POSTGIS SPATIAL TRICKS

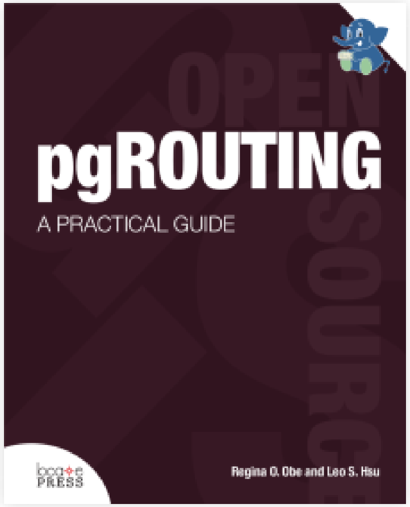
REGINA OBE

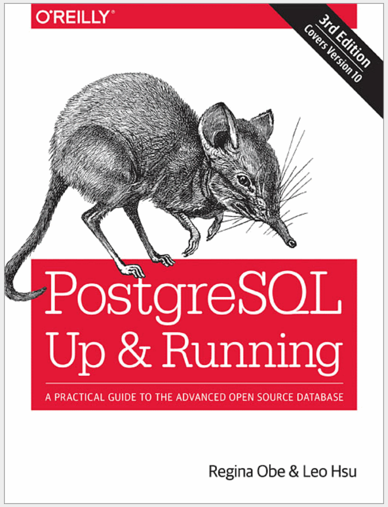
http://www.paragoncorporation.com

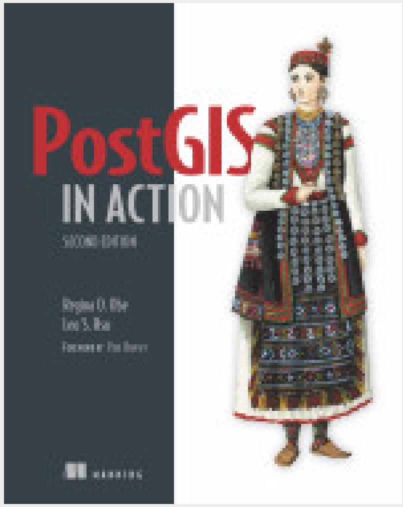
Buy our books! at http://postgis.us/page_buy_book

OUR LATEST BOOK

pgRouting: A Practical Guide http://locatepress.com/pgrouting







NEW AND ENHANCED VECTOR FUNCTIONS COMING IN POSTGIS 2.4

Note: PostGIS 2.4.0 alpha recently released.

- Mapbox Vector Tile output functions (Björn Harrtell / Carto). Requires compile with proto-buf. (He talked about this earlier today catch the afer video if you missed it.)
- ST_FrechetDistance (Shinichi Sugiyama). Requires compile with GEOS 3.7.0 (currently in development).
- ST_Centroid for geography, centroid based on round earth (Danny Götte)
- ST_CurveToLine enhancments, addition of max error argument.

PARALLELIZATION OF SPATIAL JOINS AND FUNCTIONS

Requires PostgreSQL 9.6+ and PostGIS 2.3+. Read more: http://blog.cleverelephant.ca/2016/03/parallel-postgis-joins.html

PostgreSQL 10+ allows for more kinds of workloads to take advantage of parallelism - now there is addition of parallel bitmap heap scan and parallel index scan. In 2.4.0 most aggregates, window functions, immutable, and stable functions (include both vector and raster) are marked parallel safe.

```
ALTER SYSTEM set max_worker_processes=4;
ALTER SYSTEM set max_parallel_workers=4; -- new in PG 10
set parallel_tuple_cost=0.01;
set max_parallel_workers_per_gather=4;
```

LOADING DATA

PostgreSQL + PostGIS makes it really easy to load data including spatial data. Lots of options.

- shp2pgsql, shp2pgsql-gui for loading ESRI shapefiles. Packaged with PostGIS client tools.
- raster2pgsql - for loading lots of kinds of raster data into PostGIS. Under the covers uses GDAL api http://gdal.org under the scenes. Packaged with PostGIS client tools.
- oracle_fdw https://github.com/laurenz/oracle_fdw - PostgreSQL foreign data wrapper for connecting to Oracle databases. Will expose Oracle SDO_Geometry as PostGIS geometry.
- GDAL / OGR http://gdal.org ogr2ogr is a popular command-line tool used for loading data from one vector source to another (including PostGIS), popular companion of PostGIS.
- ogr_fdw https://github.com/pramsey/pgsql-ogr-fdw- PostgreSQL foreign data wrapper can query and use to load lots of types of vector data and also non-spatial data.
- imposm, osm2pgsql, and osm2pgrouting are command line tools specifically designed for loading data from OpenStreetMap into PostGIS.

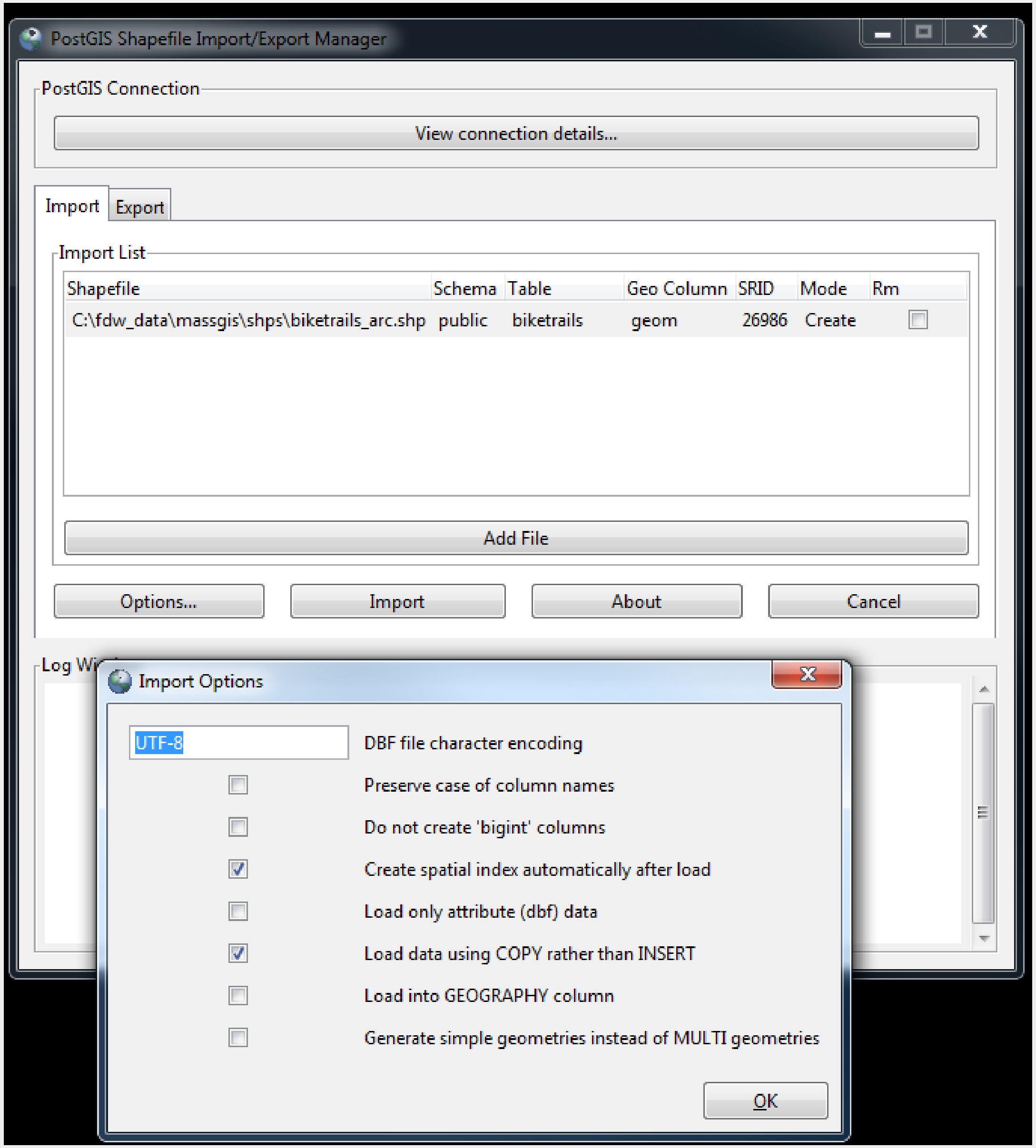
SHP2PGSQL

Converts ESRI Shapefile to SQL statements you can then load with psql

```
export PGDATABASE=foss4g2017
export PGUSER=postgres
export PGHOST=localhost
export PGPASSWORD=whatever
export PGPORT=5432
shp2pgsql -s 26986 -D biketrails_arc biketrails | psql
```

