O'REILLY
OSCON™
Open Source Convention

Tips and Tricks for Writing PostGIS Spatial Queries

Leo Hsu and Regina Obe
Paragon Corporation http://www.paragoncorporation.com
PostGIS in Action http://www.manning.com/obe (our upcoming book!)

Useful Links:
PostGIS http://postgis.refractions.net
PostGIS Trac and Wiki http://trac.osgeo.org/postgis
Boston GIS http://www.bostongis.com
Postgres On Line Journal http://www.postgresonline.com

PARAGON CORPORATION

PostGIS
Geospatial Objects for PostgreSQL

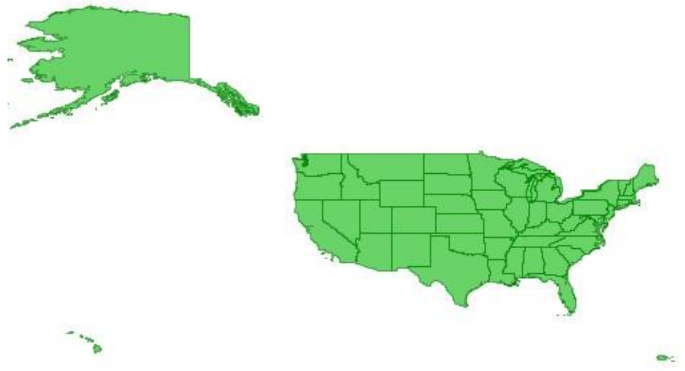
New Features in PostGIS 1.4

- Faster Aggregates
- Cascaded Union (union 40,000 polygons in seconds instead of in your dreams) (need GEOS 3.1.1 and above)
- Prepared Geometries – for improved ST_Intersects, ST_Within, ST_Contains (need GEOS 3.1+)
- It is out

Speed Test 1: Polygon union

```
2895 records unioned into 1 record

SELECT ST_Union(the_geom) FROM USMap;
```

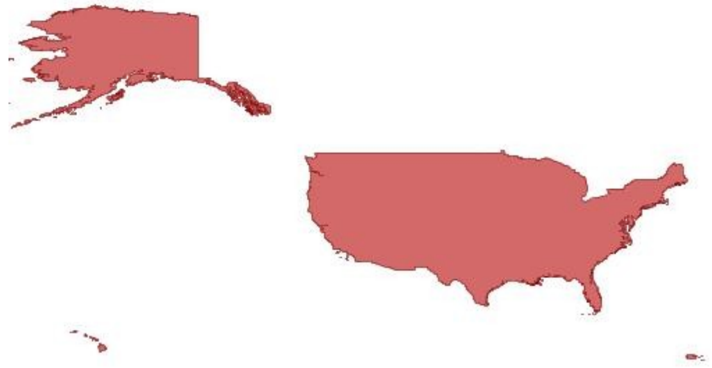


In PostGIS 1.3.6 (PostgreSQL 8.3/8.4)

Still chugging after 12 minutes.

In PostGIS 1.4 (PostgreSQL 8.3/8.4)

