

Boston Geographic Information Systems

Home About Boston GIS

Table Of Contents

UMN Mapserver Tutorials
How to Use different kinds of datasources in UMN Mapserver layers
Using Mapserver as a WMS Client.

PostGIS and PostgreSQL Tutorials
Solving the Nearest Neighbor Problem in PostGIS
PostGIS Nearest Neighbor: A Generic Solution - Much Faster than Previous Solution
Part 1 - PostGIS and SharpMap in ASP.NET 2.0 using VB.NET: Compiling SharpMap with PostGIS
Part 2 - PostGIS and SharpMap in ASP.NET 2.0 using VB.NET: Displaying the Maps
Part 1: Getting Started With PostGIS: An almost Idiot's Guide
Part 2: Introduction to Spatial Queries and SFSQL
Part 3: PostGIS Loading Data from Non-Spatial Sources
PLR Part 1: Up and Running with PL/R (PLR) in PostgreSQL: An almost Idiot's Guide
PLR Part 2: PL/R and PostGIS

Miscellaneous Tutorials/Cheatsheets/Examples
OGR2OGR Cheatsheet
Using OpenLayers: Part 1
Using OpenLayers: Part 2

PostGIS Code Snippets
Extent Expand Buffer Distance: PostGIS - ST_Extent, Expand, ST_Buffer, ST_Distance
PostGIS GeomFromText
Intersects Intersection: PostGIS - ST_Intersects, ST_Intersection
PostGIS MakeLine ST_MakeLine Examples
PostGIS MakePoint
PointFromText, LineFromText, ST_PointFromText, ST_LineFromText OGC functions - PostGIS
PostGIS Simplify
Summary (pre 1.3.1 name), ST_Summary (+1.3.1)
PostGIS: ST_Translate

UMN Mapserver Tutorials

How to Use different kinds of datasources in UMN Mapserver layers

One of the very great things about the UMN Mapserver Web system is that it can support numerous kinds of datasources. In this brief excerpt we will provide examples of how to specify the more common data sources used for layers. The examples below are for Mapserver 4.6, but for the most part are applicable to lower versions.

File locations for file based datasources such as ESRI Shape and MapInfo tab files are defined relative to the **SHAPEPATH** attribute or as absolute paths. For example the beginning declaration of your .map file might look something like the below

```
MAP
    #
    # Start of map file
    #

    NAME MYMAP

    EXTENT  732193.725550 2904132.702662 799614.090681 2971466.288170

    SIZE 500 500
    SHAPEPATH "c:\mydata\"
    :
    :
```

ESRI Shapefile

The most common kind of data used in UMN Mapserver is the ESRI shapefile which has a .shp extension. For this kind of datasource you simply specify the location of the file without even specifying the extension. Below is a sample declaration of a polygon layer that uses a shape file

```
    LAYER
        NAME buildings
        TYPE POLYGON
        STATUS DEFAULT
        DATA buildings
        PROJECTION
            "init=epsg:2249"
        END
        CLASS
            OUTLINECOLOR 10 10 10
        END
    END
```

MapInfo Tab Files

Many datasources are available to mapserver via the GDAL OGR driver. Map Info is one of those datasources. Below example is what a mapinfo layer definition looks like. Note the tab file specified should be placed in the folder denoted by SHAPEPATH at top of map file

```
    LAYER
        NAME buildings
        STATUS DEFAULT
        MINSCALE 7000
        CONNECTIONTYPE OGR
        CONNECTION "buildings.tab"
        TYPE POLYGON
        PROJECTION
            "init=epsg:2249"
        END
        # -- MapInfo has projection information built in the tab file
        # -- so you can often auto read this information with the below
        #PROJECTION
        # AUTO
        #END
        CLASS
            OUTLINECOLOR 10 10 10
        END
    END
```

PostGIS Layer

Mapserver has a custom driver for the PostGIS spatial database. In order to use this, your mapserver cgi or mapscript must be compiled with the PostGIS driver. Below is what a postgis mapserver layer looks like.

```
    LAYER
      CONNECTIONTYPE postgis
      NAME "buildings"
      CONNECTION "user=dbuser dbname=mydb host=myserver"
```

```
     # the_geom column is the name of a spatial geometry field in the table buildings
     DATA "the_geom from buildings"
     STATUS DEFAULT
     TYPE POLYGON
     # Note if you use a filter statement - this is basically like a where clause of the sql
statement
     FILTER "storyhg > 2"
     CLASS
          OUTLINECOLOR 10 10 10
     END
   END
```

More complex PostGIS layer

```
LAYER
    NAME "projects"
    CONNECTIONTYPE postgis
    CONNECTION "user=myloginuser dbname=mydbname host=mydbhost password=mypass"
    DATA "the_geom FROM (SELECT a.projid, a.projname, a.projtype, a.projyear, a.pid, parc.
the_geom
                        FROM projects a INNER JOIN parcels parc ON a.parcel_id = parc.pid
                         WHERE a.projyear = 2007) as foo USING UNIQUE projid USING
SRID=2249"
    STATUS OFF
    TYPE POLYGON
    CLASS
        NAME "Business Projects"
        EXPRESSION ('[projtype]' = 'Business')
        STYLE
            OUTLINECOLOR 204 153 51
            WIDTH 3
        END
    END
    CLASS
        NAME "Community Projects"
        EXPRESSION ('[projtype]' = 'Community')
        STYLE
            OUTLINECOLOR 204 0 0
            WIDTH 3
        END
    END

    PROJECTION
        "init=epsg:2249"
    END
    METADATA
     "wms_title" "Projects"
     "wfs_title" "Projects"
      gml_include_items "all"
      wms_include_items "all"
    END
    DUMP TRUE
    TOLERANCE 10
END
```

WMS Layer

Mapserver has the ability to act as a WMS Server as well as a WMS Client. The WMS Client capabilities are accessed by defining WMS layers that connect to WMS servers. Below is an example of a WMS layer using the Microsoft Terraservices WMS Server.

```
   LAYER
         NAME "msterraservicedoq"
         TYPE RASTER
         STATUS DEFAULT
         CONNECTION "http://terraservice.net/ogcmap.ashx?"
         CONNECTIONTYPE WMS
         MINSCALE 3000
         MAXSCALE 20000
         #DEBUG ON
         METADATA
           "wms_srs"             "EPSG:26919"
           "wms_name"            "doq"
           "wms_server_version"  "1.1.1"
           "wms_format"          "image/jpeg"
           "wms_style"         "UTMGrid_Cyan"
           "wms_latlonboundingbox" "-71.19 42.23 -71 42.40"
         END
    END
```

Using Mapserver as a WMS Client.

Example mapserver map that calls microsoft terraservice WMS.

In this example we show how to use Mapserver as a WMS client by utilizing Microsoft's Terra Service WMS server. For more details about Microsft's OGC WMS check out the GetCapabilities of Microsoft Terraservice.

download

PostGIS and PostgreSQL Tutorials

Solving the Nearest Neighbor Problem in PostGIS

A common problem encountered in GIS is the Nearest Neighbor problem. In a nutshell the problem is to find the x number of nearest neighbors given a geometry and n geometries of data.

The nearest neighbor crops up in other disciplines as well, except in other disciplines the units of measurement are not spatial distance but instead some sort of matrix distance. For this particular talk, I will focus on the GIS nature of the problem and specifically solving it in PostGIS.

The easiest and most naive way to do this is to do an exhaustive search of the whole space comparing the distances against the geometry of interest and sorting them. This solution while intuitive and easy to write is the slowest way of doing this.

If you were to write such a simple solution in PostgreSQL, then it would take something of the form shown below. The below gives you the nearest 5 neighbors from a reference row with gid = 1.

```
SELECT g1.gid As gref_gid, g1.description As gref_description,
g2.gid As gnn_gid, g2.description As gnn_description
    FROM sometable As g1, sometable As g2
WHERE g1.gid = 1 and g1.gid <> g2.gid
ORDER BY distance(g1.the_geom,g2.the_geom)
LIMIT 5
```

Now for moderate to large datasets, this exercise becomes painfully slow because distance is not an indexable function because it involves relations between two entities.

If you knew something about your data like there are no neighbors past 300 units away from your geometry of interest or that at that point you could care less what lies past 300 units, then you could significantly improve the speed of this query by writing it like this:

```
 SELECT g1.gid As gref_gid, g1.description As gref_description, g2.gid As gnn_gid,
       g2.description As gnn_description
   FROM sometable As g1, sometable As g2
   WHERE g1.gid = 1 and g1.gid <> g2.gid AND expand(g1.the_geom, 300) && g2.the_geom
   ORDER BY distance(g1.the_geom,g2.the_geom)
   LIMIT 5
```

If you needed to run this for more than one record in sometable, then you can either put this in a loop and run it for each record of interest in sometable. Alternatively you could create an SQL function that returns a table for each record and use it like shown below.

```
CREATE OR REPLACE FUNCTION fnsometable_nn(gid integer, geom1 geometry, distguess double precision, n
umnn integer)
    RETURNS SETOF sometable AS
    $BODY$
    SELECT g2.*
        FROM sometable As g2
            WHERE $1 <> g2.gid AND expand($2, $3) && g2.the_geom
        ORDER BY distance(g2.the_geom, $2) LIMIT $4
    $BODY$ LANGUAGE 'sql' VOLATILE;
```

To use the function to run for the first 100 records in sometable and return the 5 closest neighbors for each of those records, you would do this

```
SELECT g1.gid, g1.description, (fnsometable_nn(g1.gid, g1.the_geom, 300, 5)).gid As nn_gid,
    (fnsometable_nn(g1.gid, g1.the_geom, 300, 5)).description as nn_description,
    distance((fnsometable_nn(g1.gid, g1.the_geom, 300, 5)).the_geom,g1.the_geom) as dist
    FROM (SELECT * FROM sometable ORDER BY gid LIMIT 100) g1
```

A lot of people are probably thinking "HUH" at this point. I will now wave my hand and leave it as an exercise for people to figure out why this works. I will say this much. I am using a feature in PostgreSQL which from my readings is an accident of design and only works with functions written in SQL and C language. This accident will probably be later corrected in future versions so don't expect this to work moving forward.