Windows users use SET instead of export for setting variables

SHP2PGSQL-GUI: IMPORTING



Shapefile	Schema	Table	Geo Column	SRID	Mode	Rm
C:\fdw_data\massgis\shps\biketrails_arc.shp	public	biketrails	geom	26986	Create	□

UTF-8 DBF file character encoding

□ Preserve case of column names

□ Do not create 'bigint' columns

☑ Create spatial index automatically after load

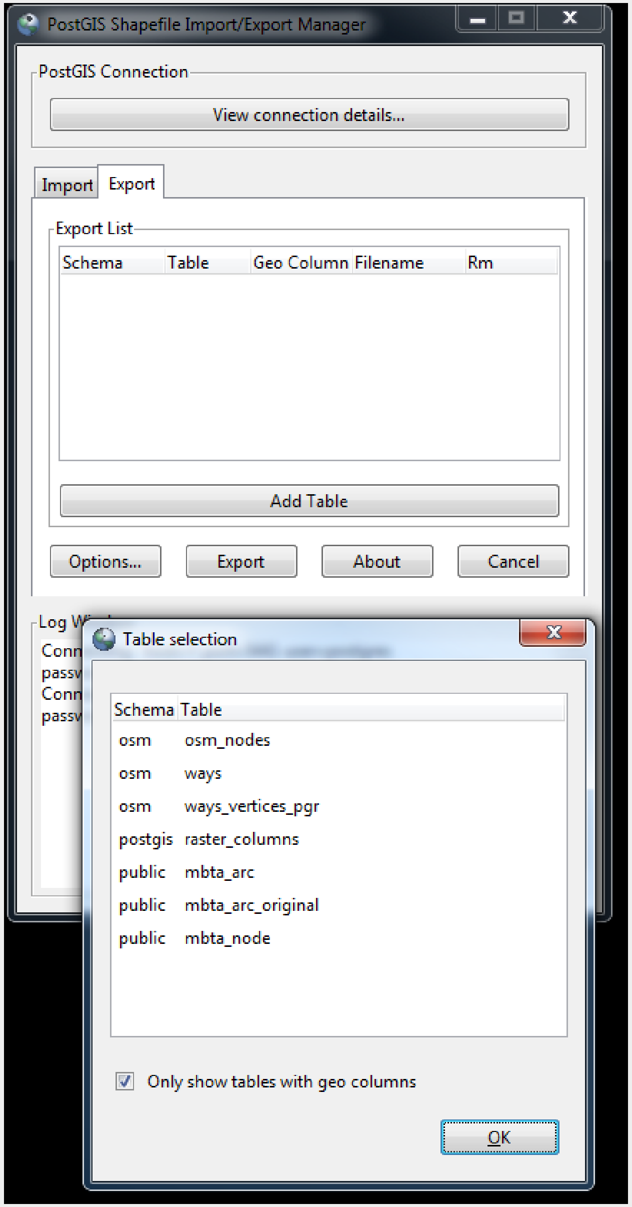
□ Load only attribute (dbf) data

☑ Load data using COPY rather than INSERT

□ Load into GEOGRAPHY column

□ Generate simple geometries instead of MULTI geometries

SHP2PGSQL-GUI: EXPORTING



Schema	Table
osm	osm_nodes
osm	ways
osm	ways_vertices_pgr
postgis	raster_columns
public	mbta_arc
public	mbta_arc_original
public	mbta_node

☑ Only show tables with geo columns

POSTGRESQL + GDAL (OGR) ~ POSTGIS = OGR_FDW POSTGRESQL FOREIGN DATA WRAPPER

Doesn't require PostGIS to use, but will expose spatial columns as PostGIS geometry if PostGIS is installed.

Many thanks to Paul Ramsey and Even Rouault.

The PostgreSQL/OGR/PostGIS bump:
(as Holly Orr says, it's like getting a hug from an ogre)

DATA WRANGLING WITH OGR_FDW

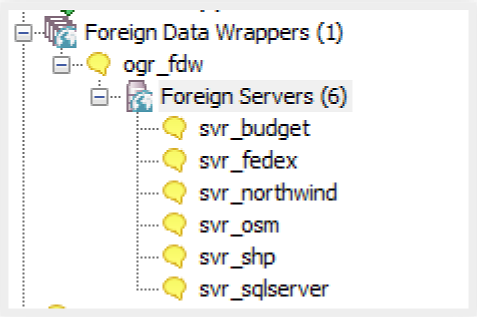
If you have all sorts of data of both a spatial and non-spatial flavor to tame, make sure you have ogr_fdw foreign data wrapper in your tool belt.

- For windows users, it's part of PostGIS bundle (versions 2.2 and up) on application stackbuilder.
- For windows/linux/mac desktop users, it's part of the BigSQL PostGIS package. Read BigSQL ogr_fdw http://bit.ly/2uZw2Ue
- For CentOS/Red Hat/Scientific etc, it's available via yum.postgresql.org
- For others, if you have PostGIS with GDAL support, just need postgresql dev package to compile. Download the source https://github.com/pramsey/pgsql-ogr-fdw

WHY IS OGR_FDW SO SEXY?

You have the combined power of GDAL, PostgreSQL, and any PostgreSQL extension you want (including PostGIS) working seamlessly together. So many kinds of data you can query and take advantage of PostgreSQL functions and any extension functions and types such as PostGIS, hstore, built-in json to tame your data.

- Spreadsheets
- ODBC datasources
- OSM files (OSM, PBF)
- ESRI Shapefiles
- Spatial web services
- Many more


Foreign Data Wrappers (1)
ogr_fdw
Foreign Servers (6)
svr_budget
svr_fedex
svr_northwind
svr_osm
svr_shp
svr_sqlserver

ENABLE IT IN YOUR DATABASE

```
CREATE EXTENSION ogr_fdw;
```

LINK IN A WHOLE FOLDER OF ESRI SHAPEFILES AND DBASE FILES

```
CREATE SERVER svr_shp FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (datasource 'C:/fdw_data/massgis/shps',
 format 'ESRI Shapefile'
);
CREATE SCHEMA shps;
-- this is a PostgreSQL 9.5 feature
IMPORT FOREIGN SCHEMA ogr_all
FROM SERVER svr_shp INTO shps;
```

```
\dE shps.*
```

```
                    List of relations
 Schema |         Name          |      Type      |  Owner
--------+-----------------------+----------------+----------
 shps   | biketrails_arc        | foreign table | postgres
 shps   | towns_arc             | foreign table | postgres
 shps   | towns_poly            | foreign table | postgres
 shps   | towns_poly_areacode   | foreign table | postgres
 shps   | towns_polym           | foreign table | postgres
 shps   | towns_pop             | foreign table | postgres
 shps   | zipcodes_nt_poly      | foreign table | postgres
(7 rows)
```

QUERY YOUR GEOMETRY_COLUMNS CATALOG

Sadly it often guesses wrong on the srid, these are NAD 83 state plane MA (26986), not NAD 83 long/lat (4269). Also note that towns_polym is a mix of polygons and multipolygons, but got registered as polygon.

```sql
SELECT f_table_name As tbl, f_geometry_column As geom, srid, type
FROM geometry_columns
WHERE f_table_schema = 'shps'
ORDER BY tbl;
```

```
      tbl         | geom | srid  |    type
------------------+------+-------+------------
 biketrails_arc   | geom |  4269 | LINESTRING
 towns_arc        | geom |  4269 | LINESTRING
 towns_poly       | geom |  4269 | POLYGON
 towns_polym      | geom |  4269 | POLYGON
 zipcodes_nt_poly | geom |  4269 | POLYGON
(5 rows)
```

BUT WE CAN FIX THAT :)

```
ALTER FOREIGN TABLE shps.towns_polym
 ALTER COLUMN geom type geometry(geometry,26986);
 -- and it believes us

SELECT ST_SRID(geom), ST_GeometryType(geom)
FROM shps.towns_polym limit 1;
```

```
 st_srid | st_geometrytype
---------+-----------------
   26986 | ST_Polygon
(1 row)
```

```
SELECT f_table_name As tbl, f_geometry_column As geom, srid, type
FROM geometry_columns
WHERE f_table_schema = 'shps' AND tbl='towns_polym'
ORDER BY tbl;
```

```
       tbl        | geom | srid    |    type
------------------+------+---------+------------
 towns_polym      | geom |  26986  | POLYGON
 (1 row)
```

