Zonal Statistics in PostGIS – a Tutorial

Zonal statistics is a common GIS operation but surprisingly hard to perform in POSTGIS today (2014 version 2.1). Solutions tend to involve highly complex queries, return answers for only one polygon instead of a table with a row for each polygon, and to perform incorrectly if the raster type is a float rather than an integer. Below are two solutions (one faster, one more accurate).

There are two methods to summarize a raster by polygons (e.g. a gridded temperature dataset by county). One involves converting the raster to polygons to do a polygon-polygon comparison. The other involves converting the polygon to raster to do a raster-raster comparison. The difference comes for pixels that cross the polygon boundary. Pixels wholly interior to the polygon or wholly exterior to the polygon (or more precisely in this case wholly in the interior of a different polygon) are counted the same in either method. But for pixels on the polygon boundary (or again more precisely spanning two polygons), the raster-raster method assigns the pixel to one polygon or the other (usually the polygon containing the center of the pixel), while the polygon-polygon method can calculate the proportion of the pixel in each polygon and do an area weighted approach that is more accurate. Thus the polygon-polygon methods is more accurate but it is an order of magnitude or two slower than the raster-raster method.

Raster-raster in PostGIS.

1) **Load the raster into POSTGIS (here historical minimum temperatures)**

```
raster2pgsql -s 2037 -c  -I  me_hist_tmin.tif hist_tmin |psql -U
   postgres -d dss
```

2) **Load the polygons into POSTGIS (here county boundaries for state of Maine)**

```
shp2pgsql -s 2037 -c -I counties.shp counties | psql -U postgres
   -d dss
```

3) **Issue a SQL query. This uses GROUP BY on cntycode which works because** ST_Union is a GROUP BY operator (just like SUM and MAX). So here one row is created for each polygon in the counties file by clipping the raster (here the whole 300x600 raster) by the polygon (this results in a smaller but still square raster surrounding the polygon with pixels outside the polygon set to NODATA). These separate rasters for each polygon in a county are then merged back into a larger raster by ST_Union which will have values (instead of NO DATA) only in areas where the raster is inside the county. This raster is then reduced to mean/sum/min/max/count by the ST_SummaryStats function which reduces a raster column to 5 columns summing across the raster (which here has already been ST_Clip'd to only have valid values within the county).

```
SELECT cntycode,
   (ST_SummaryStats(ST_Union(ST_Clip(rast,geom)))).* FROM
   hist_tmin, counties GROUP BY cntycode;
```

Polygon-polygon in PostGIS

The normal step here is to convert each pixel into a square (or trapezoidal in some projections) polygon with its associated value. Then one intersects each square polygon (formerly pixel) with each normal polygon (e.g. county) for area of overlap. This can then be used to give area weighted averages and sums with pixels on the boundary included proportional to the area of overlap.

The program is the GDAL/POSTGIS stack doesn't have a good generic solution for this POSTGIS has ST_DumpAsPolygons function which is a wrapper around the GDAL Polygonize API (also exposed in the gdal_polygonize.py command line utility). However this function is inadequate for the current purpose – it merges pixels with the same value into larger polygons. In principle this should still be acceptable but the actual implementation (in a poorly documented "feature") does this by converting every value to an integer. So pixels with a temperature of 13.49 and 12.51 are merged incorrectly (and the value stored is 13 which is also incorrect). The trick is to use the POSTGIS commandline tool raster2pgsql combined with POSTGIS functions to achieve the desired one rectangle pixel per polygon with non-integer values retained correctly.

1) **Load the polygons into POSTGIS (here county boundaries for state of Maine)**

```
shp2pgsql -s 2037 -c -I counties.shp counties | psql -U postgres
   -d dss
```

2) **The raster should ideally have points outside all polygons (e.g. outside the state** of Maine in my example) already clipped to be NODATA (its not required but it speeds things up). Load the raster into POSTGIS using the -t switch in the somewhat orthodox fashion of breaking the raster up into tiles of 1x1 pixel (normally the -t option is used to break very large tiles up into smaller, say 64x64 pixel tiles as separate rows in one table, which is the preferred way to store large rasters in POSTGIS as it allows for efficient spatial indexing (POSTGIS spatial indexing only works across rows and thus cannot "reach into" a large raster stored as a single row). Here we just go to the extreme and store each pixel in a separate row. This is incredibly wasteful in terms of disk storage and read/writes. It shouldn't be used as a permanent way to the store the raster in POSTGIS (hence I include code to deleet this table). But is overall the path to the fastest correct polygon-polygon method that I have found. Use:

```
raster2pgsql -d -s 2037  -I -t 1x1 me_hist_tmin.tif
   mht_poly|psql -U postgres -d dss >c:\temp\rast.log
```