Returning tables in the select part of an sql statement is super non-standard and should I say bizarre and unelegant, but is the only way to do it currently in PostgreSQL where the you have a set returning function whose inputs depend on values from a relating table. This is because PostgreSQL currently does not support subselects and set returning functions in the FROM clause that dynamically change based on inputs from other table fields. In fact most databases do not support this.

Unfortunately PostgreSQL doesn't support a predicate similar to the CROSS APPLY predicate that SQL Server 2005 has. I think Oracle does by pretty much just leaving out the CROSS APPLY predicate. CROSS APPLY is more elegant looking I think in general and also more self-documenting in the sense that when you see the clause - you know what is being done.

SQL Server which I use very often - introduced this feature in the SQL Server 2005 offering and in a few cases its a life-saver.

I think the next next release of PostgreSQL will probably support a feature similar to this. Just to demonstrate how powerful this feature can be, if we had the cross apply feature, we could do away with our intermediate set returning function and simply write our query like this.

```
SELECT g1.gid, g1.description, nn.gid As nn_gid, nn.description As nn_description, distance(nn.the_g
eom,g1.the_geom) as dist
FROM (SELECT * FROM sometable ORDER BY gid LIMIT 100) g1 CROSS APPLY
    (SELECT g2.*
        FROM sometable As g2
        WHERE g1.gid <> g2.gid AND expand(g1.the_geom, 300) && g2.the_geom)
ORDER BY distance(g2.the_geom, g1.the_geom) LIMIT 5) As nn
```

This would alleviate the need to create a new nn function for each table or kind of query we are formulating.

Its always interesting to see how different databases tackle the same issues. Oracle Locator and Oracle Spatial for example does have an SDO_NN and SDO_NN_DISTANCE functions which magically does all this for you, but I haven't used it so not sure of its speed or if it has any caveats. One I did notice from reading is that it doesn't deal with where clauses too well.

PostGIS Nearest Neighbor: A Generic Solution - Much Faster than Previous Solution

A generic solution to PostGIS nearest neighbor

After some heavy brainstorming, I have come up with a faster and more generic solution to calculating nearest neighbors than my previous solutions. For the gory details on how I arrived at this solution - please check out the following blog entries Boston GIS Blog entries on Nearest Neighbor Search.

In the sections that follow I will briefly describe the key parts of the solution, the strengths of the solution, the caveats of the solution, and some general use cases.

The parts of the solution

The solution is composed of a new PostgreSQL type, and 2 PostgreSQL functions. It involves a sequence of expand boxes that fan out from the bounding box of a geometry to as far out as you want. Below are the descriptions of each part.

- **pgis_nn** - a new type that defines a neighbor solution - for a given near neighbor solution it gives you, the primary key of the neighbor and the distance it is away from the reference object
- **expandoverlap_metric** - this is a helper function that returns the first fanned out box that an object is found in relative to the reference geometry. It returns an integer where the integer varies from 0 (in bounding box) to maxslices where maxslices is how many slices you want to dice up your larger expand box.
- **_pgis_fn_nn** - this is the workhorse function that should never be directly called - it is a pgsql function that returns for each geometry *k* near neighbors
- **pgis_fn_nn**(geom1 geometry, distguess double precision, numnn integer, maxslices integer, lookupset varchar(150), swhere varchar(5000), sgid2field varchar(100), sgeom2field varchar(100))) - this is our interface function written in language *sql* to fool the planner into thinking we are using an sql function. This is to get around certain anomalies of PostgreSQL that prevent pgsql set returning functions from being used if they reference fields in the from part of an SQL statement. Description of the inputs is as follows.
  1. *geom1* - this is the reference geometry.
  2. *distguess* - this is the furthest distance you want to branch out before you want the query to give up. It is measured in units of the spatial ref system of your geometries.
  3. *numnn* - number of nearest neighbors to return.
  4. *maxslices* - this is the max number of slices you want to slice up your whole expand box. The more slices the thinner the slices, but the more iterations. I'm sure there is some statistically function one can run against ones dataset to figure out the optimal number of slices and distance guess, but I haven't quite figured out what that would be.
  5. *lookupset* - this is usually the name of the table you want to search for nearest neighbors. In theory you could throw in a fairly complicated subselect statement as long as you wrap it in parethesis. Something like '(SELECT g.*, b.name FROM sometable g INNER JOIN someothertable b ON g.id = b.id)' . Althought I haven't had a need to do that yet so haven't tested that feature.
  6. *swhere* - this is the where condition to apply to your dataset. If you want the whole dataset to be evaluated, then put 'true'. In theory you can have this correlated with your reference table by gluing parameters together, so can get pretty sophisticated.
  7. *sgid2field* - this is the name of the unique id field in the dataset you want to check for nearest neighbors.
  8. *sgeom2field* - this is the name of the geometry field in the dataset you want to check for nearest neighbors

The code can be downloaded here http://www.bostongis.com/downloads/pgis_nn.txt. I will also be posting this to the PostGIS Wiki once I clean it up a bit more.

Key features

My objective in creating this solution is to give PostGIS the same expressiveness that Oracle's SDO_NN and SDO_NN_DISTANCE functions provide Oracle. Unlike the Oracle solution, this solution is not implemented as an Operator, but instead as a set returning function.

Features are outlined below

- Can handle more than one reference point at a time and return a single dataset. For example you can say give me the 5 nearest neighbors for each of my records in this query and get the result back as a single table.
- You can limit your search to a particular distance range, slice it up as granularly as you want in order to achieve the best resulting speed. Note I implemented this function as linearly expanding boxes to utilize GIST indexes. In hindsight it might have been more efficient processor wise to have the boxes expand like a normal distribution or something like that.
- You can apply where conditions to the sets you are searching instead of searching the whole table - e.g. only give me the 5 nearest hospitals that have more than 100 beds. You can also have the where conditions tied to fields of each record of interest. E.g. include all near neighbors, but don't include the reference point.
- This solution works for all geometries - e.g. you can find point near neighbors to a polygon, polygon near neighbors to a polygon etc. Nearest distance to a line etc.

Caveats

- This function will only work if you give it a primary key that is an integer. I figured an integer is what most people use for their primary keys and is the most efficient join wise.

- This function currently doesn't handle ties. E.g. if you specify 5 neighbors and you have 6 where the last 2 are the same distance away, one will arbitrarily get left out. I think it will be fairly trivial to make it handle this case though.
- If there are no neighbors for a particular reference that meet the stated criteria, that record would get left out. There are workarounds for this which I will try to provide in more advanced use cases.

Use Cases

All the data we will be using is in SRID: 26986. For a refresher course on loading data and setting up indexes, check Part 1: Getting Started With PostGIS: An almost Idiot's Guide

This returned 1000 records in 6 seconds on my dual windows 2003 server
Datasets used: buildings http://www.mass.gov/mgis/lidarbuildingfp2d.htm,
firestations: http://www.mass.gov/mgis/firestations.htm

Find nearest fire station for first 1000 buildings in Boston and sort in decreasing order of distance from the fire station such that the building furthest away from its nearest firestation is at the top.

```
SELECT g1.gid as gid_ref, f.name, g1.nn_dist/1609.344 as dist_miles, g1.nn_gid
FROM (SELECT b.gid,
    (pgis_fn_nn(b.the_geom, 1000000, 1,1000,'fire_stations', 'true', 'gid', 'the_geom')).*
    FROM (SELECT * FROM buildings limit 1000) b) As g1, fire_stations f
    WHERE f.gid = g1.nn_gid
    ORDER BY g1.nn_dist DESC
```

Data set used: Fire stations from above and Massachusetts census tiger roads -http://www.mass.gov/mgis/cen2000_tiger.htm
For each street segment in zip 02109 list the 2 nearest fire stations and the distance in meters from each fire station. Order results by street name, street id and distance from nearest fire station

```
Total query runtime: 687 ms. 296 rows retrieved.

SELECT g1.gid as gid_ref, g1.street, g1.fraddl, g1.toaddl, f.name, g1.nn_dist as dist_meters, g1.
nn_gid
FROM (SELECT b.*,
    (pgis_fn_nn(b.the_geom, 1000000, 2,1000,'fire_stations', 'true', 'gid', 'the_geom')).*
    FROM (SELECT * FROM census2000tiger_arc where zipl = '02109') b) As g1, fire_stations f
    WHERE f.gid = g1.nn_gid
    ORDER BY g1.street, g1.fraddl, g1.gid, g1.nn_dist
```

For each street segment in zip 02109 list the 2 nearest buildings with building footprint > 5000 sq meters and specify how far away they are and the areas of each building.

```
--Total query runtime: 2656 ms. 296 rows retrieved.

SELECT g1.gid as gid_ref, g1.street, g1.fraddl, g1.toaddl, area(b.the_geom) as bldg_area, g1.
nn_dist as dist_meters, g1.nn_gid
FROM (SELECT s.*,
    (pgis_fn_nn(s.the_geom, 1000000, 2,1000,'buildings', 'area(the_geom) > 5000', 'gid',
'the_geom')).*
    FROM (SELECT * FROM census2000tiger_arc where zipl = '02109') s) As g1, buildings b
    WHERE b.gid = g1.nn_gid
    ORDER BY g1.street, g1.fraddl, g1.gid, g1.nn_dist
```

More Advanced Queries

Example of two simultaneous neighbor queries in one query

Layers used
http://www.mass.gov/mgis/schools.htm - schools
http://www.mass.gov/mgis/hospitals.htm - hospitals

Now for this, I think we can safely assume that no street we care about should be further than 30,000 meters from its nearest school and hospital so we will limit our search to 30,000 meters and dice up our box into 10 expand boxes - so boxes expand at a rate of 3000 meters out per expansion.