If this was a real table, we'd have to do:

```
ALTER TABLE ...
    ALTER COLUMN geom type geometry(geometry,26986) USING ST_SetSRID(geom,26986);
```

YOU CAN FIX BAD GEOMETRIES RIGHT IN SHAPE FILE WITH POWER OF POSTGIS

Requires ogr_fdw 1.0.1+. Make sure the user that postgres runs under has edit/delete rights to the folder holding the shape files.

```
UPDATE shps.towns_polym
    SET geom = ST_MakeValid(geom)
WHERE NOT ST_IsValid(geom)
RETURNING town;
```

```
NOTICE:  Ring Self-intersection at or near point 241494.43330000341 890709.87110000104
NOTICE:  Ring Self-intersection at or near point 306590.87370000035 822452.56080000103
NOTICE:  Ring Self-intersection at or near point 273304.93349999934 802752.31069999933

Total query runtime: 320 msec
town
-----
QUINCY
YARMOUTH
TISBURY
```

OSM FILES

```
-- data from https://mapzen.com/data/metro-extracts/
CREATE SERVER svr_osm
   FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (datasource 'C:/fdw_data/boston_massachusetts.osm.pbf',format 'OSM');
  CREATE SCHEMA IF NOT EXISTS osm;
IMPORT FOREIGN SCHEMA ogr_all
FROM SERVER svr_osm INTO osm;

\dE osm.*
```

```
                 List of relations
 Schema |       Name        |     Type      |  Owner
--------+-------------------+---------------+----------
 osm    | lines             | foreign table | postgres
 osm    | multilinestrings  | foreign table | postgres
 osm    | multipolygons     | foreign table | postgres
 osm    | other_relations   | foreign table | postgres
 osm    | points            | foreign table | postgres
(5 rows)
```

```
-- requires CREATE EXTENSION hstore; to cast other_tags to hstore
-- and hstore extension has function hstore_to_jsonb that will cast hstore to jso
-- but we use that to convert to jsonb
-- 22048 rows
CREATE TABLE boston_pois AS
SELECT osm_id, name, geom::geography As geog, is_in,
    place, hstore_to_jsonb(other_tags::hstore) AS other_tags
FROM osm.points;

-- 35946 rows
CREATE TABLE boston_roads AS
SELECT osm_id, name, geom::geography As geog,
    hstore_to_jsonb(other_tags::hstore) AS other_tags
FROM osm.lines
WHERE highway > '';

-- 26986 is srid for Massachusetts state plane meters. 4326 is wgs 84 long lat
-- 267491 rows affected, 14.6 secs execution time.
CREATE TABLE boston_polys AS
SELECT osm_id, name, geom::geography As geog, ST_Transform(geom,26986) As  geom,
    hstore_to_jsonb(other_tags::hstore) AS other_tags, building
FROM osm.multipolygons;
```

EVEN SPREADSHEETS

Each workbook is considered a server and each sheet a table

```
CREATE SERVER svr_fedex FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (datasource 'C:/fdw_data/Fedex2016.xls',
 format 'XLS'
);
-- link only 1 spreadsheet preserve headers
IMPORT FOREIGN SCHEMA ogr_all  LIMIT TO (Fedex_Rates_IP)
        FROM SERVER svr_fedex INTO public OPTIONS (launder_column_names 'false');

SELECT * FROM fedex_rates_ip;
```

Before

```
 fid |     Type     | Weight | Zone A  | Zone B  | Zone C  | Zone D  | Zone E  | Zone F  | Z
-----+--------------+--------+---------+---------+---------+---------+---------+---------+--
   2 | IntlPriority |      0 |   40.25 |    41.5 |      43 |   54.75 |   116.5 |      52 |
   3 | IntlPriority |     -1 |   66.25 |   67.75 |   62.25 |   74.25 |     132 |      68 |
   4 | IntlPriority |     -2 |   70.25 |    73.5 |   65.75 |   77.25 |  156.25 |      73 |
```

```
-- unpivot a subset of columns and keep others (requires CREATE EXTENSION hstore;
WITH fkv AS (
SELECT f."Type" As type, f."Weight" As weight,
        each(hstore(f) - '{fid,Type,Weight}'::text[]) AS kv
from fedex_rates_ip AS f)
SELECT type, weight, (kv).key AS zone, (kv).value::numeric As price
FROM fkv;
```

After

```
     type     | weight |       zone        |  price
--------------+--------+-------------------+---------
 IntlPriority |      0 | Zone A            |   40.25
 IntlPriority |      0 | Zone B            |    41.5
 IntlPriority |      0 | Zone C            |      43
 IntlPriority |      0 | Zone D            |   54.75
:
```

EVEN CSV FILES

You can point at a single CSV file or a whole folder of CSV files. Each file is considered a table.

Folder of CSV files

```
CREATE SERVER svr_census FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (datasource 'C:/fdw_data/census',
 format 'CSV'
);

IMPORT FOREIGN SCHEMA ogr_all
FROM SERVER svr_census INTO public;
```

Single file

```
CREATE SERVER svr_census_income FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (datasource 'C:/fdw_data/census/income.csv',
 format 'CSV'
);

IMPORT FOREIGN SCHEMA ogr_all
FROM SERVER svr_census_income INTO public;
```

EVEN OTHER RELATIONAL DATABASES

Format for SQL Server 'ODBC:your_user/your_password@yourDSN,table1,table2'. ODBC can be slow with a lot of tables (more than 150) so filter list if you have over 200 tables

```
CREATE SERVER svr_sqlserver FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (datasource 'ODBC:pguser/whatever@MSSQLTest,dbo.IssueLog,dbo.IssueNotes',
 format 'ODBC'
);
CREATE SCHEMA IF NOT EXISTS ss;
IMPORT FOREIGN SCHEMA "dbo."
        FROM SERVER svr_sqlserver INTO ss;
```

```
\dE ss.*
```

```
                 List of relations
 Schema |      Name        |     Type      |  Owner
--------+-----------------+---------------+----------
 ss     | dbo_issuelog    | foreign table | postgres
 ss     | dbo_issuenotes  | foreign table | postgres
(2 rows)
```

MAKE SURE HAVE INDEXES IN PLACE

2D just regular spatial index

```sql
CREATE INDEX idx_boston_pois_geog_gist ON boston_pois USING gist(geog);
CREATE INDEX idx_boston_polys_geom_gist ON boston_polys USING gist(geom);
```

Don't forget about index on jsonb fields:

```sql
CREATE INDEX idx_boston_pois_other_tags_gin ON boston_pois USING gin(other_tags);
```

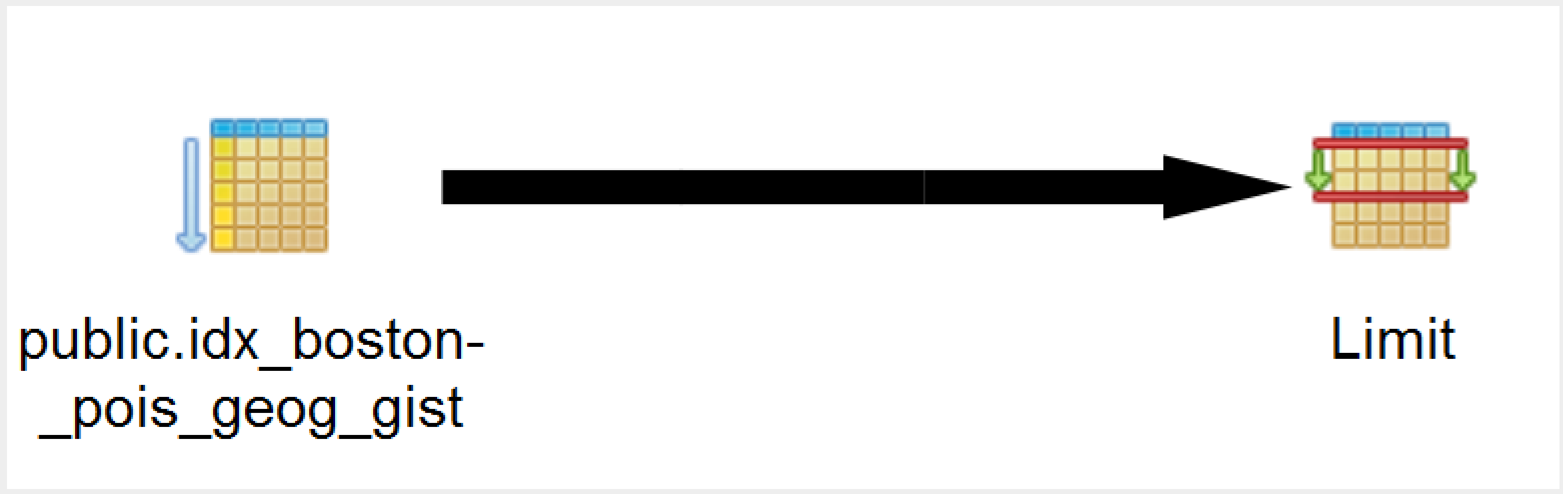
FIND N-CLOSEST PLACES (KNN)

Given a location, find the N-Closest places. Geography and n-D geometry operator support new in PostGIS 2.2. true distance check requires PostgreSQL 9.5+.

