Writing better PostGIS queries

with PostGIS 2.1 and PostgreSQL 9.3

Created by Regina Obe

http://www.postgis.us http://www.bostongis.com
http://www.postgresonline.com http://www.paragoncorporation.com

0

Make sure we have PostGIS and hstore extensions

```
CREATE EXTENSION hstore;
CREATE EXTENSION postgis;
```

Download sample data from OpenStreetMap

```
wget --progress=dot:mega -O "portland.osm"
  "http://www.overpass-api.de/api/xapi?*
[bbox=-122.7298,45.4946,-122.5599,45.5985][@meta]"
```

Load OSM data into PostGIS with osm2pgsql

```
osm2pgsql -d foss4g2014 -H localhost -U postgres -P 5443 -S
        default.style --hstore-all portland.osm
```

planet_osm tables

You'll end up with tables:

```
SELECT f_table_name As t, f_geometry_column As gc, srid, type
FROM geometry_columns;
```

```
       t          | gc  |  srid  |    type
--------------------+-----+--------+------------
planet_osm_point   | way | 900913 | POINT
planet_osm_line    | way | 900913 | LINESTRING
planet_osm_polygon | way | 900913 | GEOMETRY
planet_osm_roads   | way | 900913 | LINESTRING
(4 rows)
```

Altering spatial data type of a column

Scenario: You loaded data in Web Mercator, but decide you really need the measurement accuracy of geography

The long way: Create column with new type, updated the new, drop the old.

Quicker: Use ALTER COLUMN

```
ALTER TABLE planet_osm_polygon
ALTER COLUMN way TYPE geography(MULTIPOLYGON,4326)
USING ST_Transform(ST_Multi(way), 4326);
```

Storing as both geometry and geography

You may have a database that is heavily used for both proximity and mapping. So you want both geography and geometry. If you have large geometries, maintaining two columns might be the best options.

Create a new geography column from a geometry column

```
ALTER TABLE planet_osm_point ADD geog geography(POINT,4326);
UPDATE planet_osm_point SET geog = ST_Transform(way,4326)::geography;
CREATE INDEX idx_planet_osm_point_geog ON planet_osm_point USING gist (geog);
```

What geography layers do we have?

```
SELECT f_table_name As t, f_geography_column As gc, srid, type
FROM geography_columns;
```

```
        t          |  gc  | srid |     type
-------------------+------+------+--------------
 planet_osm_point   | geog | 4326 | Point
 planet_osm_polygon | way  | 4326 | MultiPolygon
```

Proximity problems: Find the objects within X distance of a reference object

Which restaurants are within 1KM?

Slow: Use ST_Distance

```
SELECT name
FROM
    planet_osm_point CROSS JOIN
    geography(ST_Point(-122.66317,45.5284571)) As loc
WHERE ST_Distance(geog,loc) <= 1*1000 AND tags @> 'amenity=>restaurant'
ORDER BY name;
```

```
name
----------------------
 Alexis Restaurant
 Bellagios Pizza
 Burnside Brewing Co.
 Dixie Tavern
 Doug Fir Lounge
 Frank's Noodle House
 House of Louie
 :
```

ST_Distance can't take advantage of indexes, so this solution doesn't scale well.

Better: Use ST_DWithin

```
SELECT name
FROM
    planet_osm_point CROSS JOIN
    geography(ST_Point(-122.66317,45.5284571)) As loc
WHERE ST_DWithin(geog, loc, 1*1000) AND tags @> 'amenity=>restaurant'
ORDER BY name;
```

```
name
----------------------
 Alexis Restaurant
 Bellagios Pizza
 Burnside Brewing Co.
 Dixie Tavern
 Doug Fir Lounge
 Frank's Noodle House
 House of Louie
 :
```

Can utilize a spatial index

If you just want a count of items, don't order and don't output the records

```
SELECT count(1)
FROM
    planet_osm_point CROSS JOIN
    geography(ST_Point(-122.66317,45.5284571)) As loc
WHERE ST_DWithin(geog, loc, 1*1000) AND tags @> 'amenity=>restaurant';
```

Outputting data (network effects) and ordering are very costly and it often overshadows the other costs. So if you don't care about the actual data, just count.

ST_Distance vs ST_DWithin

ST_DWithin can use an index, but ST_Distance cannot.