Give me the nearest elemenary school and the nearest hospital emergency center for street segments in select zipcodes and how far away they are from these street segments.

```
--Total query runtime: 5516 ms.(~5.5 secs). 291 rows retrieved.
SELECT emer.gid as gid_ref, emer.street, emer.fraddl, emer.toaddl, hospitals.name As emer_name,
emer.nn_dist as emer_dist_meters,
        schools.name as sch_name, sch.nn_dist as sch_dist_meters
```

```
FROM (SELECT b.gid, b.street,b.fraddl, b.toaddl,
    (pgis_fn_nn(b.the_geom, 30000, 1,10,'hospitals', 'er_status=''Y''', 'gid', 'the_geom')).*
    FROM (SELECT * FROM census2000tiger_arc where zipl IN('02109', '02110')) As b) As emer INNER
JOIN
    (SELECT b.gid,
    (pgis_fn_nn(b.the_geom, 30000, 1,10,'schools', 'true', 'gid', 'the_geom')).*
    FROM (SELECT * FROM census2000tiger_arc where zipl IN('02109', '02110')) As b) As sch ON emer.
gid = sch.gid
    INNER JOIN hospitals ON hospitals.gid = emer.nn_gid
    INNER JOIN schools ON schools.gid = sch.nn_gid
    ORDER BY emer.street, emer.fraddl, emer.gid
```

Example limit distance exclusion queries

Provide a list of all schools in Massachusetts Boston extent whose closest hospital emergency room is further than 3000 meters away

Using our new neighbor function exclusion query

```
--Total query runtime: 2969 ms. 75 rows retrieved.
SELECT sch.gid as gid_ref, sch.name as sch_name
FROM schools sch LEFT JOIN (SELECT b.gid,
    (pgis_fn_nn(b.the_geom, 3000, 1,2,'hospitals', 'er_status=''Y''', 'gid', 'the_geom')).*
    FROM (SELECT * FROM schools WHERE schools.the_geom && GeomFromText('POLYGON((225446.34375
886435.5625,225446.34375 905270.8125,242278.640625 905270.8125,242278.640625
886435.5625,225446.34375 886435.5625))',26986)) b) As g1 ON sch.gid = g1.gid
    WHERE g1.gid IS NULL AND sch.the_geom && GeomFromText('POLYGON((225446.34375
886435.5625,225446.34375 905270.8125,242278.640625 905270.8125,242278.640625
886435.5625,225446.34375 886435.5625))',26986)
ORDER BY sch.name
```

The old way doing a simple expand distance exclusion query. SOMETIMES THE OLD WAYS ARE JUST BETTER

```
-- Total query runtime: 63 ms. 75 rows retrieved.
SELECT sch.gid as gid_ref, sch.name as sch_name
FROM schools sch LEFT JOIN
(SELECT b.gid FROM schools b , hospitals h
                WHERE b.the_geom && GeomFromText('POLYGON((225446.34375 886435.5625,225446.34375
905270.8125,242278.640625 905270.8125,242278.640625 886435.5625,225446.34375 886435.5625))',26986)
AND h.er_status = 'Y'
                              AND expand(b.the_geom, 3000) && h.the_geom AND distance(b.the_geom,
h.the_geom) <= 3000) g1
ON sch.gid = g1.gid
WHERE g1.gid IS NULL AND sch.the_geom && GeomFromText('POLYGON((225446.34375
886435.5625,225446.34375 905270.8125,242278.640625 905270.8125,242278.640625
886435.5625,225446.34375 886435.5625))',26986)
ORDER BY sch.name
```

Caching data by storing closest neighbor in a table

For this exercise, we will create 2 new fields in our schools table called nearest_hospital and nearest_hospital_gid and update the fields with the closest hospital to that school. We will set our maximum distance search to 100,000 meters carved out into 10 10,000 meter expansions.

```
-- Query returned successfully: 2613 rows affected, 20625 ms (~1/3 minute) execution time.
ALTER TABLE schools ADD nearest_hospital character varying(255);
ALTER TABLE schools ADD nearest_hospital_gid integer;

UPDATE schools SET nearest_hospital = h.name , nearest_hospital_gid = h.gid
FROM (SELECT s.gid As sch_gid,
    (pgis_fn_nn(s.the_geom, 100000, 1,10,'hospitals', 'er_status=''Y''', 'gid', 'the_geom')).*
    FROM schools As s) As nn_emer INNER JOIN hospitals h ON nn_emer.nn_gid = h.gid
WHERE schools.gid = nn_emer.sch_gid
```

Later on we realize we need an alternate hospital in case the first hospital is filled or because of traffic situation we can't get there, so we create a new set of fields to hold the second closest hospital. This particular exercise demonstrates where clause dependent on the additional attribute data of the reference record.

```
-- Query returned successfully: 2613 rows affected, 23032 ms (~1/3 minute) execution time.
ALTER TABLE schools ADD nearest_hospital_2 character varying(255);
ALTER TABLE schools ADD nearest_hospital_gid_2 integer;

UPDATE schools SET nearest_hospital_2 = h.name , nearest_hospital_gid_2 = h.gid
FROM (SELECT s.gid As sch_gid,
    (pgis_fn_nn(s.the_geom, 100000, 1,10,'hospitals', 'er_status=''Y'' AND gid <> ' || CAST(s.
nearest_hospital_gid As varchar(20)) , 'gid', 'the_geom')).*
    FROM schools As s) As nn_emer INNER JOIN hospitals h ON nn_emer.nn_gid = h.gid
WHERE schools.gid = nn_emer.sch_gid AND nearest_hospital_gid IS NOT NULL
```

Part 1 - PostGIS and SharpMap in ASP.NET 2.0 using VB.NET: Compiling SharpMap with PostGIS

What is SharpMap

SharpMap is an opensource freely available mapping engine for the Microsoft.NET 2.0 Framework. It supports numerous datasources such as PostGIS, ESRI Shapefile, MSSQL Spatial, ECW and Oracle spatial. Future support is in the works for MapInfo Tab files and other datasources. Unfortunately as of yet it has not been thoroughly tested in Mono.NET and the SharpMap.UI (desktop portion) is noted to not work in Mono.Net - see notes for further details http://www.codeplex.com/Wiki/View.aspx?ProjectName=SharpMap&title=Can%20I%20use%20SharpMap%20on%20the%20Mono%20framework%20instead. What is especially interesting about SharpMap is that it is a relatively pure .NET implementation and can be used as both a Web mapping as well as a desktop toolkit.

If you are a .NET enthusiast as we are or just want to program in something like VB.NET, this is definitely something to take a look at.

Getting Started

In this exercise we will do what I call *synchronized maps* using PostGIS spatial database engine, SharpMap.Net mapping engine, and data from Boston Private Abandoned Property surveys. *Synchronized Maps* are maps laid out side by side that show the same location and are always in synch with each other. A common example of this is a keymap verses a full blown map where clicking in the key map zooms the main map and clicking on the main map zooms the key map. In our example we will be using maps that are all the same size, but represent different periods in time. I am going to use Visual Basic .NET for this example for a couple of reasons.

- People rarely show VB.NET examples in .NET and focus on C#. This leaves VB.NET and general VBA developers like me feeling *"What about me?"*
- C# doesn't have a **With** clause to my knowledge. How can people live without the convenience of a **With** clause :).
- VB.NET reads more like a spoken language so I think easier to follow and debug.
- VB.NET does a lot of transparent type-casting for you which C# doesn't. For example if you tried to glue together an int and a string, c# will yell (can't convert .. to ..) forcing you to write a mess of type-casting code where as VB.NET will observe that the integer can be converted to a string and do the casting automagically.

Pre-Requisites

- Webserver or workstation with ASP.NET 2.0 installed
- Microsoft.Net 2.0 Framework installed on workstation you will be using to compile
- PostGreSQL 8.0 or above server with PostGIS 1.0 or above installed

Compiling the SharpMap Source

The default binaries of SharpMap available do not come precompiled with PostGIS support, so in this section, we'll go over how to compile your own

Before you can start, you'll need to gather the following items

1. SharpMap.NET source - download the latest version from here, give it a .zip extension if it doesn't default to that when downloading, and unzip it unto your C drive. It should create a SharpMap folder.
2. Copy the PostGIS.cs file from the extracted folder SharpMap/SharpMap.Extensions/Data/Providers to the folder SharpMap/SharpMap/Data/Providers.
3. Download the latest PostgreSQL.NET driver (npgsql) binary for MS 2.0 from http://pgfoundry.org/projects/npgsql. For this I used **Npgsql 1.0-bin-ms2.0.zip**.