EXAMPLE: 5 CLOSEST POIS

```
-- 19ms
SELECT name,
    ST_Point(-71.04054,42.35141)::geography <-> geog As dist_m
FROM boston_pois As pois
WHERE name > ''
ORDER BY dist_m
LIMIT 5;
```

```
         name          |      dist_m
-----------------------+------------------
 World Trade Center    | 43.0799617300232
 Boston Ferry Terminal | 81.3545227312358
 Commonwealth Pier     | 141.785852676189
 7-Eleven              |  151.49969392488
 Dunkin' Donuts        | 157.350992916785
(5 rows)
```

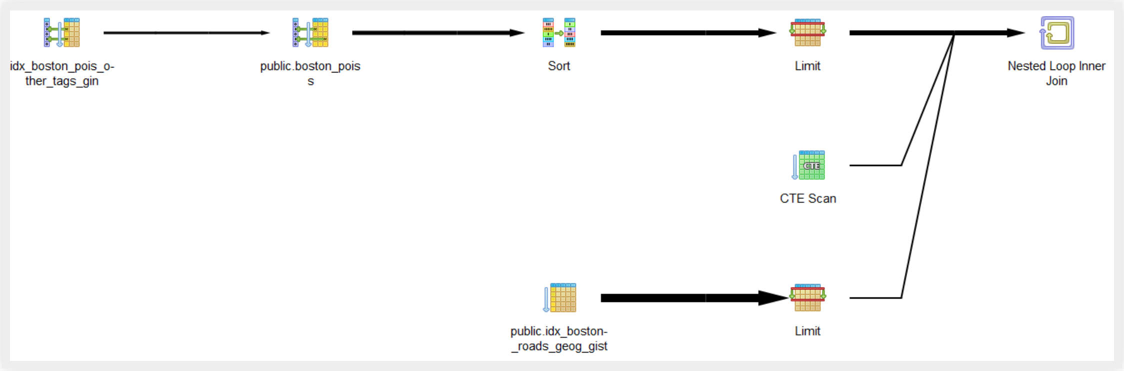


FIND 2 CLOSEST ROADS TO 4 CLOSEST FOOD SPOTS WITH CUISINE WITH LATERAL AND CTE

PostgreSQL 9.5+ and PostGIS 2.2+ for true distance.

```
-- CTE to find 4 closest spots with cuisine
WITH p AS (SELECT name, other_tags->>'cuisine' As type,
    ST_Point(-71.04054,42.35141)::geography <-> geog As dist_m, geog
FROM boston_pois As pois
WHERE other_tags ? 'cuisine'
ORDER BY dist_m LIMIT 4)
-- now for each spot find the two closest roads to each
SELECT p.name, p.type, r.name As road,
 r.dist_m_road::numeric(10,2), p.dist_m::numeric(10,2)
FROM p, LATERAL (SELECT rr.name, rr.geog <-> p.geog AS dist_m_road
FROM boston_roads AS rr WHERE rr.name > ''
ORDER BY dist_m_road LIMIT 2) As r;
```

```
        name         |               type                |        road        | dist_m_road | dist_m
---------------------+-----------------------------------+--------------------+-------------+--------
 Blue State Coffee   | vegan;international;vegetarian | Seaport Boulevard |        4.59 | 207.63
 Blue State Coffee   | vegan;international;vegetarian | Seaport Boulevard |       13.89 | 207.63
 Committee           | mediterranean;greek               | Northern Avenue    |       30.17 | 524.42
:
 Row 34              | oysters,fish                      | Congress Street    |       21.86 | 606.15
 sweetgreen          | salad                             | Stillings Street   |       11.37 | 608.47
 sweetgreen          | salad                             | Congress Street    |       22.05 | 608.47
(8 rows)
Time: 17.205 ms
```


idx_boston_pois_other_tags_gin
public.boston_pois
Sort
Limit
Nested Loop Inner Join
CTE Scan
public.idx_boston_roads_geog_gist
Limit

WHAT PLACES ARE WITHIN X-DISTANCE

Limit results set by distance rather than number of records. Like KNN, geometry can be anything like distance from a road, a lake, or a point of interest.

EXAMPLE: GEOGRAPHY WITHIN 1000 METERS OF LOCATION

What are closest fast food joints within 1500 meters. This will work for PostGIS 1.5+

```
SELECT name,    other_tags->>'cuisine' As cuisine,
        ST_Distance(pois.geog,ref.geog) As dist_m
FROM boston_pois AS pois,
        (SELECT  ST_Point(-71.04054,42.35141)::geography) As ref(geog)
  WHERE other_tags @> '{"amenity":"fast_food"}'::jsonb
        AND ST_DWithin(pois.geog, ref.geog, 1500)
ORDER BY dist_m;
```

```
         name           | cuisine  |    dist_m
------------------------+----------+---------------
 Dunkin' Donuts         | NULL     |  157.49061449
 Dunkin Donuts          | NULL     |  745.32469307
 Jimmy John's           | NULL     |  770.41451472
 McDonald's             | burger   | 1181.50916817
 Susan's Deli of Course | sandwich | 1308.09618596
 Dunkin' Donuts         | NULL     | 1308.56035564
 Subway                 | sandwich | 1320.97093007
 Al's South Street Cafe | sandwich | 1383.48220699
 Dunkin' Donuts         | NULL     | 1445.15739494
 Figaro's               | sandwich | 1457.90263811
(10 rows)

Time: 25.034 ms
```

CONTAINMENT

Commonly used for political districting and aggregating other pertinent facts. E.g. How many people gave to political campaigns in 2013 and what was the total per boro ordering by most money.