ST_DWithin will generally be faster.

Textual Plan

Part of plan showing index

```
:
-> Index Scan using idx_planet_osm_point_geog on planet_osm_point
    (cost=0.28..8.55 rows=1 width=75)
    (actual time=0.408..3.059 rows=20 loops=1)
Index Cond: (geog && _st_expand(loc.loc, 1000::double precision))
      Filter: ((tags @> '"amenity"=>"restaurant"'::hstore) AND
      (loc.loc && _st_expand(geog, 1000::double precision)) AND
      _st_dwithin(geog, loc.loc, 1000::double precision, true))
      Rows Removed by Filter: 1630
```

Planning time: 0.353 ms Execution time: 3.242 ms

Query geometry and hstore together

How do you get the planner to use an index for both?

Common approach: Create two GiST indexes

```
CREATE INDEX idx_planet_osm_point_tags ON planet_osm_point USING gist (tags);
CREATE INDEX idx_planet_osm_point_geog ON planet_osm_point USING gist (geog);
```

The planner might use both utilizing a bitmapscan. The planner might just choose one over the other.

Another approach: Compound GIST index?

You can combine geometry/geography and hstore in same GiST index, but its fatter, but planner is more likely to leverage both parts.

```
DROP INDEX IF EXISTS idx_planet_osm_point_geog;
DROP INDEX IF EXISTS idx_planet_osm_point_tags;
CREATE INDEX idx_planet_osm_point_geog_tags
    ON planet_osm_point USING gist (geog, tags);
```

Revisit our example of restaurants within 1KM using compound index. Compare the plan to geometry index plan.

```
SELECT name
    FROM planet_osm_point CROSS JOIN
     geography(ST_Point(-122.66317,45.5284571)) As loc
      WHERE ST_DWithin(geog,loc, 1*1000) AND
        tags @> 'amenity=>restaurant'
    ORDER BY name;
```

Text Explain with compound index

Note the geography expand and tags check are used in the index condition.

```
:
      Index Cond: ((geog && _st_expand(loc.loc, 1000::double precision)) AND
        (tags @> '"amenity"=>"restaurant"'::hstore>))
      Filter: ((loc.loc && _st_expand(geog, 1000::double precision)) AND
        _st_dwithin(geog, loc.loc, 1000::double precision, true))
      Rows Removed by Filter: 17
```

Planning time: 0.342 ms Execution time: 1.424 ms

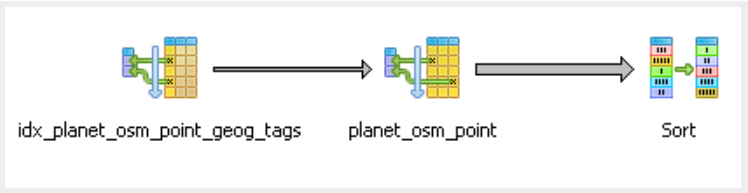
It's twice as fast as using the single geography index (not considering network effects).

What if your query involves only the second column in your compound index

Example: All Mexican restaurants

```
SELECT name
 FROM planet_osm_point
 WHERE tags @> 'amenity=>restaurant, cuisine=>mexican'
ORDER BY name;
```

Compound gist indexes can also service queries that only involve one element in the index, even if the element is second column in index.



Partial Explain using a compound index when you are only querying one column

Only querying the tags column, compound index still kicks in

```
:
Recheck Cond: (tags @> '"amenity"=>"restaurant", "cuisine"=>"mexican"'::hstore)
Rows Removed by Index Recheck: 4
Heap Blocks: exact=27
->  Bitmap Index Scan on idx_planet_osm_point_geog_tags
:
  Index Cond: (tags @> '"amenity"=>"restaurant", "cuisine"=>"mexican"'::hstore)
```

Planning time: 0.140 ms
Execution time: 1.168 ms

Same query using Single tags index

```
:
    Recheck Cond: (tags @> '"amenity"=>"restaurant", "cuisine"=>"mexican"'::hstore)
    Rows Removed by Index Recheck: 4
    Heap Blocks: exact=27
    ->  Bitmap Index Scan on idx_planet_osm_point_tags
   :
          Index Cond: (tags @> '"amenity"=>"restaurant", "cuisine"=>"mexican"'::hstore)
```

Planning time: 0.138 ms

Execution time: 1.188 ms

Using the compound index and single tags index perform about the same for this.