3) **Now we need to convert this strange storage of a raster into a parallel table** where there is one row for each pixel with the polygon stored in a "geom" column and the value stored in a 2nd column. Although in principle this could be done as part of a single complex WITH query, the ability to create an index on the geom column pays for itself several times over speedwise, so I create it as a table and then create an index. This index would also payoff even more if one forgot to index the 'geom' column in the original polygon file from step 1 (done by the –I command to shp2pgsql). I also calculate the area of each pixel in case the projection is not an equal area projection).

```
DROP TABLE IF EXISTS mht_poly_use;
SELECT ST_ConvexHull(rast) AS pixelgeom,
   ST_Area(ST_ConvexHull(rast)) AS
   pixelarea,(ST_SummaryStats(rast)).sum AS pixelval INTO
   mht_poly_use FROM mht_poly WHERE (ST_SummaryStats(rast)).sum
   IS NOT NULL;
```

```
CREATE INDEX idx_pixelgeom ON mht_poly_use USING
   GIST(pixelgeom); -- costs 6-8 sec but saves about 35 seconds
   (10% reduction) - also a good backup incase a future user
   forgets to index the polygon geometry
```

4) **Now comes a query which creates an intermediate table that has a row for each** pixel polygon that intersects each original polygon (e.g. county) and then calculating resulting areas and carrying along the raster pixel value. This is a "spatial join" and it is orders of magnitude faster if the two geometry columns are spatially indexed and a WHERE clause testing that the pixels intersect (which uses the spatial index) before creating a row. This temporary table is passed using the WITH clause into the final calculation which uses GROUP BY and appropriate area weighting formulas to get summary statistics. The answer is what comes out of this query (which could be used in SELECT INTO to create a new table with the final result).

```
WITH temp_table AS (SELECT
   ST_Area((ST_Intersection(pixelgeom,geom)).geometry) AS
   intersectarea, pixelval, pixelarea AS origarea, cntycode FROM
   mht_poly_use, counties WHERE ST_Intersects(pixelgeom,geom)
)
SELECT cntycode, SUM(pixelval*intersectarea/origarea) AS sums,
   SUM(pixelval*intersectarea)/SUM(intersectarea) AS means,
   COUNT(*) AS counts, MAX(pixelval) AS maxes, MIN(pixelval) AS
   mins, SUM(intersectarea)/1000000 AS area from temp_table
   group by cntycode;
```

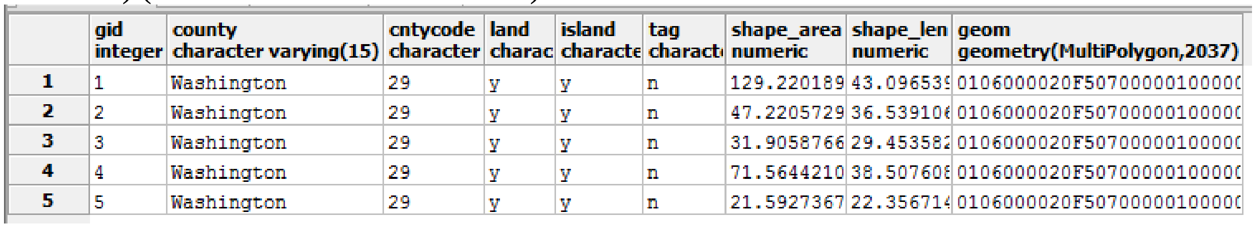
5) Finally, clean up the intermediate tables

```
DROP TABLE IF EXISTS mht_poly_use;
DROP TABLE IF EXISTS mht_poly;
```

In principle this method should also work for polygon-polygon zonal statistics (i.e. where the statistics are in one shapefile, e.g. population on census blocks, and the larger aggregating regions, e.g. counties, are in a separate shapefile) with little modification (step 2 would look like step 1 and step 3 would be dropped).