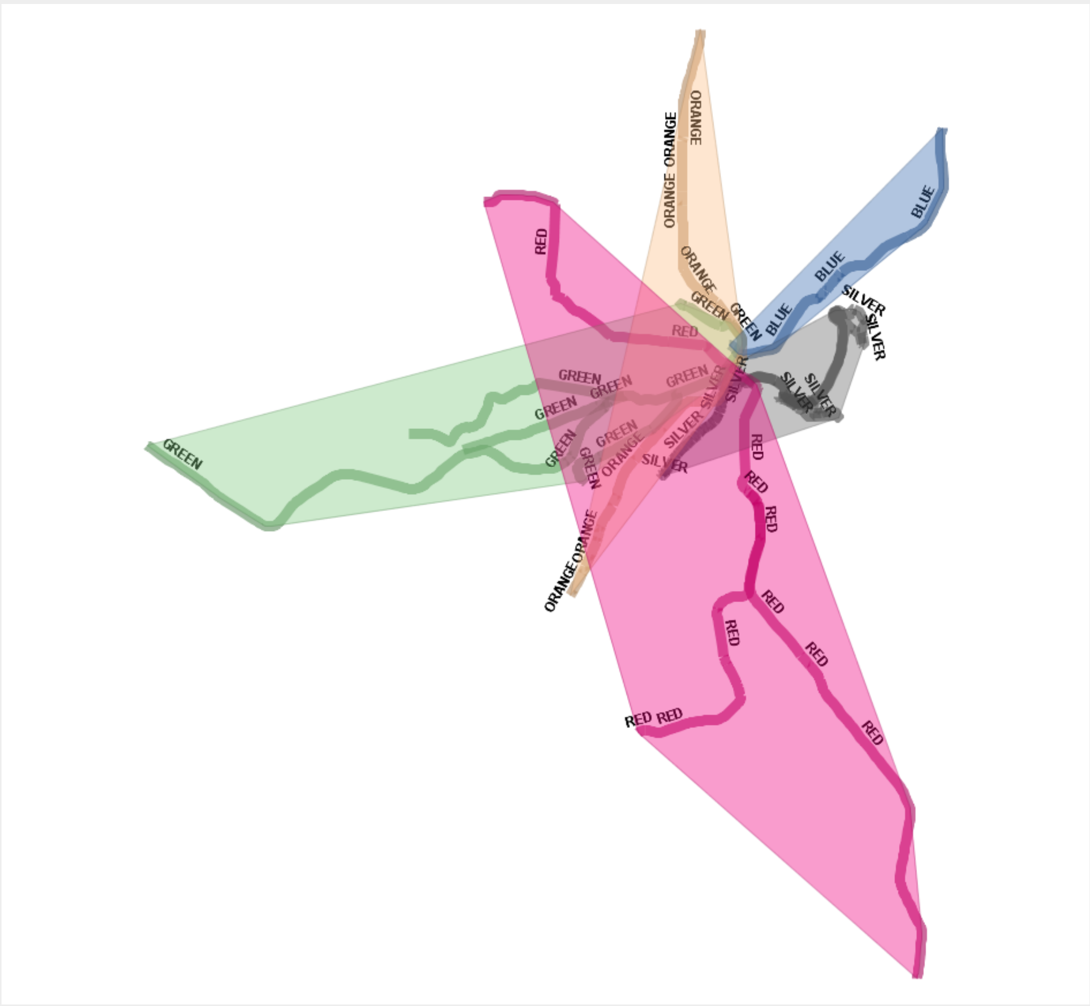
```
SELECT c.boro_name, COUNT(*) As num, SUM(amount) As total_contrib
FROM ny_campaign_contributions As m INNER JOIN nyc_boros As c ON ST_Covers(c.geom
GROUP BY c.boro_name
ORDER BY total_contrib DESC;
```

```
   boro_name    | num  | total_contrib
---------------+------+---------------
 Manhattan      | 4872 |    4313803.55
 Queens         | 3751 |    1262684.36
 Brooklyn       | 2578 |    1245226.04
 Staten Island |  813 |     248284.47
 Bronx          |  999 |     219805.02
(5 rows)
```

AGGREGATE THINGS GEOMETRICALLY

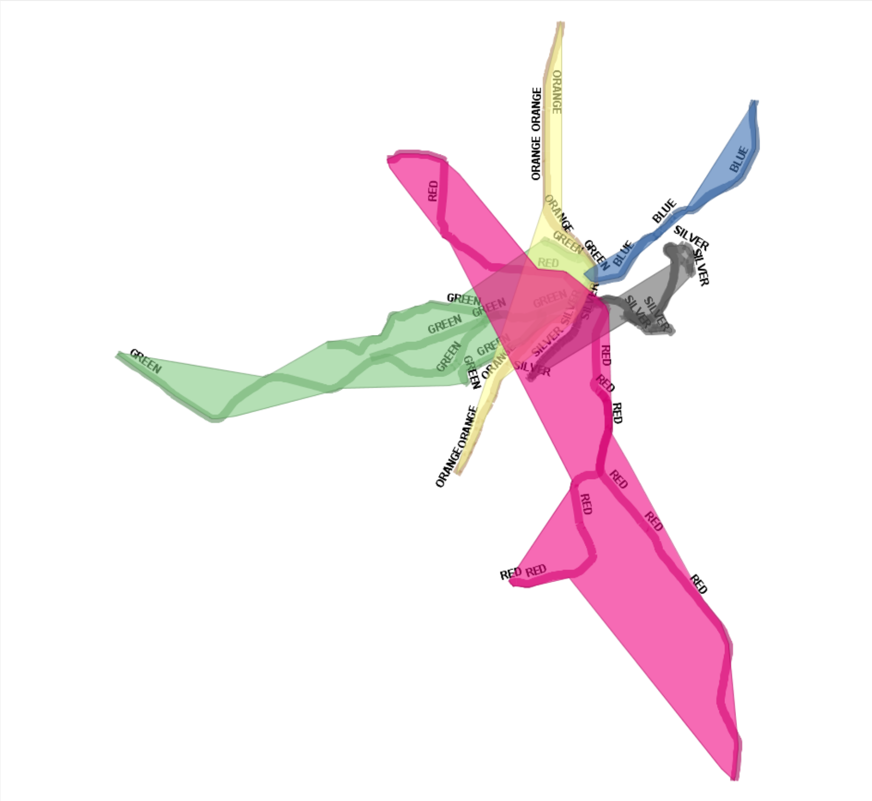
Create convex hull based on lines.

```
WITH s AS (
SELECT geom, line
FROM mbta_lines AS s)
SELECT line, ST_ConvexHull(ST_Union(geom)) As hull
FROM s
GROUP BY line;
```



Create concave hull based on station lines

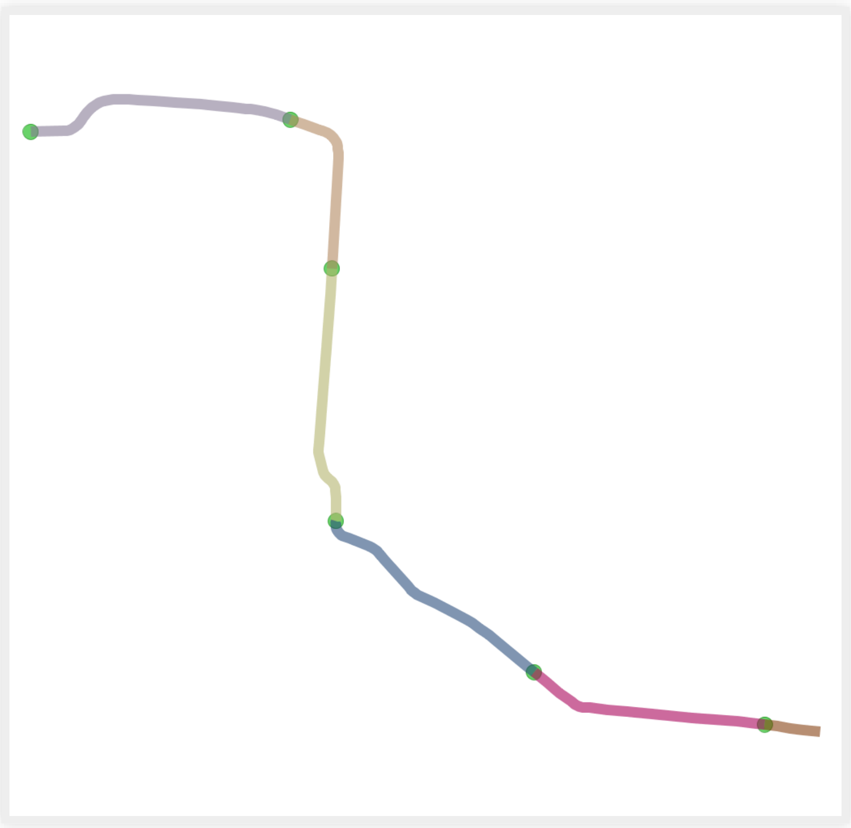
```
WITH s AS (
SELECT geom, line
FROM mbta_lines AS s)
-- last arg false means do not allow holes
SELECT line, ST_ConcaveHull(ST_Union(geom),0.8,false) As hull
FROM s
GROUP BY line;
```



BREAK LINESTRING AT POINTS OF INTEREST

Requires PostGIS 2.2+. PostgreSQL 9.4+ Snap, Split, and Dump

```
SELECT L.gid, D.ordinality As sub_id, D.geom::geometry(LINESTRING,26986) AS geom
FROM
   mbta_lines AS L
    LEFT JOIN LATERAL
    (   -- form a multipoint of all the nodes
       -- close enough to line to be considered on the line
        SELECT
            ST_Union(N.geom ORDER BY L.geom <-> N.geom) AS geom
        FROM mbta_stations AS N
        WHERE ST_DWithin(L.geom, N.geom, 10)
    ) AS MP ON TRUE
    CROSS JOIN LATERAL
-- snap the LINE to the MP which forces nodes to be injected to the line
-- then split at these node location and dump multilinestring into individual lin
    ST_Dump(
        COALESCE(  ST_Split(ST_Snap(L.geom, MP.geom, 10), MP.geom), L.geom)
        ) WITH ORDINALITY AS D;
```

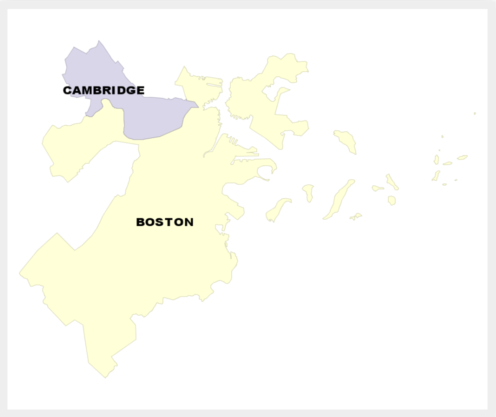


DIVIDE LARGE GEOMETRIES INTO SMALLER ONES WITH ST_SUBDIVIDE

New in PostGIS 2.2. Works for non-point geometries (only 2D). Second arg is max number of points to allow per divide.

