parks.geometry)

 UNION ALL

 SELECT
 ST_Difference(parks.geometry,firm.geometry) AS g,
              firm."OID"  AS cid,
              parks."OID"  AS rid
 FROM firm, parks

UNION ALL

 SELECT
 ST_Difference(firm.geometry,parks.geometry) AS g,
              firm."OID"  AS cid,
              parks."OID"  AS rid
 FROM firm, parks

) AS a

LEFT JOIN firm ON firm."OID" = cid
LEFT JOIN parks ON parks."OID" = rid
```

Identity

```
SELECT parcels.*, B.*
FROM parcels, (
     SELECT parks.*, a.*, g
     FROM parks, (
               SELECT parcels.ogc_fid as paroid, parks.ogc_fid
as poid, st_difference(parks.geometry, parcels.geometry) as g
               FROM parcels, parks
               WHERE
st_intersects(parcels.geometry,parks.geometry)
               ) AS a
      WHERE parks.ogc_fid = a.poid) AS B
WHERE parcels.ogc_fid = paroid
```