At this point, you can either use a development tool such as Visual Studio 2005, the freely available SharpDevelop (http://www.sharpdevelop.com/OpenSource/SD/Default.aspx) or Visual C# Express (http://msdn.microsoft.com/vstudio/express/visualcsharp/) (a free download) and the included solution files to compile the source, or you can just compile with a command line using the .NET framework that you have already installed. I'm just going to describe how to do it with a commandline command since that doesn't assume any additional prerequisites and is simpler to explain.

1. Create a bin folder in your extracted SharpMap/SharpMap folder and copy the Npgsql.* files and Mono.Security.dll from npgsql zip files into this new folder
2. Creat a batch file in the bin folder with the following lines in it

```
set croot=C:\SharpMap\SharpMap\
"%SystemRoot%\microsoft.NET\Framework\v2.0.50727\csc.exe" /t:library /debug:full /out:%croot%bin
\SharpMap.dll /recurse:%croot%*.cs /r:System.web.dll /r:System.data.dll /r:System.Xml.dll /r:
system.dll /r:%croot%bin\Npgsql.dll
pause
```

   Then run the batch script. Running the batch should create 2 files a .dll and a .pdb file. The pdb is used for debugging so it can highlight lines that break in the internal library.
3. Now create an application folder on your webserver complete with a bin folder. Drop the SharpMap.dll, SharpMap.dbg, npgsql.dll, Mono.Security.dll files into the bin folder.

Loading the Test Data

I have packaged in the download file for this exercise, 2 sql scripts one to load neighborhoods, one for abandoned survey data. Below is the

batch script to load all the data. For details on how to create your own load files, read Part 1: Getting Started With PostGIS: An almost Idiot's Guide or OGR2OGR Cheatsheet

Batch Script

```
c:\pgutils\psql -d gisdb -h localhost -U postgres -f neighborhoods.sql
c:\pgutils\psql -d gisdb -h localhost -U postgres -f abansurveys.sql
pause
```

Part 2 - PostGIS and SharpMap in ASP.NET 2.0 using VB.NET: Displaying the Maps

The Fun Part: Mapping the Data

There are 5 key things we need to do when presenting data on web maps. They are

- Plotting the data
- Maintaining view state - where was the user last - if this is there first visit - initialize the state.
- Figuring out where a user clicked on an image map and converting these coordinates to spatial coordinates relative to view state.
- Taking additional inputs such as request for specific layers or filtering of layers.
- Doing something with these coordinates and inputs - e.g. do we Zoom in, Zoom Out, Pan, filter data

Everthing beyond those 5 basic items is presentational sugar.

Initializing Properties

For this exercise, we will use arrays to keep track of the 4 maps we have. This makes it easy to extend for more maps.

```
  Protected justmaps_string() As String = {"2002", "2003", "2004", "2005"}
  Protected justmaps_maps(3) As SharpMap.Map 'its a zeroth based array
```

The Presentation Layer

The key parts of our aspx is shown below. Note how we have named the image controls to be in line with our array variables. See the Demo in action

```
      <table>
          <tr>
              <td nowrap colspan="2">
                  <asp:RadioButtonList ID="rblMapTools" runat="server" RepeatDirection="Horizontal">
                      <asp:ListItem Value="0">Zoom in</asp:ListItem>
                      <asp:ListItem Value="1">Zoom out</asp:ListItem>
                      <asp:ListItem Value="2" Selected="True">Pan</asp:ListItem>
                  </asp:RadioButtonList>
                  <asp:Button ID="cmdReset" Text="Reset" runat="server"/>
              </td>
           </tr>
           <tr>
              <td nowrap>Unit Type:
                  <asp:DropDownList ID="ddlLU" runat="server" AutoPostBack="true">
                      <asp:ListItem Value="ALL">ALL</asp:ListItem>
                      <asp:ListItem Value="A">A</asp:ListItem>
                      <asp:ListItem Value="C">C</asp:ListItem>
                      <asp:ListItem Value="CM">CM</asp:ListItem>
                      <asp:ListItem Value="I">I</asp:ListItem>
                      <asp:ListItem Value="R1">R1</asp:ListItem>
                      <asp:ListItem Value="R2">R2</asp:ListItem>
                      <asp:ListItem Value="R3">R3</asp:ListItem>
                      <asp:ListItem Value="RC">RC</asp:ListItem>
                  </asp:DropDownList>
              </td>
              <td>
                  Abandoned Type:
                  <asp:DropDownList ID="ddlAbandtype" runat="server" AutoPostBack="true">
                      <asp:ListItem Value="ALL">ALL</asp:ListItem>
                      <asp:ListItem Value="A">Abandoned</asp:ListItem>
                      <asp:ListItem Value="B">Burned</asp:ListItem>
                      <asp:ListItem Value="D">Boarded</asp:ListItem>
                  </asp:DropDownList>
             </td>
          </tr>
          <tr>
              <td><b>Survey 2002</b><br />
                      <asp:ImageButton Width="300" Height="300" ID="imgMap2002" runat="server" style="border: 1px solid
#000;" />
              </td>
              <td>
                  <b>Survey 2003</b><br />
```

```
                    <asp:ImageButton Width="300" Height="300" ID="imgMap2003" runat="server" style="border: 1px solid
#000;" />
                </td>
            </tr>
            <tr>
                <td>
                    <b>Survey 2004</b><br />
                    <asp:ImageButton Width="300" Height="300" ID="imgMap2004" runat="server" style="border: 1px solid
#000;" />
                </td>
                <td>
                    <b>Survey 2005</b><br />
                    <asp:ImageButton Width="300" Height="300" ID="imgMap2005" runat="server" style="border: 1px solid
#000;" />
                </td>
            </tr>
        </table>
```

Initializing the Data

In ASP.NET the basic state is initialized in the Page Load event. Below is what our page load looks like

```
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
        Dim i As Integer
        Dim img As System.Web.UI.WebControls.Image

        For i = 0 To justmaps_string.GetUpperBound(0)
            img = Me.FindControl("imgMap" & justmaps_string(i))
            justmaps_maps(i) = Me.InitializeMap(New System.Drawing.Size(img.Width.Value, img.Height.Value), justmaps_string
(i))
        Next

        If Page.IsPostBack Then
            'Page is post back. Restore center and zoom-values from viewstate
            For i = 0 To justmaps_string.GetUpperBound(0)
                justmaps_maps(i).Center = ViewState("mapCenter")
                justmaps_maps(i).Zoom = ViewState("mapZoom")
            Next
        Else
            'Save the current mapcenter and zoom in the viewstate by just picking the first map
            ViewState.Add("mapCenter", justmaps_maps(0).Center)
            ViewState.Add("mapZoom", justmaps_maps(0).Zoom)
            For i = 0 To justmaps_string.GetUpperBound(0)
                GenerateMap(justmaps_maps(i), justmaps_string(i))
            Next
        End If
    End Sub
```

Observe in the above, we create an instance of a sharpmap object to correspond with each .NET Image control. Below is what our initialize map code looks like.

```
    Public Function InitializeMap(ByVal size As System.Drawing.Size, ByVal maptype As String) As SharpMap.Map
        HttpContext.Current.Trace.Write("Initializing map...")
        'Initialize a new map of size 'imagesize'
        Dim map As SharpMap.Map = New SharpMap.Map(size)
        Dim ConnStr As String = ConfigurationManager.AppSettings("DSN")

        Dim dtItem As New SharpMap.Data.Providers.PostGIS(ConnStr, "abansurveys", "the_pointft")
        Dim dtNeighborhoods As New SharpMap.Data.Providers.PostGIS(ConnStr, "neighborhoods", "the_geom")
        Dim strWhere As String = "(yr = " & maptype & " AND abandtype <> 'N') "

        Dim lyrNeighborhoods As SharpMap.Layers.VectorLayer = New SharpMap.Layers.VectorLayer("Neighborhoods")
        Dim lyrNeighborhoodNames As SharpMap.Layers.LabelLayer = New SharpMap.Layers.LabelLayer("NeighborhoodNames")
        Dim lyrItem As SharpMap.Layers.VectorLayer = New SharpMap.Layers.VectorLayer("Item")

        If Not IsNumeric(maptype) Then 'map type should be an integer otherwise the request is a potential hack
            Return map
        End If
        With lyrNeighborhoods
            .DataSource = dtNeighborhoods
            .Style.Fill = Brushes.White
            .Style.Outline = System.Drawing.Pens.Black
            .Style.EnableOutline = True
```

```
        End With
        map.Layers.Add(lyrNeighborhoods)

        If Me.ddlLU.SelectedValue <> "ALL" Then
            strWhere &= "AND lu = '" & Me.ddlLU.SelectedValue & "' "
        End If

        If Me.ddlAbandtype.SelectedValue <> "ALL" Then
            strWhere &= "AND abandtype = '" & Me.ddlAbandtype.SelectedValue & "' "
        End If

        dtItem.DefinitionQuery = strWhere

        With lyrItem
            .DataSource = dtItem
            .Enabled = True
            With .Style
                .SymbolScale = 0.8F
            End With
        End With
        map.Layers.Add(lyrItem)

        With lyrNeighborhoodNames
            .DataSource = dtNeighborhoods
            .Enabled = True
            .LabelColumn = "name"
            .Style = New SharpMap.Styles.LabelStyle
            With .Style
                .ForeColor = Color.Black
                .Font = New Font(FontFamily.GenericMonospace, 11, FontStyle.Bold)
                .HorizontalAlignment = SharpMap.Styles.LabelStyle.HorizontalAlignmentEnum.Center
                .VerticalAlignment = SharpMap.Styles.LabelStyle.VerticalAlignmentEnum.Middle
                .Halo = New Pen(Color.Yellow, 2)
                .CollisionBuffer = New System.Drawing.SizeF(30, 30)
                .CollisionDetection = True
            End With
            .TextRenderingHint = Text.TextRenderingHint.AntiAlias
        End With
        map.Layers.Add(lyrNeighborhoodNames)

        map.BackColor = Color.White
        map.ZoomToExtents()
        HttpContext.Current.Trace.Write("Map initialized")
        Return map
    End Function
```

In the page load for each map we call a GenerateMap which renders the image of the map to the browser. The code looks like the below

```
    '<summary>
    'Grabs the map corresponding to a given image control, inserts it into the cache and sets the Image control Url to the
cached image
    '</summary>
    Private Sub GenerateMap(ByVal aMap As SharpMap.Map, ByVal maptype As String)
        Dim img As System.Drawing.Image = aMap.GetMap()
        Dim imgID As String = SharpMap.Web.Caching.InsertIntoCache(1, img)
        CType(Me.FindControl("imgMap" & maptype), System.Web.UI.WebControls.Image).ImageUrl = "Getmap.aspx?ID=" +
HttpUtility.UrlEncode(imgID)
    End Sub
```

Handling user requests

In this particular example, we've got a couple of actions that a user can perform.