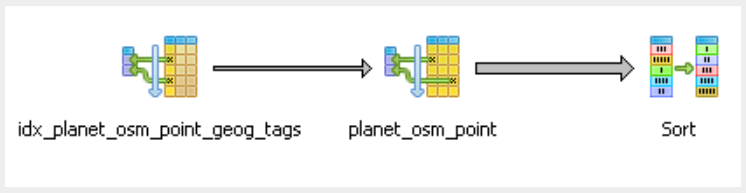
Find all things within a certain distance from me sorted by distance

Use ST_DWithin in the WHERE and ST_Distance in column output.

Closest 5 Mexican restaurants within 2km sorted by distance

```
SELECT name, ST_Distance(geog,loc) As dist
FROM planet_osm_point
    CROSS JOIN geography(ST_Point(-122.66317,45.5284571)) As loc
WHERE tags @> 'amenity=>restaurant, cuisine=>mexican'::hstore
    AND ST_DWithin(geog,loc,2*1000)
ORDER BY dist
LIMIT 5;
```

```
                name                |      dist
------------------------------------+----------------
 Aztec Willie & Joey Rose Taqueria |    1254.126992
 Robo Taco                         | 1267.277465181
 Santeria                          | 1322.559970998
 Burrito Bar                       |  1353.30868868
 Los Gorditos Perla                | 1448.850095646
```



Web Mercator for proximity analysis

Web Mercator is the commonly used projection for web mapping (OSM, Google, Bing, MapQuest). It preserves angles and shapes of small objects, but distorts size and shape of large objects. Poles are greatly distorted. As far as distance goes, distances are far from accurate. If you at the poles, you are better off using a different projection.

Restaurants within 1km in Web Mercator

Web Mercator alone: Not good

BAD: 1 km is not REALLY 1 km

Depends where in world you are how bad this is.

This get's worse the further you are from the equator.

```
SELECT name
FROM planet_osm_point CROSS JOIN
 ST_Transform(
    ST_SetSRID(ST_Point(-122.66317,45.5284571),4326),900913
    ) As loc
  WHERE
        tags @> 'amenity=>restaurant'::hstore
   AND ST_DWithin(way,loc, 1*1000)
ORDER BY name;

             name
---------------------------------
Burnside Brewing Co.
```

Recall we got more answers with geography ST_DWithin

Approach 1: ST_DWithin with Web Mercator Over-shoot by at least twice

Then do a true distance check

```
SELECT name
FROM planet_osm_point CROSS JOIN
 geography(ST_Point(-122.66317,45.5284571)) As loc
  WHERE ST_Expand(ST_Transform(loc::geometry,900913),2*1000) && way
      AND ST_DWithin(ST_Transform(way,4326)::geography,loc,1*1000)
        AND tags @> 'amenity=>restaurant'
ORDER BY name ;
```

```
             name
---------------------------------
 Alexis Restaurant
 Bellagios Pizza
 Burnside Brewing Co.
 Dixie Tavern
:
 Nicholas Restaurant
```

Now same answers as geography

Approach 2: ST_DWithin with web mercator

Use a geography functional index

Then you can skip the ST_Expand call.

```
CREATE INDEX idx_planet_osm_point_way_geog_tags
    ON planet_osm_point
  USING gist(geography(ST_Transform(way,4326)), tags);

SELECT name
FROM planet_osm_point CROSS JOIN
 geography(ST_Point(-122.66317,45.5284571)) As loc
  WHERE ST_DWithin(ST_Transform(way,4326)::geography,loc,1*1000)
        AND tags @> 'amenity=>restaurant'
ORDER BY name ;
```

Same answer shorter syntax, but may not perform well with big geometries.