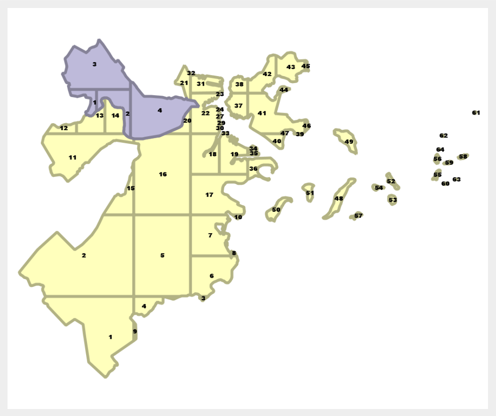
```
SELECT town,  f.ord, f.geom
FROM shps.towns polym, ST_SubDivide(geom, 100) WITH ordinality f(geom,ord)
WHERE town IN('BOSTON', 'CAMBRIDGE');
```

Before had 2 rows



```
   town     | st_npoints
------------+------------
 BOSTON     |       1893
 CAMBRIDGE  |        235
(2 rows)
```

After have 68 rows, no geometry has more than 100 points



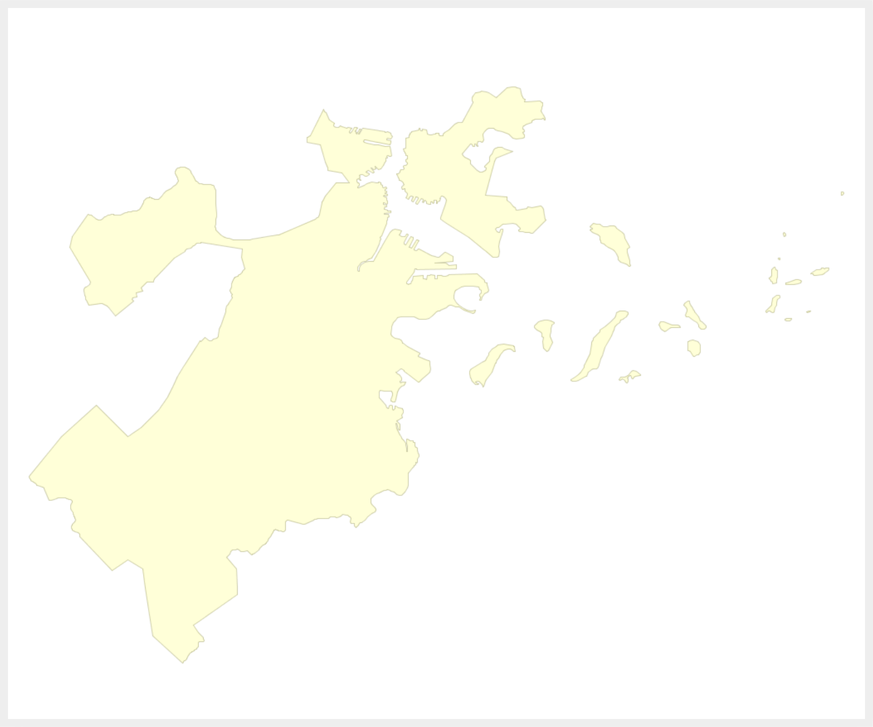
```
   town     | ord  | st_npoints
------------+------+------------
 BOSTON     |    1 |          89
 BOSTON     |    2 |          62
 :
 BOSTON     |   22 |          97
 :
 BOSTON     |   64 |           6
 CAMBRIDGE  |    1 |          40
 :
 CAMBRIDGE  |    4 |          63
(68 rows)
```

NEW IN POSTGIS 2.3
ST_GENERATEPOINTS

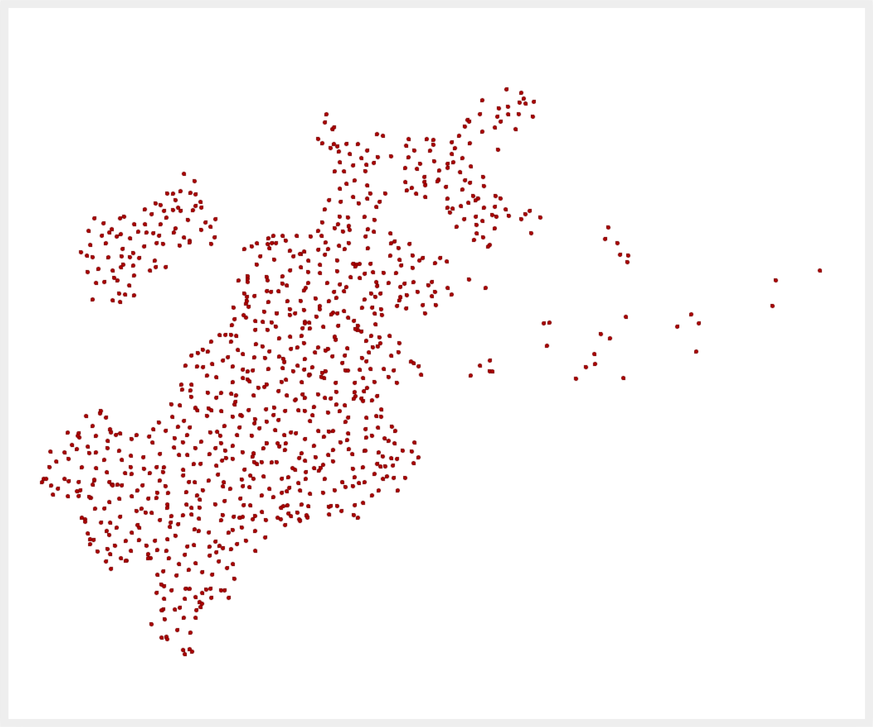
Converts multipolygon/polygon to multpoint - random space filling the area

```
SELECT town, ST_GeneratePoints(geom, 1000) AS geom
FROM shps.towns_polym
WHERE town =  'BOSTON';
```

Before: 1 Multi-Polygon



After: 1 multipoint of 1000 points



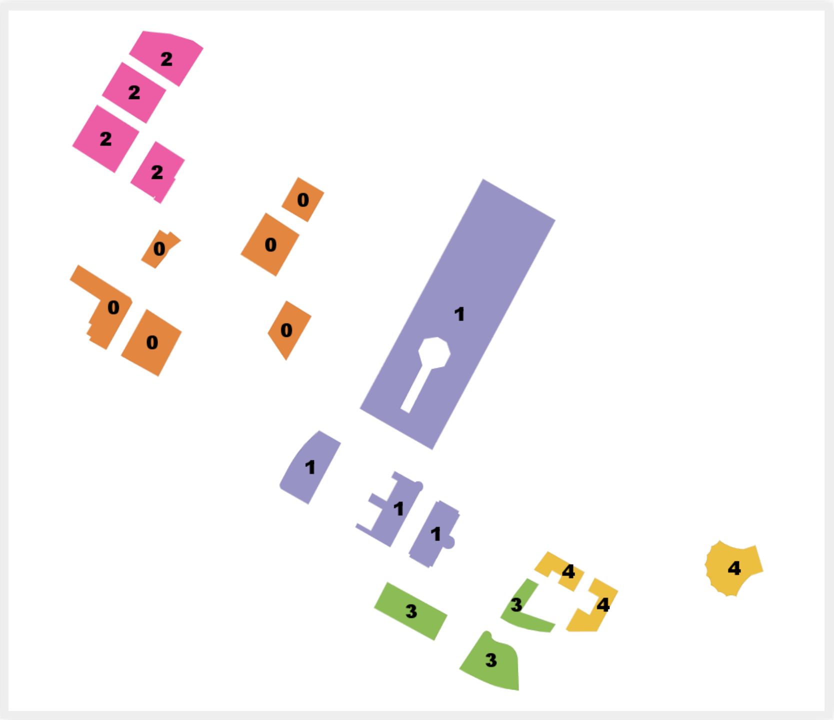
CLUSTERING GEOMETRIES USING WINDOW FUNCTIONS: NEW IN POSTGIS 2.3

- 2.3: ST_ClusterKMeans
- 2.3: ST_ClusterDbSCAN

ST_CLUSTERKMEANS: NUMBER BUILDINGS FROM 0-4 BY PROXIMITY TO EACH OTHER

```sql
SELECT name, ST_ClusterKMeans(geom, 5) OVER() AS bucket
    FROM boston_polys
WHERE name > '' AND building > ''
AND ST_DWithin(geom,
        ST_Transform(
            ST_GeomFromText('POINT(-71.04054 42.35141)', 4326), 26986),
           500)
ORDER BY bucket, name;
```