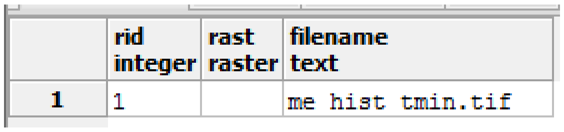
In my test case the raster is roughly 300x600 pixels and the polygon data represents, 16 counties broken into 6,890 polygons – lots of islands in Maine – stored as POLYGON with a code linking multiple polygons within one county, not as MULTIPOLYGON). The polygon-polygon method takes about five minutes on a laptop (client and server on same laptop). The raster-raster method takes about two minutes. For my application (many pixels inside the mainland polygon, but also for some counties many island polygons so small they don't include the center of the pixel they occur within) the differences for inland counties (represented as one or a few large polygons) were very small (usually in the 3rd or 4th significant digit, i.e. 0.1 °C or 0.01 °C), but for coastal counties with many (often hundreds) of very small off-shore islands, the means were close (usually 3rd significant digit), but the count of # of pixels included and the resulting sums were about 20% smaller. The max and min were surprisingly close even in the coastal counties, but this relies on the fact that my raster (temperature) was also strongly autocorrelated/spatially smooth. More rugged, heterogenous data would see larger effects on min and max. Different ratios between polygon size and raster size would of course give different amounts of accuracy, but this data nicely captures the endpoints – many pixels in few large polygons means the two methods are very close, while many small polygons (especially even smaller than the pixels) combined with very rugose boundaries give unacceptable errors for sum but probably acceptable results for mean.

The counties table I used (loaded from a shapefile already projected into the same projection as the raster) (and limited to the first 5 rows):



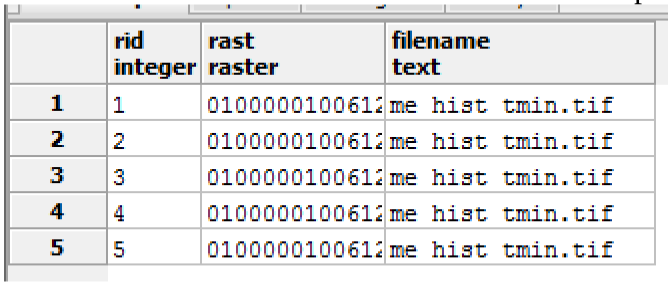
	gid integer	county character varying(15)	cntycode character	land charac	island characte	tag characte	shape_area numeric	shape_len numeric	geom geometry(MultiPolygon,2037)
1	1	Washington	29	y	y	n	129.220189	43.09653	0106000020F5070000010000
2	2	Washington	29	y	y	n	47.2205729	36.53910	0106000020F5070000010000
3	3	Washington	29	y	y	n	31.9058766	29.45358	0106000020F5070000010000
4	4	Washington	29	y	y	n	71.5644210	38.50760	0106000020F5070000010000
5	5	Washington	29	y	y	n	21.5927367	22.35671	0106000020F5070000010000

Here is the raster table loaded in the raster-raster method



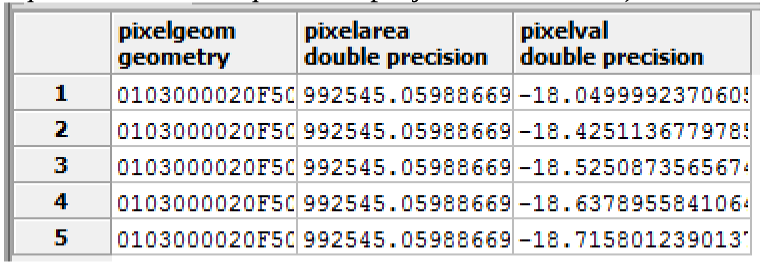
	rid integer	rast raster	filename text
1	1		me hist tmin.tif

And the first few rows of the raster loaded in the polygon-polygon method (1x1=pixel tiles)