Approach 3: Use a mutant geography/mercator buffer and use ST_Intersects

Warning YMMV. Not guaranteed to be right like the other especially for non-points.

```
SELECT name
FROM planet_osm_point CROSS JOIN
 geography(ST_Point(-122.66317,45.5284571)) As loc
  WHERE ST_Intersects(way,
    ST_Transform(ST_Buffer(loc,1*1000)::geometry,
        900913) )
        AND tags @> 'amenity=>restaurant'
ORDER BY name ;
```

Finding the N-closest things to me

The top 5 restaurants closest to my location

Brute-Force: Find N-closest things

```
WITH loc AS (SELECT ST_Point(-122.66317,45.5284571)::geography As loc)
SELECT name, ST_Distance(geog, loc) As dist
 FROM planet_osm_point CROSS JOIN loc
 WHERE name > ''
ORDER By dist
LIMIT 5;
```

Problem: ST_Distance can't use an index so solution doesn't scale. Will be really slow for large numbers of objects.

Total runtime: 23.768 ms

How to use a spatial index to solve N-closest things problem?

Use PostGIS KNN operators

Based on **geometry bounding box**, NOT the geometry itself

- **<->** centroid box distance
- **<#>** box distance

Index must be against a geometry constant in query and operator only uses index when used in ORDER BY clause.

Solve N-closest things with a spatial index

Remember KNN gives bounding box distance, not true distance, so we need to use a CTE to force materialize of our sample set, and then do a true distance sort. We also use a CTE to avoid repeating our location coords.

```
WITH loc AS (SELECT
    ST_Transform(ST_SetSRID(
        ST_Point(-122.66317,45.5284571),4326),
            900913) As loc) ,
s1 AS (SELECT name, ST_Distance(ST_Transform(way,4326)::geography,
         ST_Transform(loc,4326)::geography
        ) As dist
 FROM planet_osm_point CROSS JOIN loc
 WHERE name > ''
ORDER BY way <-> (SELECT loc FROM loc)
LIMIT 100)
SELECT name, dist
FROM s1
ORDER BY dist LIMIT 5;
```

```
               name                  |     dist
-------------------------------------+--------------
 Anzen                               | 148.447253838
 Northeast Martin Luther King & Hoyt | 159.138214466
 Convention Center                   | 193.226544998
 Convention Center                   | 196.543209603
 Northeast Oregon & Grand            | 209.305288106
```

KNN Explain index portion

```
:
        ->  Index Scan using planet_osm_point_index on planet_osm_point
            :
          Order By: (way <-> $1) Filter: (name > ''::text)
          Rows Removed by Filter: 836
          Buffers: shared hit=876
         ->  CTE Scan on loc loc_1  (cost=0.00..0.02 rows=1 width=32)
            :
        Sort Key: s1.dist
        Sort Method: top-N heapsort  Memory: 25kB
```

Total runtime: 4.968 ms

KNN with just point data we can do better

If you have your geometry in a measure preserving or more or less sort preserving spatial reference system, you can skip the first limit. Web mercator is not measure preserving, but for KNN sorting its pretty decent.

KNN with Web Mecator, no true distance check

Distance no good, but relative distances are

The answer is much faster and query shorter.

```
WITH loc AS (SELECT
    ST_Transform(ST_SetSRID(
        ST_Point(-122.66317,45.5284571),4326),
            900913) As loc)
SELECT name, ST_Distance(way, loc) As goofy_dist,
    ST_Distance(ST_Transform(way,4326)::geography,
        ST_Transform(loc,4326)::geography
        ) As true_dist
 FROM planet_osm_point CROSS JOIN loc
 WHERE name > ''
ORDER BY way <-> (SELECT loc FROM loc)
LIMIT 5;
```

```
               name                |    goofy_dist    |   true_dist
-----------------------------------+------------------+---------------
 Anzen                             | 211.538084278065 | 148.447253838
 Northeast Martin Luther King & Hoyt | 227.175763100742 | 159.138214466
 Convention Center                 | 276.141895004035 | 193.226544998
 Convention Center                 | 280.885631307582 | 196.543209603
 Northeast Oregon & Grand          | 298.271287716346 | 209.305288106
(5 rows)
```

Wow ordering is same as KNN with post-check, so though the distances are skewed relative point distance ordering seems to be maintained at least for around Portland and performance is much better.