Need to add geom column to view



```
                   name                   | bucket
------------------------------------------+--------
 100 Northern Avenue                      |      0
 100 Pier 4                               |      0
 101 Seaport                              |      0
 District Hall                            |      0
 The Institute of Contemporary Art        |      0
 Watermark Seaport                        |      0
 Seaport Boston Hotel                     |      1
 Seaport Hotel & World Trade Center       |      1
 World Trade Center East                  |      1
 World Trade Center West                  |      1
 One Marina Park Drive                    |      2
 Twenty Two Liberty                       |      2
 Vertex                                   |      2
 Vertex                                   |      2
 Manulife Tower                           |      3
 Renaissance Boston Waterfront Hotel      |      3
 Waterside Place                          |      3
 Blue Hills Bank Pavilion                 |      4
 Park Lane Seaport I                      |      4
 Park Lane Seaport II                     |      4
(20 rows)


Time: 3.543 ms
```

ST_CLUSTERDBSCAN: SIMILAR TO KMEANS, BUT USES DESIRED MAX DISTANCE AND MINIMUM NUMBER ITEMS

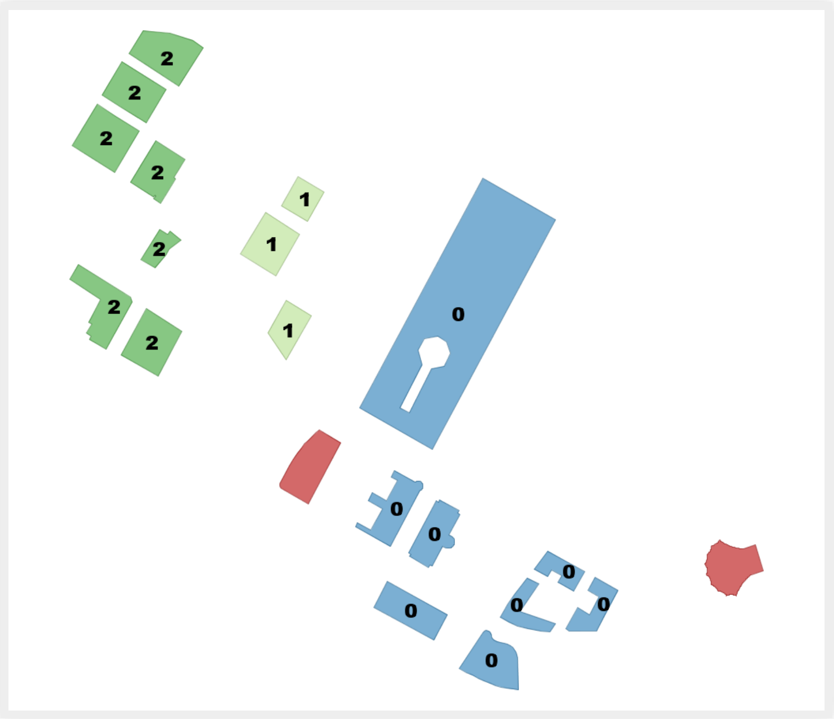
Cluster together buildings that are within 50 meters of each other and require cluster have at least 2 buildings. Note where requirement can't be satisfied you get **null** for bucket.

```sql
SELECT name, ST_ClusterDBSCAN(geom,50, 2) OVER() AS bucket
FROM boston_polys
WHERE name > '' AND building > ''
AND ST_DWithin(geom,
        ST_Transform(
            ST_GeomFromText('POINT(-71.04054 42.35141)', 4326), 26986),
           500)
ORDER BY bucket, name;
```

Need to add geom column to view

```
                 name                  | bucket
---------------------------------------+--------
 Manulife Tower                        |      0
 Park Lane Seaport I                   |      0
 Park Lane Seaport II                  |      0
 Renaissance Boston Waterfront Hotel   |      0
 Seaport Boston Hotel                  |      0
 Seaport Hotel & World Trade Center    |      0
 Waterside Place                       |      0
 World Trade Center East               |      0
 100 Northern Avenue                   |      1
 100 Pier 4                            |      1
 The Institute of Contemporary Art     |      1
 101 Seaport                           |      2
 District Hall                         |      2
 One Marina Park Drive                 |      2
 Twenty Two Liberty                    |      2
 Vertex                                |      2
 Vertex                                |      2
 Watermark Seaport                     |      2
 Blue Hills Bank Pavilion              |   NULL
 World Trade Center West               |   NULL
(20 rows)
```

```
Time: 3.812 ms
```



GDAL CONJOINS WITH POSTGIS = POSTGIS RASTER

- We already saw OGR_FDW Bump (the vector side of GDAL (aka OGR) bumping with PostgreSQL and sometimes PostGIS vector)
- Now the PostGIS Raster Bump (the raster side of GDAL bumping with PostGIS)

POSTGIS + GDAL = POSTGIS RASTER

A long time ago, a crazy man named Pierre Racine had a very crazy idea: https://trac.osgeo.org/postgis/wiki/WKTRaster and he got others Bborie Park, Sandro Santilli, Mateusz Loskot, Jorge Arévalo, David Zwarg and others to help implement his crazy scheme.

REGISTER YOUR RASTERS WITH THE DATABASE: OUT OF DB

You could with raster2pgsql the -R means just register, keep outside of database. Without the -R the data is stored in Db

```
raster2pgsql -I -C -R C:/data/nc_aerials/*.tif -F aerials | psql
```

OR (useful especially if you are on windows to force recursion of folders). Requires PostgreSQL 9.3+ PostGIS 2.1+

```
CREATE TABLE dir_list(file_name text);
COPY dir_list FROM PROGRAM 'dir C:\data\nc_aerials\*.tif /b /S'
        WITH (format 'csv');

CREATE TABLE aerials( rid serial PRIMARY KEY,rast raster, filename text);
INSERT INTO aerials(rast, filename)
SELECT
        ST_AddBand(
                NULL::raster,
                d.file_name, NULL::int[]
        ), d.file_name
FROM dir_list AS d;

SELECT AddRasterConstraints('aerials', 'rast');
--verify constraints
SELECT srid, scale_x, scale_y, blocksize_x As width,
blocksize_y As height, pixel_types, out_db
FROM raster_columns
WHERE r_table_name = 'aerials';
```

```
 srid | scale_x | scale_y | width | height |    pixel_types     | out_db
------+---------+---------+-------+--------+--------------------+---------
 2264 |     0.5 |    -0.5 | 10000 |  10000 | {8BUI,8BUI,8BUI} | {t,t,t}
(1 row)
```

```
CREATE INDEX idx_aerials_rast ON aerials USING gist(ST_ConvexHull(rast));
analyze aerials;
analyze aerials;
```

LET'S TILE THE RASTER TO 200X200 CHUNKS RIGHT IN DB

Requires PostGIS 2.1+. ST_Tile, if working on out-db keeps out-db and very fast.

```
CREATE TABLE aerials_200_200(rid serial primary key, rast raster, filename text);
INSERT INTO aerials_200_200(rast,filename)
SELECT ST_Tile(rast,200,200) As rast, filename
FROM aerials;
SELECT AddRasterConstraints('aerials_200_200', 'rast');
--verify constraints
SELECT srid, scale_x, scale_y, blocksize_x As width,
 blocksize_y As height, pixel_types, out_db
FROM raster_columns
WHERE r_table_name = 'aerials_200_200';
```

```
 srid | scale_x | scale_y | width | height |    pixel_types     | out_db
------+---------+---------+-------+--------+------------------+---------
 2264 |     0.5 |    -0.5 |   200 |    200 | {8BUI,8BUI,8BUI} | {t,t,t}
(1 row)
```

```
CREATE INDEX idx_aerials_200_200_rast ON aerials_200_200 USING gist(ST_ConvexHull
analyze aerials_200_200;
```