1. Zoom In
2. Zoom Out
3. Pan
4. Filter by Unit Type
5. Filter by Abandoned Type

To handle the first 3 actions, we define a single event handler that covers clicking on any of the maps. That code looks like

```
    Protected Sub imgMap_Click(ByVal sender As Object, ByVal e As System.Web.UI.ImageClickEventArgs) Handles imgMap2002.
Click, imgMap2003.Click, imgMap2004.Click, imgMap2005.Click
        Dim i As Integer
        '-- Set center of the map to where the client clicked
        For i = 0 To justmaps_string.GetUpperBound(0)
            justmaps_maps(i).Center = justmaps_maps(i).ImageToWorld(New System.Drawing.Point(e.X, e.Y))
            '-- Set zoom value if any of the zoom tools were selected
            If rblMapTools.SelectedValue = "0" Then '//Zoom in
                justmaps_maps(i).Zoom = justmaps_maps(i).Zoom * 0.5
            ElseIf rblMapTools.SelectedValue = "1" Then '//Zoom out
                justmaps_maps(i).Zoom = justmaps_maps(i).Zoom * 2
            End If
            '--//Save the new map's zoom and center in the viewstate
            ViewState.Add("mapCenter", justmaps_maps(i).Center)
            ViewState.Add("mapZoom", justmaps_maps(i).Zoom)
            '--//Create the map
            GenerateMap(justmaps_maps(i), justmaps_string(i))
        Next
    End Sub
```

To handle query filter requests, again we use a single event handler. Note that most of the logic for filtering is in our InitializeMap routine.

```
    Protected Sub ddl_SelectedIndexChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles ddlLU.
SelectedIndexChanged, ddlAbandtype.SelectedIndexChanged
        Dim i As Integer
        For i = 0 To justmaps_string.GetUpperBound(0)
            '-- Initialize the zoom and center to what user had last
            justmaps_maps(i).Zoom = ViewState("mapZoom")
            justmaps_maps(i).Center = ViewState("mapCenter")
            '--//Create the map
            GenerateMap(justmaps_maps(i), justmaps_string(i))
        Next
    End Sub
```

There are other ways of tying the event handler to a presentation object. We could have just as easily and perhaps more extensibly defined an onclick event for each of our controls which would have looked something like

```
<asp:ImageButton Width="300" Height="300" ID="imgMap2002" runat="server" style="border: 1px solid #000;"
onclick="imgMap_Click"/>
```

The benefit of that approach over the one we chose is that it would be easier to implement a dynamic number of maps say if you have a map for each data item in a datalist.

download

Part 1: Getting Started With PostGIS: An almost Idiot's Guide

What Is PostGIS?

PostGIS is an opensource, freely available , and fairly OGC compliant spatial database extender for the PostGreSQL Database Management System. In a nutshell it adds spatial functions such as distance, area, and specialty geometry data types to the database. PostGIS is very similar in functionality to ESRI ArcSDE, Oracle Spatial, and DB II spatial extender. The latest release version now comes packaged with the PostgreSQL DBMS installs as an optional add-on.

We will assume a windows environment for this tutorial, but most of the tutorial will apply to other supported platforms such as Linux, Unix, BSD, Mac etc. We will also be using Massachusetts/Boston data for these examples.

Installing PostgreSQL with PostGIS Functionality

We will not go into too much detail here since the install wizard (at least the windows one) is pretty good. Below are the basic steps.

1. Download the install for your specific platform from the PostGreSQL Binary Download (http://www.postgresql.org/ftp/binary/) . As of this writing the latest version is 8.2.4. The below options follow the basic sequence of the postgresql windows installer.
2. Make sure to check the PostGIS Spatial Extensions option, PgAdmin III, psql, option
3. If you want to access this server from other than the server itself. Check the "Accept connection on all addresses, not just localhost". NOTE: You can change this later by editing the postgresql.conf -> listen_addresses property and if you don't like the default port of 5432 you can change this as well in the postgresql.conf -> port property.
4. For encoding UTF-8 is preferred because you can convert to other encodings. SQL_ASCII is generally the default on Windows because UTF-8 in previous versions was not supported well for Windows.
5. For language make sure to check PL/pgsql. If you forget, you can always use the createlang plpgsql command to install in a specific database.
6. Enable contrib - you should check Adminpack at the very least. This simplifies adminstrative management of the server via PgAdmin III.
7. From experience it is best not to install PostGIS in the master template database unless you really want it included in every database you create moving forward and want it in exactly the same way. The reason for this is that it does add a lot of functions, and you may not want all that clutter in every database you create especially if you are not using it just for spatial databases. Also for specific databases, you may choose to have the functions in a different schema. **Please note for many install packages - particularly windows. When you choose PostGIS as an option, the system will create a template_postgis template database for you that has PostGIS functions included.**

Creating a spatial database

PostGreSQL comes packaged with a fairly decent admin tool called PgAdmin3. If you are a newbie, its best just to use that tool to create a new database.

- On windows PgAdmin III is under Start->Programs->PostGreSQL 8.2->PgAdmin III
- Login with the super user usually postgres and the password you chose during install. If you forgot it, then go into pg_hba.conf (just open it with an editor such as notepad or a programmer editor). Set the line

```
host all all 127.0.0.1/32 md5
to
host all all 127.0.0.1/32 trust
```

This will allow any person logging locally to the computer that PostGreSQL is installed on to access all databases without a password. (127.0.0.1/32) means localhost only (32 is the bit mask). Note you can add additional lines to this file or remove lines to allow or block certain ip ranges. The lines highest in the file take precedence.

So for example if you wanted to allow all users logging in access as long as they successfully authenticate with an md5 password, then you can add the line

```
host all all 0.0.0.0/0 md5
```

. If it is below, you will still be able to connect locally without a password but non-local connections will need a valid username and password.

Note: - The newer versions of PgAdmin III (1.4 something on) allow editing Postgresql.conf and pg_hba.conf using the PgAdmin III tool. These are accessible from Tools->Server Configuration and provide a fairly nice table editor to work with.

- Now for the fun part - Create your database. Call it gisdb or whatever you want. In newer versions of PostgreSQL, if you chose PostGIS in the first part, there is a template database called **template_postgis**. Chose this as a template.
- Its generally a good idea to create a user too that owns the database that way you don't need to use your superuser account to access it.
- UPDATE: - The remaining steps in this section are not needed if you chose **template_postgis** for your new database. However if you are trying to spatially enable an existing database or you didn't get the template_postgis option. Do the remaining steps. Next go to tools->Query tool in pgAdmin III and browse to the postgresql install contrib lwpostgis.sql file (on Windows the default install is Program files\Postgresql\8.2\share\contrib\lwpostgis.sql
- On the Query tool, make sure you gisdb is selected and then click the green arrow. You'll get a bunch of notices - not to be alarmed.
- Next open up the spatial_ref_sys.sql (on windows Program files\Postgresql\8.1\share\contrib\spatial_ref_sys.sql. Click the green arrow again. This step loads thousands of spatial reference system records which is used by PostGIS Projection library to transform from one spatial

coordinate system to another.

Loading GIS Data Into the Database

Now we have a nice fully functional GIS database with no spatial data. So to do some neat stuff, we need to get some data to play with.

Get the Data

Download data from the MassGIS site.
For this simple exercise just download Towns with Coast
Extract the file into some folder. We will only be using the _POLY files for this exercise.

NOTE: Someone asked how you extract the file if you are on a linux box.

---FOR LINUX USERS ---

If you are on Linux/Unix, I find the exercise even easier. If you are on linux or have Wget handy - you can do the below to download the file after you have cded into the folder you want to put it in.

```
wget ftp://data.massgis.state.ma.us/pub/shape/state/towns.exe
```

Now to extract it simply do the following from a shell prompt

```
unzip towns.exe
```

---END FOR LINUX USERS ---

Figure out SRID of the data

You will notice one of the files it extracts is called TOWNS_POLY.prj. A .prj is often included with ESRI shape files and tells you the projection of the data. We'll need to match this descriptive projection to an SRID (the id field of a spatial ref record in the spatial_ref_sys table) if we ever want to reproject our data.