Takes 26 secs



O'REILLY OSCON Open Source Convention

Speed Test 2: Union and Transform

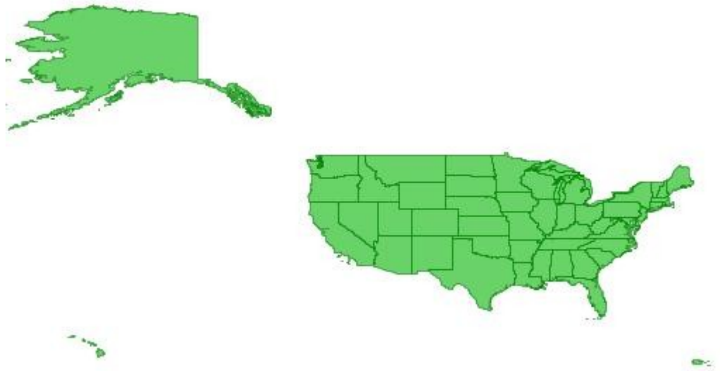
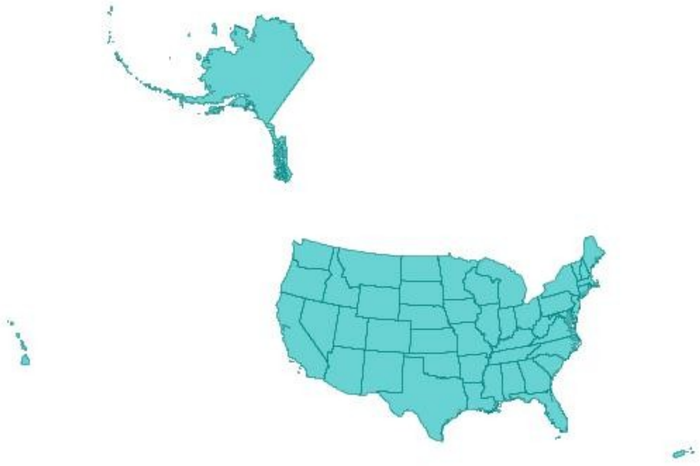
2895 records unioned and transformed

From NAD 83 longlat to US National Atlas Equal Area Meters into 53 records

```
SELECT state, state_fips, ST_Union(ST_Transform(the_geom,2163)) As the_geom
INTO us.states
FROM statesp020 As s
GROUP BY s.state, s.state_fips;
```

In PostGIS 1.3 -- Still running after 10 minutes

In PostGIS 1.4 -- Takes 18 secs

O'REILLY OSCON Open Source Convention

PostgreSQL 8.4 Enhancements

- Windowing Functions
- Common Table Expressions and Recursive Common Table Expressions
- Unnest, array_agg
- More efficient query planner – better results with COUNT, IN and EXISTS and INTERSECTS and EXCEPT clauses, improved Hash indexes
- Faster database restore
- PgMigrator for in place upgrade from 8.3 to 8.4

O'REILLY OSCON™ Open Source Convention

Tip: Add indexes AFTER bulk insert

Bulk insert

```
INSERT INTO sometable(field1,field2,…)
SELECT field1,field2, ..
FROM super_lots_of_data
```

Spatial index on geometry columns

```
CREATE INDEX idx_sometable_the_geom ON sometable USING
    gist(the_geom);
```

Btree index on attribute columns used in common WHERE clauses

```
CREATE INDEX idx_sometable_imp_attrib ON sometable USING
    btree(imp_attrib);
```

Tip: Always run vacuum analyze after bulk insert

Bulk Insert

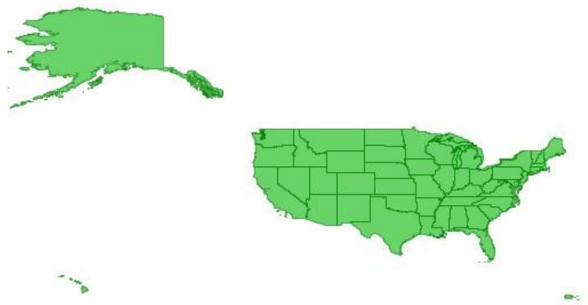
```
INSERT INTO sometable(field1,field2,…)
SELECT field1,field2, ..
FROM super_lots_of_data
```

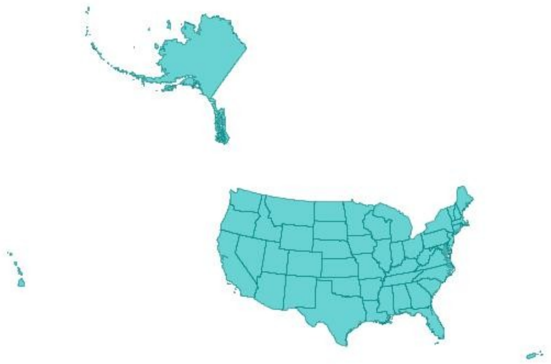
Run VACUUM ANALYZE and add a verbose to see what is happening.

```
VACUUM ANALYZE VERBOSE sometable;
```

Tip: Keep data in form most suitable for your workload

If you do mostly distance calculations and can find suitable SRID to cover your area use that.