CREATE OVERVIEWS RIGHT IN DB

Requires PostGIS 2.2+. This will make in-db raster from out-db so might take a while. Took 8 minutes for my aerials table that had 30 10000x10000 raster links.

```
SELECT ST_CreateOverview('aerials'::regclass, 'rast', 4);
```

```
st_createoverview
------------------
o_4_aerials
```

```
CREATE INDEX idx_o_4_aerials_rast ON o_4_aerials USING gist(ST_ConvexHull(rast));
```

```
SELECT srid, scale_x, scale_y, blocksize_x As width,
 blocksize_y As height, pixel_types, out_db
FROM raster_columns
WHERE r_table_name = 'o_4_aerials';
```

```
 srid | scale_x | scale_y | width | height |   pixel_types    | out_db
------+---------+---------+-------+--------+------------------+---------
 2264 |       2 |      -2 | 10000 |  10000 | {8BUI,8BUI,8BUI} | {f,f,f}
(1 row)
```

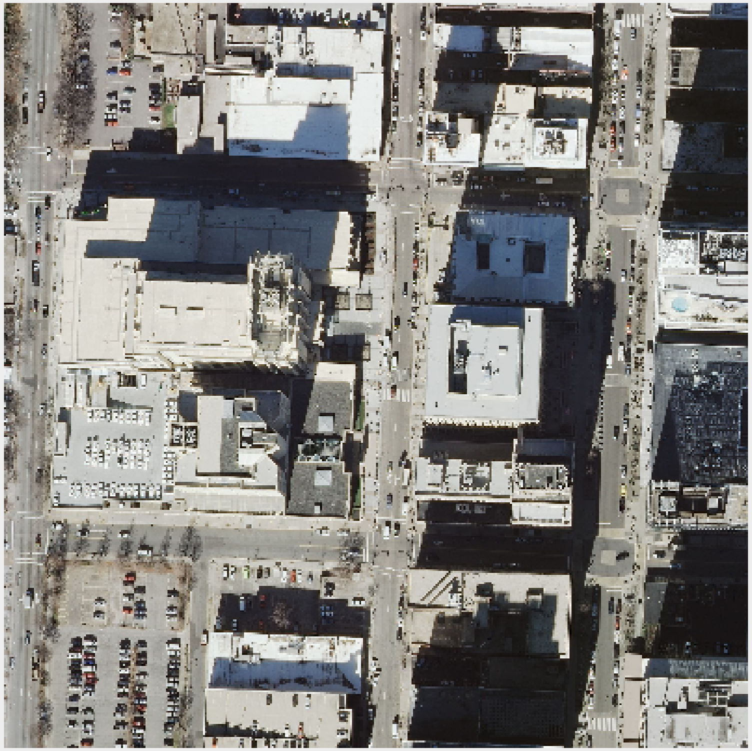
RETURN AN AREA: 500 FEET AROUND US

Project to same spatial ref as raster (2264 NC State Plane ft)

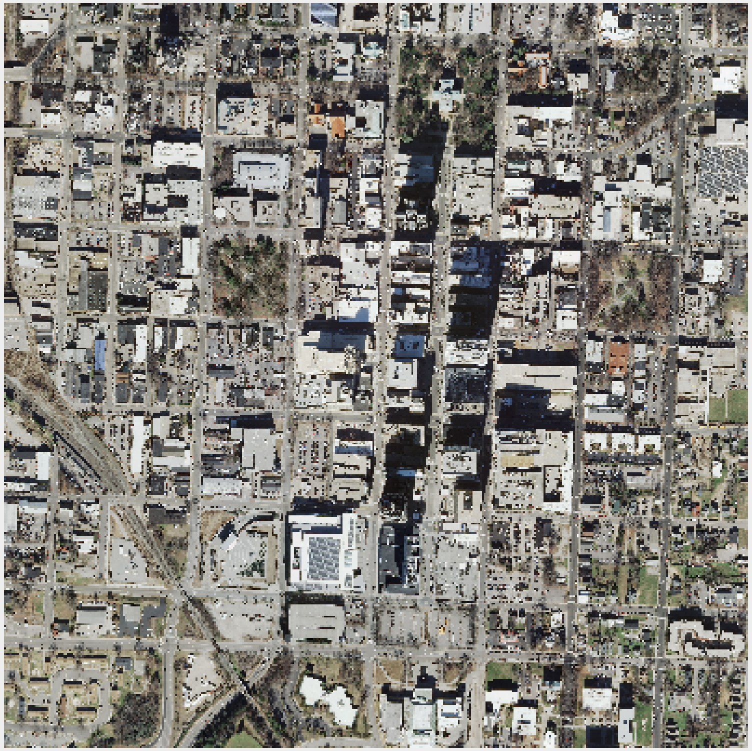
```
SELECT ST_AsPNG(ST_Resize(ST_Union(ST_Clip(rast, geom)), 0.20,0.20)), count(*)
FROM aerials_200_200 AS a,
        ST_Expand(
                ST_Transform(ST_SetSRID(ST_Point(-78.6404,35.77627),4326),
                        2264),500) As geom
WHERE ST_Intersects(a.rast,geom);
```

Using aerials: 4 secs (1 row), aerials_200_200: 5.9 sec (120 rows)

o_4_aerials resize 0.5 (980ms 1 row)



Using o_4_aerials resize 0.2, 2000 ft - 5.7 secs



ADDRESS STANDARDIZATION / GEOCODING / REVERSE GEOCODING

PostGIS 2.2 comes with extension address_standardizer. Also included since PostGIS 2.0 is postgis_tiger_geocoder (only useful for US).

In works improved address standardizer and worldly useful geocoder - refer to: https://github.com/woodbri/address-standardizer

ADDRESS STANDARDIZATION

Need to install address_standardizer, address_standardizer_data_us extensions (both packaged with PostGIS 2.2+). Using json to better show non-empty fields

```
SELECT *
FROM json_each_text(to_json(standardize_address('us_lex', 'us_gaz','us_rules'
, 'One Seaport Lane',
  'Boston, Massachusetts 02210' )))
WHERE value > '';
```

```
   key     |     value
-----------+---------------
 house_num | 1
 name      | SEAPORT
 suftype   | LANE
 city      | BOSTON
 state     | MASSACHUSETTS
 postcode  | 02210
(6 rows)
```

Same exercise using the packaged postgis_tiger_geocoder tables that standardize to abbreviated instead of full name

```
SELECT *
FROM json_each_text( to_json(
        standardize_address('tiger.pagc_lex', 'tiger.pagc_gaz','tiger.pagc_rules'
, 'One Seaport Lane',
  'Boston, Massachusetts 02210' )))
WHERE value > '';
```

```
    key     |  value
-----------+---------
 house_num | 1
 name      | SEAPORT
 suftype   | LN
 city      | BOSTON
 state     | MA
 postcode  | 02210
(6 rows)
```

GEOCODING USING POSTGIS TIGER GEOCODER

Given a textual location, ascribe a longitude/latitude. Uses postgis_tiger_geocoder extension requires loading of US Census Tiger data.

```sql
SELECT pprint_addy(addy) As address,
    ST_X(geomout) AS lon, ST_Y(geomout) As lat, rating
    FROM geocode('1 Seaport Lane, Boston, MA 02210',1);
```

```
            address             |        lon        |       lat        | rating
--------------------------------+-------------------+------------------+--------
 1 Seaport Ln, Boston, MA 02210 | -71.0411493412951 | 42.3497520198983 |      0
(1 row)
```

REVERSE GEOCODING

Given a longitude/latitude or GeoHash, give a textual description of where that is. Using postgis_tiger_geocoder reverse_geocode function

```
SELECT pprint_addy(addrs) AS padd,
  array_to_string(r.street,',') AS cross_streets
FROM reverse_geocode(ST_Point(-71.04115,42.34975)) AS r
    , unnest(r.addy) As addrs;
```

```
              padd              | cross_streets
--------------------------------+---------------
 Northern Ave, Boston, MA       | Seaport Ln
 5 Seaport Ln, Boston, MA 02210 | Seaport Ln
(2 rows)
```

FIN

BUY OUR BOOKS

HTTP://WWW.POSTGIS.US