- Open up the .prj file in a text editor. You'll see something like *NAD_1983_StatePlane_Massachusetts_Mainland_FIPS_2001* and *UNIT ["Meter",1.0]*
- Open up your PgAdmin III query tool and type in the following statement
  ```
  select srid, srtext, proj4text from spatial_ref_sys where srtext ILIKE '%Massachusetts%'
  ```
  And then click the green arrow. This will bring up about 10 records.
- Note the srid of the closest match. In this case its **26986**. NOTE: srid is not just a PostGIS term. It is an OGC standard so you will see SRID mentioned a lot in other spatial databases, gis webservices and applications. Most of the common spatial reference systems have globally defined numbers. So 26986 always maps to NAD83_StatePlane_Massachusetts_Mainland_FIPS_2001 Meters. Most if not all MassGIS data is in this particular projection.

Loading the Data

The easiest data to load into PostGIS is ESRI shape data since PostGIS comes packaged with a nice command line tool called shp2pgsql which converts ESRI shape files into PostGIS specific SQL statements that can then be loaded into a PostGIS database.

This file is located in the PostGresql bin folder which default location in Windows is *Program Files/PostGreSQL/8.2/bin*

Make a PostGIS mini toolkit

Since these files are so embedded, it is a bit annoying to navigate to. To create yourself a self-contained toolkit you can carry with you anywhere, copy the following files from the bin folder into say c:\pgutils:

```
 comerr32.dll krb5_32.dll libeay32.dll
libiconv-2.dll libintl-2.dll libpq.dll pgsql2shp.exe psql.exe
pg_dump.exe pg_restore.exe shp2pgsql.exe ssleay32.dll
```

Load Towns data

- Open up a command prompt.
- Cd to the folder you extracted the towns data

- Run the following command:
```
c:\pgutils\shp2pgsql -s 26986 TOWNS_POLY towns > towns.sql
```
- Load into the database with this command:
```
psql -d gisdb -h localhost -U postgres -f towns.sql
```

If you are on another machine different from the server, you will need to change localhost to the name of the server. Also you may get prompted for a password. For the above I used the default superuser postgres account, but its best to use a non-super user account.

Indexing the data

Table indexes are very important for speeding up the processing of most queries. There is also a downside to indexes and they are the following

1. Indexes slow down the updating of indexed fields.
2. Indexes take up space. You can think of an index as another table with bookmarks to the first similar to an index to a book.

Given the above, it is often times tricky to have a good balance. There are a couple general rules of thumb to go by that will help you a long way.

1. Never put indexes on fields that you will not use as part of a where condition or join condition.
2. Be cautious when putting index fields on heavily updated fields. For example if you have a field that is frequently updated and is frequently used for updating, you'll need to do benchmark tests to make sure the index does not cause more damage in update situations than it does for select query situations. In general if the number of records you are updating at any one time for a particular field is small, its safe to put in an index.
3. Corrollary to 2. For bulk uploads of a table - e.g. if you are loading a table from a shape, its best to put the indexes in place after the data load because if an index is in place, the system will be creating indexes as its loading which could slow things down considerably.
4. If you know a certain field is unique in a table, it is best to use a unique or primary index. The reason for this is that it tells the planner that once its found a match, there is no need to look for another. It also prevents someone from accidentally inserting a duplicate record as it will throw an error.
5. For spatial indexes - use a gist index. A gist basically stores the bounding box of the geometry as the index. For large complex geometries unfortunately, this is not too terribly useful.

The most common queries we will be doing on this query are spatial queries and queries by the town field. So we will create 2 indexes on these fields.

```
CREATE INDEX idx_towns_the_geom
ON towns
USING gist(the_geom);


CREATE INDEX idx_towns_town
ON towns
USING btree(town);
```

Querying Data

Go back into PgAdmin III and refresh your view. Verify that you have a towns database now.

Test out the following queries from the query tool

```
select extent(the_geom) from towns where town = 'BOSTON';
select area(geomunion(the_geom)) from towns where town = 'BOSTON';
```

Viewing the Data

If you are a GIS newbie, I highly recommend using Quantum GIS. Quantum GIS has ability to view PostGIS data directly, do simple filters on it, is free, is cross-platform (Linux, and Windows and some others) and is the least threatening of all the GIS Viewers I have seen out there for people new to GIS.

Part 2: Introduction to Spatial Queries and SFSQL

What is SFSQL?

One of the greatest things about Spatial Relational Databases is that they bring GIS to a new level by allowing you to apply the expressive SQL declarative language to the spatial domain. With spatial relational databases, you can easily answer questions such as what is the average household income of a neighborhood block. What political district does X reside in. This new animal that marries SQL with GIS is called Simple Features for SQL (SFSQL). In essence SFSQL introduces a new set of functions and aggregate functions to the SQL Language.

Some Common Queries

In this next section we'll go over some common queries utilizing the data we downloaded in Part 1. Although we are using PostGIS for this exercise, most Spatial Relational Databases such as Oracle Spatial, ArcSDE, DB Spatial Extender, have similar syntax.

Transforming from One Coordinate System to Another

NAD 83 Meters to NAD 83 Ft

As noted in Part 1, the data we downloaded is in NAD 83 Meters MA State Plane. What if we needed the data in NAD 83 feet. To accomplish this, we would need to transform our spatial data to the new coordinate system with the transform function. If you look in the spatial_ref_sys table you'll notice that srid = 2249 is the srid of the Nad 83 ft MA State Plane. Our query would look something like this.

```
SELECT town, transform(the_geom, 2249) as the_geom_nad83ft FROM towns
```

Getting Latitude and Longitude Centroid

In order to get the latitude and longitude of our data, we need our coordinate reference system to be in some for of longlat. To begin with, we first transform our data from NAD 83 Meters Massachusetts State Plane to some variant of longlat - closest match is NAD 83 North American Datum (srid = 4269). Then we find the centroid and then the x and y coordinates of that.

```
SELECT town, x(centroid(transform(the_geom, 4269))) as longitude,
y(centroid(transform(the_geom, 4269))) as latitude
FROM towns
```

Aggregate Functions

Spatial aggregate functions are much like regular SQL aggregate functions such as AVG, SUM, COUNT in that they work with GROUP BY and HAVING predicates and collapse multiple records into a single record. If you are unfamiliar with the above terms - take a look at Summarizing data with SQL (Structured Query Language)

Extent

The extent function is an aggregate function that gives you the bounding box of a set of geometries. It is especially useful for determining the bounding rectangle of the area you are trying to map. For example if we wanted to find the bounding box of the boston area in NAD 83 feet, we would do something like this.

```
SELECT town, extent(transform(the_geom, 2249)) as the_extent FROM towns WHERE town = 'BOSTON'
GROUP BY town
```

GeomUnion

The geomunion function is an aggregate function that takes a set of geometries and unions them together to create a single geometry field. For our towns data, we have multiple records per town. To get a single geometry that represents the total region of a town, we could use the geomunion function like the example below.

```
select town,  geomunion(the_geom) as thegeom  from towns group by town;
```

It is important to note that while the above query will give you one record per town. Our original plain vanilla of

```
select town, the_geom as thegeom from towns;
```

will give you multiple records per town.

Seeing Results Visually

To get a visual sense of what all these different queries look like, you can dump out the above outputs as an ESRI shape file using the pgsql2shp tool and view it using a shape viewer such as ESRI's freely available ArcExplorer.

```
pgsql2shp -f myshp -h myserver -u apguser -P apgpassword -g thegeom mygisdb "select town,
geomunion(the_geom) as thegeom from towns group by town"
```

One caveat: the shape dumper utility can only dump out fields of type geometry. So for example to dump out a bbox type such as what is returned by the **extent** function, you'll need to cast the output as a geometry something like

```
SELECT town, extent(transform(the_geom, 2249))::geometry as theextent
FROM towns
WHERE town = 'BOSTON' GROUP BY town
```

Distance Queries

Getting Minimal Distance using Distance function

One common question asked for of a spatial database is how far one object is from another. PostGIS like similar spatial databases has a function called **distance** which will tell you the minimum distance between one geometry from another.

An example of such a use is below. The below query will tell you **How far each zip in your zipcode table is from another zip 02109?**. In this case we are comparing the field the_geom_meter which we have in NAD 83 State Plane meters. Distance is always measured in the metric of your spatial reference system.

```
SELECT z.zipcode As zip1, z2.zipcode As zip2, distance(z.the_geom_meter,z2.the_geom_meter) As
thedistance FROM zipcode z, zipcode z2 WHERE z2.zipcode = '02109'
```

If the above geometries were polygons, then we would be getting the minimal distance from any point on each polygon to the other. For point geometries there is no distinction between a min distance and other distances.

Getting Average Distance using the Centroid function

If we are not dealing with just points, then there are all kinds of distances we could be asking for. For example If we wanted the average distance between 2 polygons, then we would want the distance between the centroids. An average distance compare would then look like.

```
SELECT z.zipcode As zip1, z2.zipcode As zip2, distance(centroid(z.the_geom_meter),centroid(z2.
the_geom_meter)) As averagedistance FROM zipcode z, zipcode z2 WHERE z2.zipcode = '02109'
```

Getting a list of objects that are within X distance from another object

An often asked question is what zipcodes are within x away from my zip?. To do that one would think that simply doing a distance < x would suffice. This would suffice except it would be slow since it is not taking advantage of PostGIS indexes and also because if the geometries you are comparing are fairly complex, the search could be exhaustive. To write the query in the most efficient way, we would include the use of the **Expand** function. Like so

```
SELECT z.zipcode FROM zipcode z, zipcode z2 WHERE z2.zipcode = '02109' AND expand(z2.
the_geom_meter, 3000) && z.the_geom_meter AND distance(z.the_geom, z2.the_geom) <= 3000
```

The Expand function takes the bounding box of a geometry and expands it out X in all directions. So in this case we would be expanding our 02109 zip geometry out 3000 meters.