WGS 84 -

--yields 1.23567.. Degrees
(what do we do with this?)

```
  SELECT a.state As st_a, b.state As st_b,
       ST_Distance(a.the_geom, b.the_geom) As
dist_deg
  FROM us.states_wgs84 AS a
               CROSS JOIN us.states_wgs84 AS b
 WHERE a.state = 'Maine'
     and b.state = 'Rhode Island';
```

--yields -- 131,103 meters

```
SELECT a.state As st_a, b.state As st_b,
ST_Distance(a.the_geom, b.the_geom) As
dist_m
  FROM us.states AS a
       CROSS JOIN us.states AS b
 WHERE a.state = 'Maine'
and b.state = 'Rhode Island';
```

O'REILLY OSCON Open Source Convention

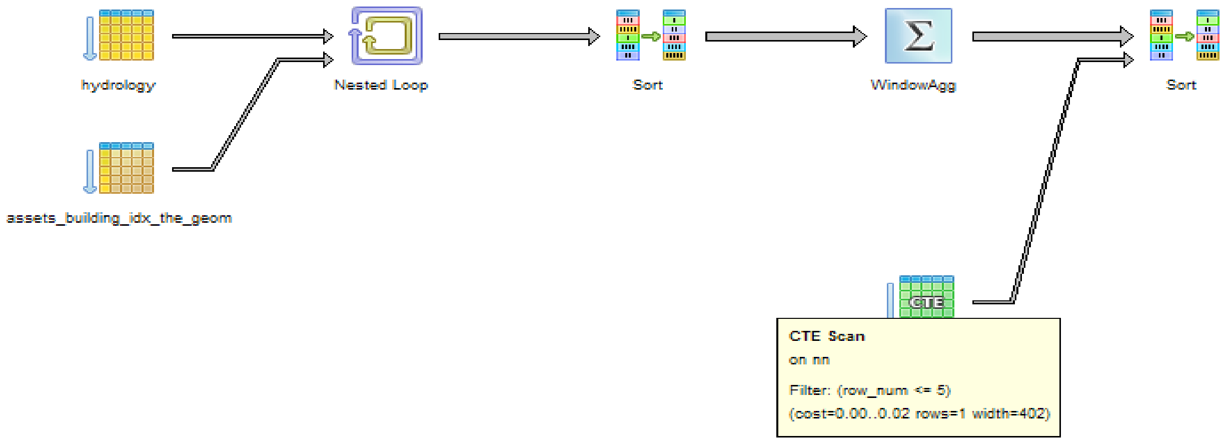
Tip: Use the graphical explain in PgAdmin

```
WITH nn AS (
SELECT h.gid AS hyd_id,
      h.hyd_name,ROW_NUMBER() OVER(PARTITION BY h.gid
                    ORDER BY ST_Distance(h.the_geom, b.the_geom)) As
row_num,
      b.bldg_name,b.bldg_type,ST_Distance(b.the_geom, h.the_geom) As
dist_to_lake
FROM building As b INNER JOIN
   hydrology As h ON (ST_DWithin(h.the_geom, b.the_geom, 50000) )
   )
SELECT nn.*
FROM nn
WHERE nn.row_num <= 5
ORDER BY nn.hyd_name, nn.hyd_id, nn.row_num;
```

PgAdmin 1.10 has cute icons to show off new windows agg and CTE use

Thickness of arrows gives relative cost of each segment of plan.

Click on an icon and get cost detail for that part.



```
CTE Scan
on nn
Filter: (row_num <= 5)
(cost=0.00..0.02 rows=1 width=402)
```

Tip: Plain is nice too but a lot of information

```
WITH nn AS (
SELECT h.gid AS hyd_id,
          h.hyd_name,ROW_NUMBER() OVER(PARTITION BY h.gid
                          ORDER BY ST_Distance(h.the_geom, b.the_geom)) As row_num,
          b.bldg_name,b.bldg_type,ST_Distance(b.the_geom, h.the_geom) As dist_to_lake
FROM building As b INNER JOIN
   hydrology As h ON (ST_DWithin(h.the_geom, b.the_geom, 50000) )
   )
SELECT nn.*
FROM nn
WHERE nn.row_num <= 5
ORDER BY nn.hyd_name, nn.hyd_id, nn.row_num;
```