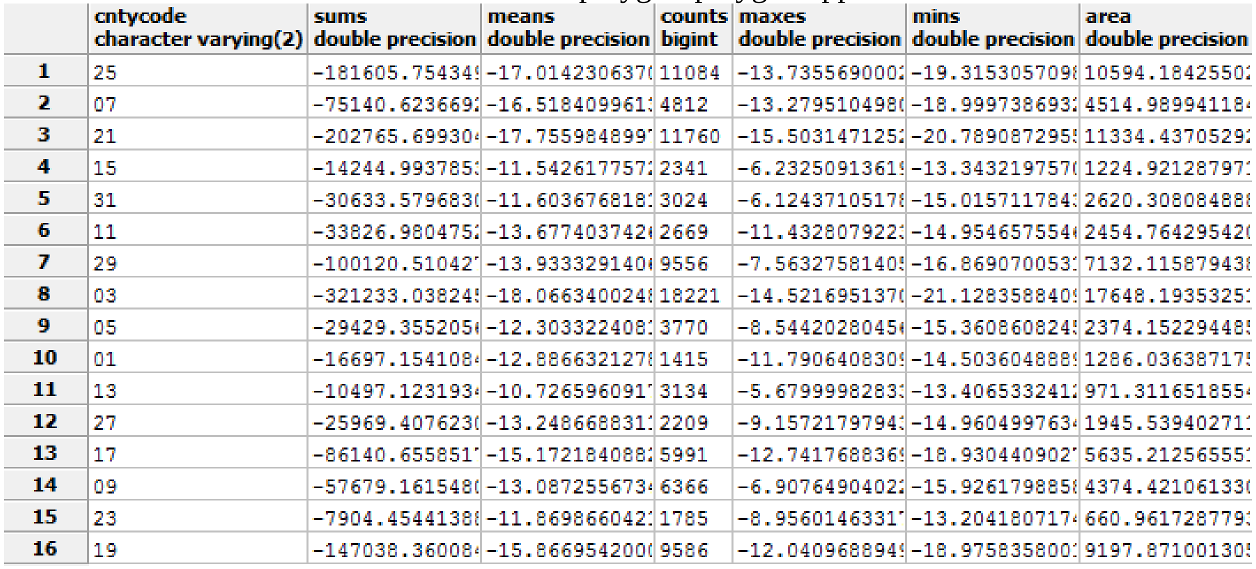
	rid integer	rast raster	filename text
1	1	010000010061	me hist tmin.tif
2	2	010000010061	me hist tmin.tif
3	3	010000010061	me hist tmin.tif
4	4	010000010061	me hist tmin.tif
5	5	010000010061	me hist tmin.tif

And the intermediate table (called mht_poly_use in the code above) from the raster-raster method (note that the raster column has been converted to a geometry column AND a pixelval column and that the area of the pixel has been calculated for future use (here the areas are all equal because an equal-area projection was used).



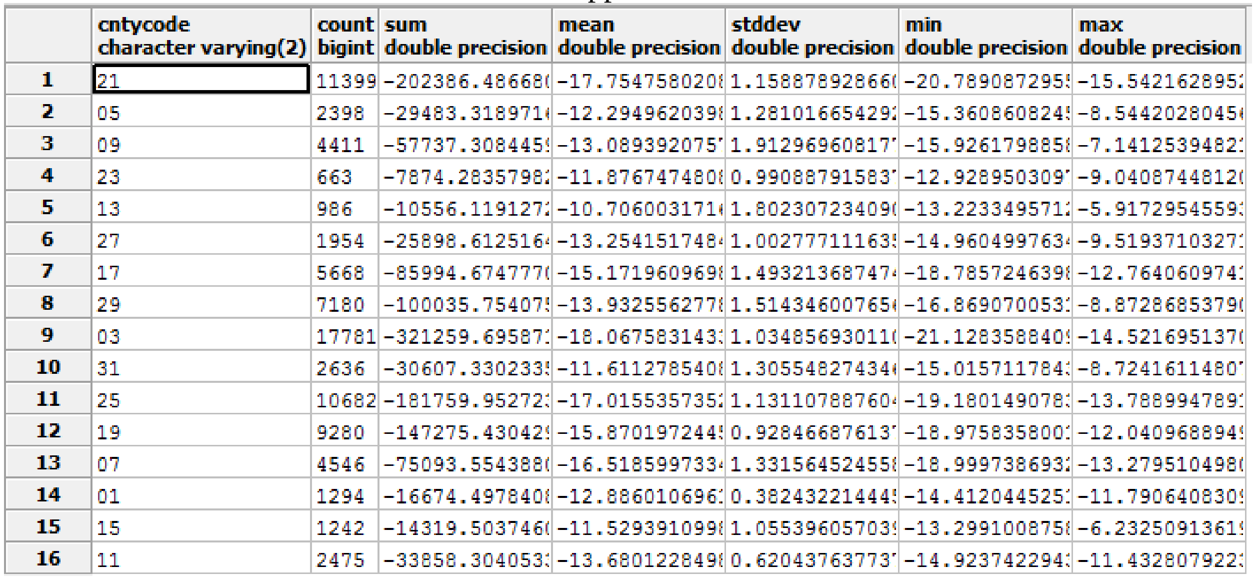
	pixelgeom geometry	pixelarea double precision	pixelval double precision
1	0103000020F50	992545.05988669	-18.04999923706
2	0103000020F50	992545.05988669	-18.42511367797
3	0103000020F50	992545.05988669	-18.52508735656
4	0103000020F50	992545.05988669	-18.63789558410
5	0103000020F50	992545.05988669	-18.71580123901