The && operator is the Overlaps operator that returns true if two geometries bounding boxes overlap and false if they do not. It is a much faster especially if you have indexes on your geometries because it utilizes the bounding box index.

The below may seem a little redundant because not doing the expand && would give you the same answer, but what the expand && call does is limits the number of records that need to be inspected by the more exhaustive distance function.

Part 3: PostGIS Loading Data from Non-Spatial Sources

Often you will receive data in a non-spatial form such as comma delimited data with latitude and longitude fields. To take full advantage of PostGIS spatial abilities, you will want to create geometry fields in your new table and update that field using the longitude latitude fields you have available.

General Note: All the command statements that follow should be run from the **PgAdminIII Tools - Query Tool** or any other PostGreSQL Administrative tool you have available. If you are a command line freak - you can use the **psql** command line tool packaged with PostGreSQL.

Getting the data

For this exercise, we will use US zip code tabulation areas instead of just Boston data. The techniques here will apply to any data you get actually.

First step is to download the data from US Census. http://www.census.gov/geo/www/gazetteer/places2k.html

Importing the Data into PostGreSQL

PostGreSQL comes with a COPY function that allows you to import data from a delimited text file. Since the ZCTAs data is provided in fixed-width format, we can't import it easily without first converting it to a delimited such as the default tab-delimited format that COPY works with. Similarly for data in other formats such as DBF, you'll either want to convert it to delimited using tools such as excel, use a third party tool that will import from one format to another, or one of my favorite tools Microsoft Access that allows you to link any tables or do a straight import and export to any ODBC compliant database such as PostGreSQL.

Create the table to import to

First you will need to create the table in Postgres. You want to make sure the order of the fields is in the same order as the data you are importing.

```
CREATE TABLE zctas
(
  state char(2),
  zcta char(5),
  junk varchar(100),
  population_tot int8,
  housing_tot int8,
  water_area_meter float8,
  land_area_meter float8,
  water_area_mile float8,
  land_area_mile float8,
  latitude float8,
  longitude float8
)
WITHOUT OIDS;
```

Convert from Fixed-width to Tab-Delimited

For this part of the exercise, I'm going to use Microsoft Excel because it has a nice wizard for dealing with fixed-width and a lot of windows users have it already. If you open the zcta file in Excel, it should launch the Text Import Wizard. MS Access has a similarly nice wizard and can deal with files larger than excels 65000 some odd limitation. Note there are trillions of ways to do this step so I'm not going to bother going over the other ways. For non-MS Office users other office suites such as Open-Office probably have similar functionality.

1. Open the file in Excel.
2. Import Text Wizard should launch automatically and have Fixed-Width as an option
3. Look at the ZCTA table layout spec http://www.census.gov/geo/www/gazetteer/places2k.html#zcta and set your breakouts the same as specified. **For the above I broke out the Name field further into first 5 for zcta and the rest for a junk field.**
4. Next File->Save As ->Text (Tab delimited)(*.txt) -give it name of zcta5.tab
5. Copy the file to somewhere on your PostGreSQL server.

The COPY command

Now copy the data into the table using the COPY command. **Note the Copy command works using the PostGreSQL service so the file location must be specified relative to the Server.**

```
COPY zctas FROM 'C:/Downloads/GISData/zcta5.tab';
```

Creating and Populating the Geometry Field

Create the Geometry Field

To create the Geometry field, use the AddGeometryColumn opengis function. This will add a geometry field to the specified table as well as adding a record to the geometry_columns meta table and creating useful constraints on the new field. A summary of the function can be found here http://postgis.refractions.net/docs/ch06.html#id2526109.

```
SELECT AddGeometryColumn( 'public', 'zctas', 'thepoint_lonlat', 4269, 'POINT', 2 );
```

The above code will create a geometry column named *thepoint_longlat* in the table zctas that validates to make sure the inputs are 2-dimensional points in SRID 4269 (NAD83 longlat).

Populate the Geometry Field using the Longitude and Latitude fields

```
UPDATE zctas
       SET thepoint_lonlat = PointFromText('POINT(' || longitude || ' ' || latitude ||
')',4269)
```

The above code will generate a Text representation of a point and convert this representation to a PostGis geometry object of spatial reference SRID 4269.

There are a couple of things I would like to point out that may not be apparently clear to people not familiar with PostGreSQL or PostGis

- || is a string concatenator. It is actually the ANSI-standard way of concatenating strings together. In MySQL you would do this using the CONCAT function and in Microsoft SQL Server you would use +. Oracle also uses ||. So what the inner part of the code would do is to generate something that looks like **POINT(-97.014256 38.959448)**.
- You can't just put any arbitrary SRID in there and expect the system to magically transform to that. The SRID you specify has to be the reference system that your text representation is in.

Transforming to Another spatial reference system

The above is great if you want your geometry in longlat spatial reference system. In many cases, longlat is not terribly useful. For example if you want to do distance queries with your data, you don't want your distance returned back in longlat. You want it in a metric that you normally measure things in.

In the code below, we will create a new geometry field that holds points in the **WGS 84 North Meter** reference system and then updates that field accordingly.

```
SELECT AddGeometryColumn( 'public', 'zctas', 'thepoint_meter', 32661, 'POINT', 2 );

UPDATE zctas SET thepoint_meter = transform(setsrid(makepoint(longitude, latitude),4269), 32661);
```

Please note in earlier versions of this article I had suggested doing the above with

```
UPDATE zctas
SET thepoint_meter = transform(PointFromText('POINT(' || longitude || ' ' || latitude ||
')',4269),32661) ;
```

On further experimentation, it turns out that PointFromText is perhaps the slowest way of doing this in Postgis. Using a combination of setsrid and makepoint is on the magnitude of 7 to 10 times faster at least for versions of PostGIS 1.2 and above. GeomFromText comes in second (replace PointFromText with GeomFromText) at about 3 to 1.5 times slower than setsrid makepoint. See about PointFromText on why PointFromText is probably slower. In GeomFromText, there appears to be some caching effects so on first run with similar datasets GeomFromText starts out about 3-4 times slower than the makepoint way and then catches up to 1.5 times slower. This I tested on a dataset of about 150,000 records and all took - 26 secs for PointFromText fairly consistently, 10.7 secs for first run of GeomFromText then 4.1 secs for each consecutive run, 3.5 secs fairly consistently for setsrid,makepoint on a dual Xeon 2.8 Ghz, Windows 2003 32-bit with 2 gig RAM).

Index your spatial fields

One of the number one reasons for poor query performance is lack of attention to indexes. Putting in an index can make as much as a 100 fold difference in query speed depending on how many records you have in the table. For large updates and imports, you should put your indexes in after the load, because while indexes help query speed, updates against indexed fields can be very slow because they need to create index records for the updated/inserted data. In the below, we will be putting in GIST indexes against our spatial fields.

```
CREATE INDEX idx_zctas_thepoint_lonlat ON zctas
  USING GIST (thepoint_lonlat);
```

```
CREATE INDEX idx_zctas_thepoint_meter ON zctas
  USING GIST (thepoint_meter);

ALTER TABLE zctas ALTER COLUMN thepoint_meter SET NOT NULL;
CLUSTER idx_zctas_thepoint_meter ON zctas;

VACUUM ANALYZE zctas;
```

In the above after we create the indexes, we put in a constraint to not allow nulls in the **thepoint_meter** field. The not null constraint is required for clustering since as of now, clustering is not allowed on gist indexes that have null values. Next we cluster on this index. Clustering basically physically reorders the table in the order of the index. In general spatial queries are much slower than attribute based queries, so if you do a fair amount of spatial queries especially bounding box queries, you get a significant gain.

Please note: (for those familiar with Cluster in SQL Server - Cluster in PostgreSQL pretty much means the same thing, except in PostgreSQL - at least for versions below 8.3 (not sure about 8.3 and above), if you cluster and then add new data or update data, the system does not ensure that your new or updated data is ordered according to the cluster as it does in SQL Server. The upside of this is that there is no additional penalty in production mode with clustering as you have in SQL Server. The downside is you must create a job of some sort to maintain the cluster for tables that are constantly updated.

In the above we vacuum analyze the table to insure that index statistics are updated for our table.

PLR Part 1: Up and Running with PL/R (PLR) in PostgreSQL: An almost Idiot's Guide

R is both a language as well as an environment for doing statistical analysis. R is available as Free Software under the GPL. For those familiar with environments such as S, MatLab, and SAS - R serves the same purpose. It has powerful constructs for manipulating arrays, packages for importing data from various datasources such as relational databases, csv, dbf files and spreadsheets. In addition it has an environment for doing graphical plotting and outputting the results to both screen, printer and file.

For more information on R check out R Project for Statistical Computing.

What is PL/R?

PL/R is a PostGreSQL language extension that allows you to write PostgreSQL functions and aggregate functions in the R statistical computing language.

With the R-language you can write such things as aggregate function for **median** which doesn't exist natively in PostgreSQL and exists only in a few relational databases natively (e.g. Oracle) I can think of. Even in Oracle the function didn't appear until version 10.

Another popular use of R is for doing Voronoi diagrams using the R Delaunay Triangulation and Dirichlet (Voronoi) Tesselation (deldir) Comprehensive R Archive Network (CRAN) package. When you combine this with PostGIS you have an extremely powerful environment for doing such things as nearest neighbor searches and facility planning.