EXPLAIN VERBOSE ANALYZE sql_here
lots of info at one glance sometimes too much .
8.4 now with verbose provides detail of output of fields and memory use

```
--------------------------------------------------------------------------------------------------------------------------------------------------------------------
--
 Sort  (cost=30.28..30.29 rows=1 width=402) (actual time=1149.990..1149.999 rows=20 loops=1)
   Output: nn.hyd_id, nn.hyd_name, nn.row_num, nn.bldg_name, nn.bldg_type, nn.dist_to_lake
   Sort Key: nn.hyd_name, nn.hyd_id, nn.row_num
   Sort Method:  quicksort  Memory: 19kB
   CTE nn
     ->  WindowAgg  (cost=30.22..30.25 rows=1 width=980) (actual time=773.909..1146.397 rows=1968 loops=1)
           Output: h.gid, h.hyd_name, row_number() OVER (?), b.bldg_name, b.bldg_type, st_distance(b.the_geom, h.the_geom)
           ->  Sort  (cost=30.22..30.23 rows=1 width=980) (actual time=773.847..777.443 rows=1968 loops=1)
                 Output: h.gid, h.hyd_name, b.bldg_name, b.bldg_type, b.the_geom, h.the_geom
                 Sort Key: h.gid, (st_distance(h.the_geom, b.the_geom))
                 Sort Method:  external merge  Disk: 1736kB
                 ->  Nested Loop  (cost=0.00..30.21 rows=1 width=980) (actual time=0.149..755.012 rows=1968 loops=1)
                       Output: h.gid, h.hyd_name, b.bldg_name, b.bldg_type, b.the_geom, h.the_geom
                       Join Filter: (_st_dwithin(h.the_geom, b.the_geom, 50000::double precision) AND (h.the_geom && st_expand(b.the_geom, 50000::double precision)))
                       ->  Seq Scan on hydrology h  (cost=0.00..1.04 rows=4 width=354) (actual time=0.008..0.015 rows=4 loops=1)
                             Output: h.gid, h.hyd_name, h.hyd_type, h.the_geom
                       ->  Index Scan using assets_building_idx_the_geom on building b  (cost=0.00..7.27 rows=1 width=626) (actual time=0.054..0.748 rows=492
loops=4)
                             Output: b.gid, b.bldg_name, b.bldg_type, b.the_geom
                             Index Cond: (b.the_geom && st_expand(h.the_geom, 50000::double precision))
   ->  CTE Scan on nn  (cost=0.00..0.02 rows=1 width=402) (actual time=773.920..1149.886 rows=20 loops=1)
         Output: nn.hyd_id, nn.hyd_name, nn.row_num, nn.bldg_name, nn.bldg_type, nn.dist_to_lake
```

O'REILLY OSCON Open Source Convention

Nearest neighbor queries

Find n closest geometries

- Use ST_DWithin wherever possible (though this requires you guess at bounding range of farthest closest)
- Scenario 1 – 1 reference geom, many geometries – find n closest. USE LIMIT with ORDER BY.
- Scenario 2 – Many reference geoms, many geometries, find closest. Use DISTINCT ON.
- Scenario 3 – Many reference geoms, many geometries, find n closest to each reference geom. Use windowing functions. (Requires PostgreSQL 8.4)

O'REILLY OSCON™ Open Source Convention

NN Scenario 1: 1 reference geom, many geometries

USE ST_DWithin so you can take advantage of indexes.