And here is the results for the more accurate polygon-polygon approach:



	cntycode character varying(2)	sums double precision	means double precision	counts bigint	maxes double precision	mins double precision	area double precision
1	25	-181605.75434	-17.014230637	11084	-13.735569000	-19.315305709	10594.1842550
2	07	-75140.623669	-16.518409961	4812	-13.279510498	-18.999738693	4514.98994118
3	21	-202765.69930	-17.755984899	11760	-15.503147125	-20.789087295	11334.4370529
4	15	-14244.993785	-11.542617757	2341	-6.2325091361	-13.343219757	1224.92128797
5	31	-30633.579683	-11.603676818	3024	-6.1243710517	-15.015711784	2620.30808488
6	11	-33826.980475	-13.677403742	2669	-11.432807922	-14.954657554	2454.76429542
7	29	-100120.51042	-13.933329140	9556	-7.5632758140	-16.869070053	7132.11587943
8	03	-321233.03824	-18.066340024	18221	-14.521695137	-21.128358840	17648.1935325
9	05	-29429.355205	-12.303322408	3770	-8.5442028045	-15.360860824	2374.15229448
10	01	-16697.154108	-12.886632127	1415	-11.790640830	-14.503604888	1286.03638717
11	13	-10497.123193	-10.726596091	3134	-5.6799998283	-13.406533241	971.311651855
12	27	-25969.407623	-13.248668831	2209	-9.1572179794	-14.960499763	1945.53940271
13	17	-86140.655851	-15.172184088	5991	-12.741768836	-18.930440902	5635.21256555
14	09	-57679.161548	-13.087255673	6366	-6.9076490402	-15.926179885	4374.42106133
15	23	-7904.4544138	-11.869866042	1785	-8.9560146331	-13.204180717	660.961728779
16	19	-147038.36008	-15.866954200	9586	-12.040968894	-18.975835800	9197.87100130

And the less accurate but faster raster-raster approach:



	cntycode character varying(2)	count bigint	sum double precision	mean double precision	stddev double precision	min double precision	max double precision
1	21	11399	-202386.48668	-17.754758020	1.15887892866	-20.789087295	-15.542162895
2	05	2398	-29483.318971	-12.294962039	1.28101665429	-15.360860824	-8.5442028045
3	09	4411	-57737.308445	-13.089392075	1.91296960817	-15.926179885	-7.1412539482
4	23	663	-7874.2835798	-11.876747480	0.99088791583	-12.928950309	-9.0408744812
5	13	986	-10556.119127	-10.706003171	1.80230723409	-13.223349571	-5.9172954559
6	27	1954	-25898.612516	-13.254151748	1.00277711163	-14.960499763	-9.5193710327
7	17	5668	-85994.674777	-15.171960969	1.49321368747	-18.785724639	-12.764060974
8	29	7180	-100035.75407	-13.932556277	1.51434600765	-16.869070053	-8.8728685379
9	03	17781	-321259.69587	-18.067583143	1.03485693011	-21.128358840	-14.521695137
10	31	2636	-30607.330233	-11.611278540	1.30554827434	-15.015711784	-8.7241611480
11	25	10682	-181759.95272	-17.015535735	1.13110788760	-19.180149078	-13.788994789
12	19	9280	-147275.43042	-15.870197244	0.92846687613	-18.975835800	-12.040968894
13	07	4546	-75093.554388	-16.518599733	1.33156452455	-18.999738693	-13.279510498
14	01	1294	-16674.497840	-12.886010696	0.38243221444	-14.412044525	-11.790640830
15	15	1242	-14319.503746	-11.529391099	1.05539605703	-13.299100875	-6.2325091361
16	11	2475	-33858.304053	-13.680122849	0.62043763773	-14.923742294	-11.432807922

Compare results for an island (few large polygons) county like #21 (Piscataquis – a single polygon) vs a coastal (many small polygons county) county like #13 (Knox)