Total runtime: 1.662 ms

No KNN operators for geography type

but, you can piggy back on geometry

We could in theory use geometry with geography (and utilize single index) with a super wacky index

```
CREATE INDEX idx_planet_osm_point_geog_geom_tags
  ON planet_osm_point
  USING gist
  (geog, geometry(geog), tags);
```

```
WITH loc AS (SELECT ST_Point(-122.66317,45.5284571)::geography As loc),
 pot AS (SELECT name, ST_Distance(geog, loc) As dist
 FROM planet_osm_point CROSS JOIN loc
 WHERE name > ''
ORDER BY geog::geometry <-> (SELECT loc FROM loc)::geometry
LIMIT 100
)
SELECT *
FROM pot ORDER By dist LIMIT 5;
```

```
               name                 |     dist
------------------------------------+--------------
 Anzen                              | 148.447253838
 Northeast Martin Luther King & Hoyt | 159.138214467
 Convention Center                  | 193.226544996
 Convention Center                  | 196.543209601
 Northeast Oregon & Grand           | 209.305288105
(5 rows)
```

KNN GIST geography hack plan

```
:
    ->  Index Scan using idx_planet_osm_point_geog_geom_tags on planet_osm_point
(cost=0.28..3533.26 rows=2331 width=75) (actual time=0.401..2.205 rows=100 loops=1)
          Order By: ((geog)::geometry <-> ($1)::geometry)
          Filter: (name > ''::text)
          Rows Removed by Filter: 785
```

Total runtime: 3.903 ms

KNN in geometry 4326 (Platte Carree) is okay but generally worse than Mercator

```
ALTER TABLE planet_osm_point ADD geom_4326 geometry(POINT,4326);
    UPDATE planet_osm_point SET geom_4326 = geog::geometry;

WITH loc AS (SELECT ST_SetSRID(
           ST_Point(-122.66317,45.5284571), 4326) As loc)
    SELECT name, ST_Distance(geom_4326, loc) As dist_deg
       , ST_Distance(geom_4326::geography,loc::geography) As true_dist
     FROM planet_osm_point CROSS JOIN loc
     WHERE name > ''
    ORDER BY geom_4326 <-> (SELECT loc FROM loc)
    LIMIT 5;
```

name	dist_deg	true_dist
Northeast Martin Luther King & Hoyt	0.0017368150805678	159.1382144
Convention Center	0.00185370925463961	193.2265449
Convention Center	0.00188205346321126	196.5432096
Anzen	0.001900257921027	148.4472538
Spirit of '77	0.00245076555336035	236.9106835

Different from Geography KNN hack, but not super horrible. Anzen is badly sorted.

In order for the KNN operators to use a spatial index, one of the geometries needs to remain a constant in the query. This makes it difficult to use where you interested in more than one location.

How to trick KNN to work with non-constants

Use Case: I want to find the 2 closest transportation stops to my set of locations.

LATERAL and KNN

Remember KNN one geometry has to be constant for index to be used.
What if for each record we need more than one answer?

Use LATERAL

What are the 2 Closest public transportation to each plade

LATERAL + KNN

```
SELECT p.name As place, pubt.name As transport
FROM
 planet_osm_point As p
   CROSS JOIN
     LATERAL(
   SELECT name
     FROM planet_osm_point As t
        WHERE tags ? 'public_transport'
        ORDER BY p.way <-> t.way ASC LIMIT 2) As pubt
     WHERE tags @>
   'amenity=>restaurant, cuisine=>japanese'::hstore;
```

Segmentize a Linestring in Geography

New PostGIS 2.1 ST_Segmentize(geography) can create great circles

Segmentize in geography output as geometry wkt

```
SELECT ST_AsText(
 ST_Segmentize('LINESTRING(-118.4079 33.9434, 2.5559 49.0083)'::geography,
  10000) );

LINESTRING(-118.4079 33.9434,-118.365191634689 33.9946750650617,
 -118.322351004015 34.0460320153076,
 ...,2.48756947085441 49.0516183725212,2.5559 49.0083)
```