USE LIMIT, ORDER BY distance to limit number

```
SELECT b.bldg_name, b.bldg_type, ST_Distance(b.the_geom, h.the_geom) As dist_to_lake
FROM building As b INNER JOIN hydrology As h
ON ST_DWithin(h.the_geom, b.the_geom, 50000)
WHERE h.gid = 4
ORDER BY ST_Distance(b.the_geom, h.the_geom)
LIMIT 5;
```

```
SELECT b.bldg_name, b.bldg_type, ST_Distance(b.the_geom, h.the_geom) As dist_to_lake
FROM building As b INNER JOIN
   (SELECT ST_GeomFromText('LINESTRING(50858 901316,250860 901318)',26986) As the_geom) As h
ON ST_DWithin(h.the_geom, b.the_geom, 50000)
ORDER BY ST_Distance(b.the_geom, h.the_geom)
LIMIT 5;
```

O'REILLY OSCON Open Source Convention

NN Scenario 2: Many reference geoms, many geoms, find closest 1

USE ST_DWithin so you can take advantage of indexes.

USE DISTINCT ON with ORDER BY id, distance to get only one back for each reference

Find closest building to each water body

```
SELECT DISTINCT ON(h.gid) h.gid AS hyd_id, h.hyd_name, b.bldg_name, b.bldg_type,
       ST_Distance(b.the_geom, h.the_geom) As dist_to_lake
FROM building As b INNER JOIN hydrology As h
ON ST_DWithin(h.the_geom, b.the_geom, 50000)
ORDER BY h.gid, ST_Distance(b.the_geom, h.the_geom);
```

NN Scenario 3: Many reference geoms, many geoms, find n closest to each reference geom

USE ST_DWithin so you can take advantage of indexes.

For Windowing functions, need PostgreSQL 8.4.

Find 5 closest buildings to each water body arbitrarily pick ties .

If you want to include ties use RANK() instead of ROW_NUMBER())

```
SELECT nn.*
FROM (
SELECT
     h.gid AS hyd_id,
     h.hyd_name,
     ROW_NUMBER() OVER(PARTITION BY h.gid ORDER BY ST_Distance(h.the_geom,b.the_geom))
     As row_num,
     b.bldg_name,
     b.bldg_type,
     ST_Distance(b.the_geom, h.the_geom) As dist_to_lake
FROM building As b INNER JOIN hydrology As h
ON ST_DWithin(h.the_geom, b.the_geom, 50000) As nn
WHERE nn.row_num <= 5
ORDER BY nn.hyd_name, nn.hyd_id, nn.row_num;
```

Tip: If what is too slow, ask the opposite question

If you know what is then you can determine what is not.

Sometimes asking what is not is faster than asking what is.

Tip: Don’t forget about the left join

You know what is not if you can ask for the universe and what is.

How do you ask what is without losing the universe?

Use a LEFT JOIN instead of an INNER JOIN

```
SELECT t1.field1, t1.field2 FROM t1 LEFT JOIN t2 ON (the what is condition) WHERE t2.some_non_null_key IS NULL;
```

Example: What has no close neighbors

Find all geometries that have no reference geometries within x units.

Use ST_DWithin because it will use an index (but how?)

Find all that have close neighbors and throw them out.

What is left are the ones with no close neighbors.

```
SELECT h.house_name, h.house_id
FROM houses As h LEFT JOIN rivers As r
ON ST_DWithin(h.the_geom, r.the_geom, 3000)
WHERE r.river_id IS NULL;
```

Tip: Simplify your geometry to gain performance

```
SELECT a.state As st_a, b.state As st_b,
   ST_NPoints(a.the_geom) As num_points_ca,
   ST_NPoints(b.the_geom) As num_points_tx,
   ST_NPoints(ST_SimplifyPreserveTopology(a.the_geom,700)) As num_points_simp_ca,
   ST_NPoints(ST_SimplifyPreserveTopology(b.the_geom,700)) As num_points_simp_tx
FROM states AS a CROSS JOIN states AS b
WHERE a.state = 'California' and b.state = 'Texas';
```

The more vertices you have the slower your distance calculation: CA has 10,210 pts and TX has 12,167 pts. After simplification , CA has 873 pts, TX has 1653 pts.

```
SELECT a.state As a, b.state as b, ST_Distance(a.the_geom, b.the_geom) As dist_m
FROM states AS a CROSS JOIN states AS b
WHERE a.state = 'California' AND b.state = 'Texas';
```

Result: 745222.745755735 meters (~1.5 minutes)

```
SELECT a.state As st_a, b.state As st_b,
   ST_Distance(ST_SimplifyPreserveTopology(a.the_geom,700),
   ST_SimplifyPreserveTopology(b.the_geom,700)) As dist_m
FROM states AS a CROSS JOIN states AS b
WHERE a.state = 'California' and b.state = 'Texas';
```

Result: 745258.2697633 meters (~1.5 secs)

We increased our speed 60 fold with minimum loss in accuracy.

O'REILLY OSCON™ Open Source Convention

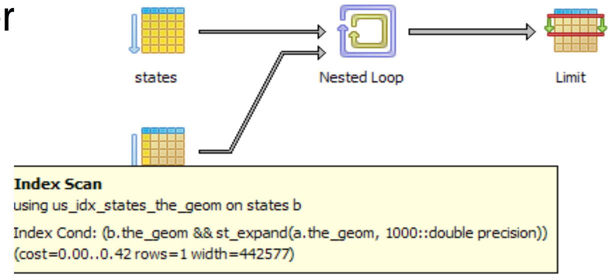
Tip: Use Simplify to speed up queries (Be careful to not throw away index)

- Which pairs of states are within 1000 meters of each other

--Uses an index but more costly DWithin check (893 ms)

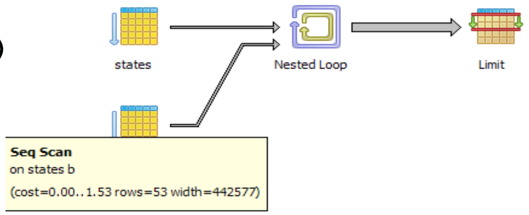
--As you increase limit count this starts lossing (limit 2: 10,342 ms)

```
 SELECT a.state As st_a, b.state As st_b
  FROM states AS a CROSS JOIN states AS b
 WHERE NOT (a.state = b.state)
   AND ST_DWithin(a.the_geom, b.the_geom, 1000)
 LIMIT 1;
```



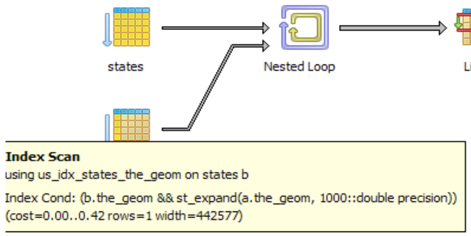
Index Scan
using us_idx_states_the_geom on states b
Index Cond: (b.the_geom && st_expand(a.the_geom, 1000::double precision))
(cost=0.00..0.42 rows=1 width=442577)

```
 --doesn't use an index but less costly dwithin check (9,032 ms)
 -- but at 2 or more beats the above for this small dataset (limit 2: 9,734 ms)
 SELECT a.state As st_a, b.state As st_b
  FROM states AS a CROSS JOIN states AS b
 WHERE
 NOT (a.state = b.state)
   AND ST_DWithin(ST_SimplifyPreserveTopology(a.the_geom,700),
   ST_SimplifyPreserveTopology(b.the_geom,700),1000)
 LIMIT 1;
```



Seq Scan
on states b
(cost=0.00..1.53 rows=53 width=442577)

```
--uses an index and less costly dwithin check (422 ms, at limit 2: 656 ms)
--If you dared run this across all the states -- (no limit )
-- finishes in 42,687 ms, other 2 you'd be waiting a long time
 (note can get faster with even more simplification)
SELECT a.state As st_a, b.state As st_b
  FROM states AS a CROSS JOIN states AS b
 WHERE
 NOT (a.state = b.state)
   AND (ST_Expand(a.the_geom,700) && b.the_geom)
AND _ST_DWithin(ST_SimplifyPreserveTopology(a.the_geom,700),  ST_SimplifyPreserveTopology(b.the_geom,700),1000)
LIMIT 1;
```