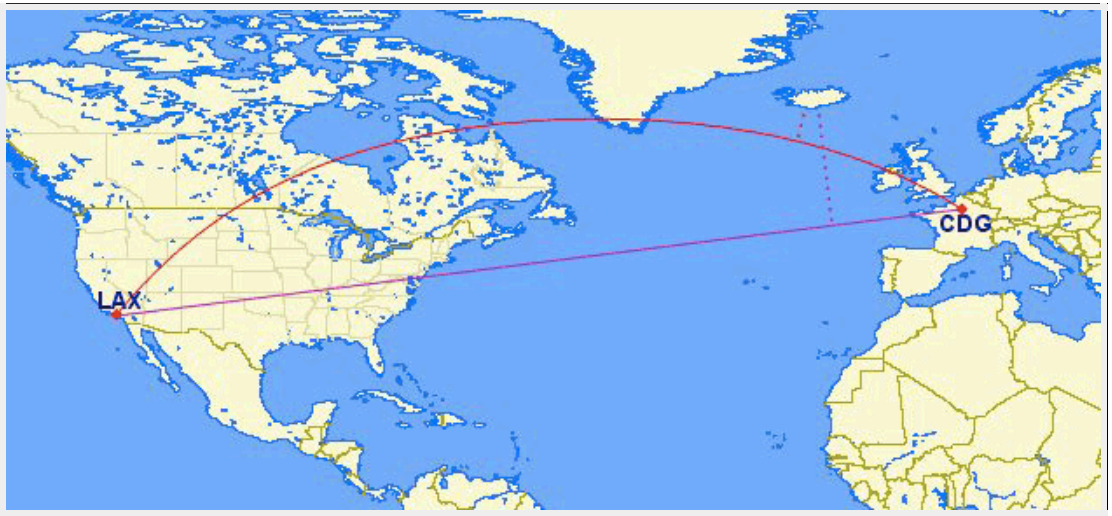
Segmentize and output as Google encoded line

PostGIS 2.2 we have ST_AsEncodedPolyline useful for drawing on google maps and use in Leaflet. ST_LineFromEncodedPolyline for getting back a geometry.

```
SELECT ST_AsEncodedPolyline(
  ST_Segmentize(
    'LINESTRING(-118.4079 33.9434, 2.5559 49.0083)'::geography,
       10000)::geometry,
   4);
```

```
gqdnEjpuqUo_I}iG}_IwjGo`IqkG_aImlGoaIgmG..~mGskLvmGajL
```

Geography segmentize vs. Geometry segmentize on a map



From BoundlessGeo docs

Lessons learned for geography and geometry

- hstore works well with PostGIS
- Use ST_DWithin (not ST_Distance) for indexable distance checking
- Geography doesn't support KNN
- compound gist indexes can replace two separate gist indexes and often performs better, but experiment
- Web mercator is not so good for distance checking but pretty good for KNN point distance
- Use geography for accurate measurement
- Limitations in each type can be compensated by the other to some extent
- LATERAL with KNN operators are a really good combination

PostGIS Raster specific lessons

Preamble: Loading the data

First we need raster data

http://www.oregon.gov/DAS/CIO/GEO/pages/alphalist.aspx

- http://www.oregon.gov/DAS/CIO/GEO/pages/alphalist.aspx
- DEMS: rawdems: 45122E6
- Aerial Imagery: aerial imagery multnomah county.
- wget ftp://159.121.106.159/imagery/DOQ_NAPP_2/45122/e/*

Load raster elevation data

```
export PGHOST=localhost
export PGPORT=5444
export PGPASSWORD=whatever
raster2pgsql -s 26710 -I -e -F -C -Y -t auto 5122E6DG portland_elev \
 | psql -U postgres -d presentation
```

Load aerial

```
gdal_translate ortho_1-1_1n_s_or051_2009_1.sid \
 -of JPEG -outsize 25% 25% portland.jpg

raster2pgsql -s 26710 -I -e -F -C -Y -t auto portland.jpg portland_aer \
 | psql -U postgres -d presentation
```

What raster tables we have?

```
SELECT r_table_name As tbl, r_raster_column As col, srid,
 scale_x As sx, scale_y As sy,
 blocksize_x, blocksize_y, pixel_types As pt
 FROM raster_columns
 where r_table_name = 'portland_elev';
```

```
     tbl      | col  | srid  | sx | sy  | blocksize_x | blocksize_y |   pt
--------------+------+-------+----+-----+-------------+-------------+---------
 portland_elev | rast | 26710 | 30 | -30 |          65 |          93 | {16BSI}
```

Carving out an area of interest

This is the first step you should do for any raster analysis involving multiple pixels.