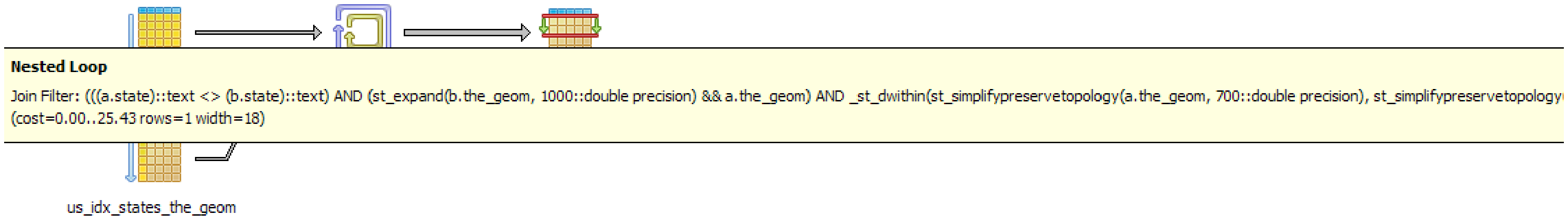
Index Scan
using us_idx_states_the_geom on states b
Index Cond: (b.the_geom && st_expand(a.the_geom, 1000::double precision))
(cost=0.00..0.42 rows=1 width=442577)

Compartmentalize common used constructs
an SQL function is transparent to the planner

If your function can benefit from an index, try to make it transparent to the planner by using SQL – the below still uses an index

```
CREATE FUNCTION sql_ST_DWithin_Simplify(geom1 geometry, geom2 geometry, dist double precision,
simplify_tolerance double precision)
RETURNS boolean
 AS
  $$ SELECT ST_Expand($1, $3) && $2 AND ST_Expand($2, $3) && $1
AND _ST_DWithin(ST_SimplifyPreserveTopology($1,$4),ST_SimplifyPreserveTopology($2,$4), $3)
$$
language 'sql' IMMUTABLE;

---uses an index and less costly dwithin (limit 5: 1906 ms, no limit: 42,141 ms)
SELECT a.state As st_a, b.state As st_b
  FROM states AS a
CROSS JOIN states AS b
 WHERE
 NOT (a.state = b.state)
AND sql_ST_DWithin_Simplify(a.the_geom, b.the_geom, 1000,700)
 limit 2;
```



Nested Loop

Join Filter: (((a.state)::text <> (b.state)::text) AND (st_expand(b.the_geom, 1000::double precision) && a.the_geom) AND _st_dwithin(st_simplifypreservetopology(a.the_geom, 700::double precision), st_simplifypreservetopology

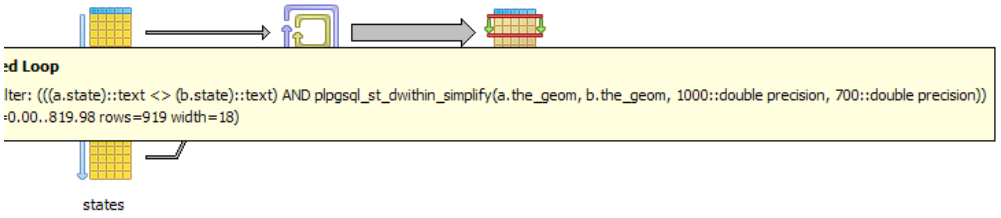
(cost=0.00..25.43 rows=1 width=18)

This function is opaque so planner doesn't know an index might help

```
 CREATE FUNCTION plpgsql_ST_DWithin_Simplify(geom1 geometry, geom2 geometry,
     dist double precision, simplify_tolerance double precision)
RETURNS boolean
 AS
  $$
BEGIN
  RETURN ST_Expand($1, $3) && $2 AND ST_Expand($2, $3) && $1
AND _ST_DWithin(ST_SimplifyPreserveTopology($1,$4),ST_SimplifyPreserveTopology($2,$4), $3);
END;
$$
language 'plpgsql' IMMUTABLE;
```

---Does not use index (function is opaque) but less costly dwithin (limit 5: 1859ms, no limit 55,500ms)
—-Strangley on PostGIS 1.4 and PostgreSQL 8.4 this is slightly faster than the sql function
– for the 1 - 5 limit case. But for full is 55,500ms which is slower.
– Presumably cost of loading up the index is adding more percent wise to limit times.

```
SELECT a.state As st_a, b.state As st_b
  FROM states AS a
CROSS JOIN states AS b
 WHERE
 NOT (a.state = b.state)
AND plpgsql_ST_DWithin_Simplify(a.the_geom, b.the_geom, 1000,700)
 Limit 5;
```



```
ed Loop
lter: (((a.state)::text <> (b.state)::text) AND plpgsql_st_dwithin_simplify(a.the_geom, b.the_geom, 1000::double precision, 700::double precision))
=0.00..819.98 rows=919 width=18)
```

O'REILLY OSCON Open Source Convention

Tip: Use CTEs to organize queries (Dicing exmaple)

Dice Texas using a 10x10 or x by y count grid

Using 3 CTEs

```
WITH

usext AS -- Define a CTE to store our base variables (extent and our x,y grid count)
(SELECT ST_SetSRID(CAST(ST_Extent(the_geom) As geometry),2163) As the_geom_ext, 10 as x_gridcnt, 10 as y_gridcnt
 FROM states As s
 WHERE state = 'Texas'),

grid_dim AS -- Define a CTE to store our grid dimension width and height that uses usext
(SELECT
    (ST_XMax(the_geom_ext) - ST_XMin(the_geom_ext))/x_gridcnt As g_width,
    ST_XMin(the_geom_ext) As xmin, ST_xmax(the_geom_ext) As xmax,
    (ST_YMax(the_geom_ext) - ST_YMin(the_geom_ext))/y_gridcnt As g_height,
    ST_YMin(the_geom_ext) As ymin, ST_YMax(the_geom_ext) As ymax
FROM usext),

grid As -- Define CTE to store our grid that uses usext and grid_dim
(SELECT x, y, ST_SetSRID(ST_MakeBox2d(ST_Point(xmin + (x - 1)*g_width, ymin + (y-1)*g_height),
    ST_Point(xmin + x*g_width, ymin + y*g_height)), 2163) As grid_geom
FROM
    (SELECT generate_series(1,x_gridcnt) FROM usext) As x CROSS JOIN
    (SELECT generate_series(1,y_gridcnt) FROM usext) As y CROSS JOIN
    grid_dim
)

--Use grid to clip Texas and bulk insert new clipped to a new on-the fly table
SELECT state, state_fips, ST_Intersection(s.the_geom, grid_geom) As newgeom
INTO us.texas_diced_g10
FROM states As s INNER JOIN grid ON s.state = 'Texas' AND ST_Intersects(s.the_geom, grid.grid_geom);
```

Texas diced into 100x100 grids -- takes 343,578 ms
Texas diced into 10x10 grids -- takes 4,797 ms

Texas before and after chainsaw massacre

Tip: Use populate_geometry_columns in PostGIS 1.4

The fast way to register a new geometry and put constraints on it. No need for AddGeometryColumn if you have PostGIS 1.4

```
SELECT populate_geometry_columns('us.texas_diced_g10'::regclass);
```

Automatically adds an entry to geometry_columns table for us.texas_diced_g10 by inspecting our table for type, dimension, and SRID of geometry columns.

Creates a constraint on the new table column if it can (SRID, geometry type, dimension check constraints)

Tip: Use the knife trick to bisect geometries

```
SELECT foo.path[1] As gid, ST_AsText(ST_SnapToGrid(foo.geom,
  0.0000001)) As wktpoly
FROM (SELECT g1.geom2 As the_knife_cut,
  (ST_Dump(ST_Difference(g1.geom1, g1.geom2))).*
FROM (SELECT ST_GeomFromText('POLYGON((2 4.5,3 2.6,3 1.8,2 0,-
  1.5 2.2,0.056 3.222,-1.5 4.2,2 6.5,2 4.5))') As geom1,
ST_Buffer(ST_GeomFromText('LINESTRING(-0.62 5.84,-0.8
  0.59)'),0.00000001) As geom2) AS g1
WHERE ST_Intersects(g1.geom1,g1.geom2)) As foo;
```