Naive User: Carving out an area of interest

Unions tiles first and then clips, or doesn't clip at all.

```
SELECT ST_Clip(ST_Union(rast),loc)
FROM portland_aer INNER JOIN
 ST_Expand(ST_Transform(
    ST_SetSRID(
        ST_Point(-122.66226,45.53007),4326),2992),600) As loc
 ON ST_Intersects(rast, loc );
```

This is often orders of magnitude more work than experienced way of clipping first and then unioning because union has more pixels to deal with.

Experienced User: Carving out an area of interest

Clips first and then unions the clippings.

```
SELECT ST_Union(ST_Clip(rast,loc))
FROM portland_aer INNER JOIN
 ST_Expand(ST_Transform(
    ST_SetSRID(
        ST_Point(-122.66226,45.53007),4326),2992),600) As loc
 ON ST_Intersects(rast, loc );
```

Clippling is a much faster operation than unioning large areas.

Using ST_Resize for proportionate scaling

ST_Resize is finicky. If you need pixels, cast your values to integers otherwise you'll get some surprises, if you want to do by percentage, use float lower than 1 or text '50%'.

Inexperienced raster user tries to resize

```
SELECT ST_Resize(rast,300,
   (ST_Height(rast)*1.00/ST_Width(rast)*300) )
 FROM (SELECT ST_Union(ST_Clip(rast,loc)) As rast
FROM portland_aerc INNER JOIN
 ST_Expand(ST_Transform(
    ST_SetSRID(
        ST_Point(-122.663,45.53007),4326),26710),600*0.35) As loc
 ON ST_Intersects(rast, loc ) ) As f;
```

ERROR: Percentages must be a value greater than zero and less than or equal to one, e.g. 0.5 for 50%

Experienced raster user successfully resizes

Casts to integer first and gets a pretty picture.

```
SELECT ST_Resize(rast,300,
   (ST_Height(rast)*1.00/ST_Width(rast)*300)::int )
 FROM (SELECT ST_Union(ST_Clip(rast,loc)) As rast
FROM portland_aerc INNER JOIN
 ST_Expand(ST_Transform(
    ST_SetSRID(
        ST_Point(-122.663,45.53007),4326),26710),600*0.35) As loc
 ON ST_Intersects(rast, loc ) ) As f;
```

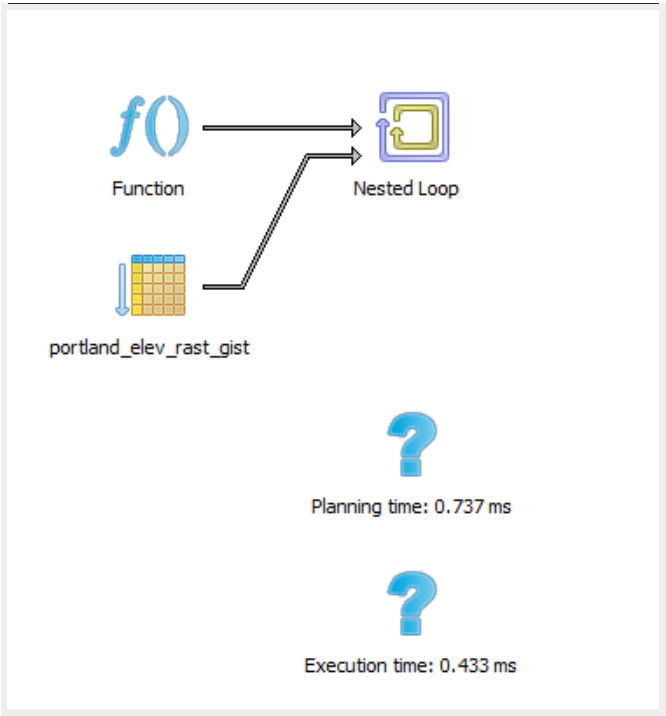
Transform your location not your data

What is the elevation at a point?

```
SELECT ST_Value(rast,1, loc)
FROM portland_elev
INNER JOIN ST_Transform(
ST_SetSRID(
 ST_Point(-122.66226,45.53007),4326),26710) As loc
ON ST_Intersects(rast,loc);
```

```
33
```

Verify index used



Raster lessons learned

- Reproject your area of interest not your raster.
- Clip first before doing other raster operations like ST_Union, ST_SummaryStats, and ST_Resize
- Do an explain plan to verify indexes are used

FIN

Thank you. Buy our books! http://www.postgis.us