In the past, PL/R was only supported on PostgreSQL Unix/Linux/Mac OSX environments. Recently that has changed and now PLR can be run on PostgreSQL windows installs. For most of this exercise we will focus on the Windows installs, but will provide links for instructions on Linux/Unix/Mac OSX installs.

Installing R and PL/R

In order to use PLR, you must first have the R-Language environment installed on the server you have PostgreSQL on. In the next couple of sections, we'll provide step by step instructions.

Installing PostgreSQL and PostGIS

It goes without saying. If you don't have PostgreSQL already - please install it and preferably with PostGIS support. Checkout Getting started with PostGIS

Installing R

1. Next install R-Language:
    1. Pick a CRAN Mirror from http://cran.r-project.org/mirrors.html
    2. In Download and Install R section - pick your OS from the Precompiled binary section. In my case I am picking Windows (95 and later). Note there are binary installs for Linux, MacOS X and Windows.
    3. If you are given a choice between base and contrib. Choose base. This will give you an install containing the base R packages. Once you are up and running with R, you can get additional packages by using the builit in package installer in R or downloading from the web which we will do later.
    4. Run the install package. As of this writing the latest version of R is 2.5. The windows install file is named R-2.5.0-win32.exe
2. Once you have installed the package - open up the RGUI. NOTE: For windows users - this is located on Start menu - Start -> Programs - >R -> R. 2.5.0. If for some reason you don't find it on the start menu - it should be located at **"C:\Program Files\R\R-2.5.0\bin\Rgui.exe"**.
3. Run the following command at the R GUI Console.
```
update.packages()
```
4. Running the above command should popup a dialog requesting for a CRAN MIRROR - pick one closest to you and then click **OK**.
5. A sequence of prompts will then follow requesting if you would like to replace existing packages. Go ahead and type **y** to each one. After that you will be running the latest version of the currently installed packages.

Installing PL/R

Now that you have both PostgreSQL and R installed, you are now ready to install PLR procedural language for PostgreSQL.

1. Go to http://www.joeconway.com/plr/
2. For non-Windows users, follow the instructions here http://www.joeconway.com/plr/doc/plr-install.html.

   For Windows users:
    1. download the installation file from here http://www.davidgis.fr/download/plr-8.2.0.4_0_win32.exe
    2. Run the install. Accept the defaults.
    3. Restart your computer

Loading PL/R functionality into a database

In order to start using PL/R in a database, you need to load the help functions in the database. To do so do the following.

1. Using PgAdmin III - select the database you want to enable with PL/R and then click the **SQL** icon to get to the query window.
2. Choose -> File -> Open -> path/to/PostgreSQL/8.2/contrib/plr.sql (NOTE: on Windows the default location is C:\Program Files\PostgreSQL

\8.2\contrib\plr.sql

3. Click the Green arrow to execute

Testing out PL/R

Next run the following commands from PgAdminIII or psql to test out R

```
SELECT * FROM plr_environ();
SELECT load_r_typenames();
SELECT * FROM r_typenames();
SELECT plr_array_accum('{23,35}', 42);
```

Next try to create a helper function (this was copied from (http://www.joeconway.com/plr/doc/plr-pgsql-support-funcs.html) - and test with the following

```
CREATE OR REPLACE FUNCTION plr_array (text, text)
RETURNS text[]
AS '$libdir/plr','plr_array'
LANGUAGE 'C' WITH (isstrict);

select plr_array('hello','world');
```

Using R In PostgreSQL

Creating Median Function in PostgreSQL using R

Below is a link creating a median aggregate function. This basically creates a stub aggregate function that calls the median function in R. http://www.joeconway.com/plr/doc/plr-aggregate-funcs.html **NOTE: I ran into a problem here installing median from the plr-aggregate-funcs via PgAdmin. Gave R-Parse error when trying to use the function. I had to install median function by removing all the carriage returns (\r\n) so put the whole median function body in single line like below to be safe. Evidentally when copying from IE - IE puts in carriage returns instead of unix line breaks. When creating PL/R functions make sure to use Unix line breaks instead of windows carriage returns by using an editor such as Notepad++ that will allow you to specify unix line breaks.**

```
create or replace function r_median(_float8)
        returns float as 'median(arg1)' language 'plr';

CREATE AGGREGATE median (
  sfunc = plr_array_accum,
  basetype = float8,
  stype = _float8,
  finalfunc = r_median
);

create table foo(f0 int, f1 text, f2 float8);
insert into foo values(1,'cat1',1.21);
insert into foo values(2,'cat1',1.24);
insert into foo values(3,'cat1',1.18);
insert into foo values(4,'cat1',1.26);
insert into foo values(5,'cat1',1.15);
insert into foo values(6,'cat2',1.15);
insert into foo values(7,'cat2',1.26);
insert into foo values(8,'cat2',1.32);
insert into foo values(9,'cat2',1.30);

select f1, median(f2) from foo group by f1 order by f1;
```

In the next part of this series, we will cover using PL/R in conjunction with PostGIS.

PLR Part 2: PL/R and PostGIS

PL/R and PostGIS

In this tutorial we will explore using PostGIS and PL/R together. Some examples we will quickly run thru.

- Median Function in conjunction with PostGIS
- Voronoi Diagrams

If you missed part one of our series and are not familiar with R or PL/R, please read
http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgis_tut01
Getting started with PostGIS and

http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgresql_plr_tut01 PLR Part 1: Up and Running with PL/R (PLR) in PostgreSQL: An almost Idiot's Guide

Next create a new PostgreSQL database using the template_postgis database as template and load in the PLR support functions as described in the above article. For this particular exercise, we will assume the database is called **testplrpostgis**.

Loading Some Test GIS Data

We will be working with census data from MassGIS. Click the following "Download This Layer" - census2000blockgroups_poly.exe, from this link
http://www.mass.gov/mgis/cen2000_blockgroups.htm
Get the Massachusetts Town Polygon geometries from this link
http://www.mass.gov/mgis/townssurvey.htm Townsurvey points - ESRI Shapefiles
Get the Massachusetts Census Blocks from this link
http://www.mass.gov/mgis/cen2000_blocks.htm

Get Massachusetts Schools Layer from here
http://www.mass.gov/mgis/schools.htm

Extract the files into a folder (running the exe,right-click extract, and for Linux/Unix/MacOS just extracting with unzip or stuffit or some other similar package)

Now load the shape files into your test database using shp2pgsql by doing the following at your OS commandline. Note for non-windows users or if you installed PostgreSQL in a non-standard directory - change the path to yours - default being "C:\Program Files\PostgreSQL\8.2\bin\". Also replace items in italics with those that match your environment.

```
"C:\Program Files\PostgreSQL\8.2\bin\shp2pgsql" -s 26986 C:\Data\massgis
\census2000blockgroups_poly cens2000bgpoly > cens2000bgpoly.sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -f
cens2000bgpoly.sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -c "CREATE
INDEX idx_cens2000bgpoly_the_geom ON cens2000bgpoly USING GIST(the_geom)"

"C:\Program Files\PostgreSQL\8.2\bin\shp2pgsql" -s 26986 C:\Data\massgis\TOWNSSURVEY_POLY towns
> towns.sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -f towns.
sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -c "CREATE
INDEX idx_towns_the_geom ON towns USING GIST(the_geom)"

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -c "CREATE
INDEX idx_towns_town ON towns USING btree(town)"

"C:\Program Files\PostgreSQL\8.2\bin\shp2pgsql" -s 26986 C:\Data\massgis\SCHOOLS_PT schools >
schools.sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -f schools.
sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -c "CREATE
INDEX idx_schools_the_geom ON towns USING GIST(the_geom)"

"C:\Program Files\PostgreSQL\8.2\bin\shp2pgsql" -s 26986 C:\Data\massgis\census2000blocks_poly
cens2000blocks > cens2000blocks.sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -f
cens2000blocks.sql

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -c "CREATE
INDEX idx_cens2000blocks_the_geom ON cens2000blocks USING GIST(the_geom)"

"C:\Program Files\PostgreSQL\8.2\bin\psql" -h localhost -U postgres -d testplrpostgis -c "vacuum
analyze"
```

Using Median function in conjunction with POSTGIS constructs

In this example - we will find out the median population of a town census blockgroup, total population of all block groups within the town, and average population of a town census blockgroup, and count of block groups for each town that is within each towns boundary for all Massachusetts towns. **Note: we have to do a geomunion here because the towns layer has multiple records per town. Unioning will consolidate so we have a single multipolygon for each town.**

The below example assumes you installed the PL/R aggregate median function as described in part one of this tutorial.

```
SELECT t.town, count(bg.gid) as total_bg, avg(bg.total_pop) as avg_bg_pop,
median(bg.total_pop) as median_pop, sum(bg.total_pop) as totalpop
FROM cens2000bgpoly bg,
(SELECT towns.town, geomunion(the_geom) as the_geom
     FROM towns GROUP BY towns.town) t
WHERE bg.the_geom && t.the_geom AND within(bg.the_geom, t.the_geom)
GROUP BY t.town
ORDER BY t.town
```

The above took about 37859ms ms on my windows 2003 server and 71922 ms on my Windows XP pc, but leaving out the median call did not speed it up much so bottleneck is in the geometry joining part.
If the subquery was a commonly used construct, then we would materialize it as a table and index it appropriately.

```
CREATE TABLE townsingle AS
SELECT MAX(gid) as gid, towns.town, geomunion(the_geom) as the_geom
     FROM towns GROUP BY towns.town;

CREATE INDEX idx_townsingle_the_geom ON townsingle USING GIST(the_geom);
CREATE UNIQUE INDEX idx_townsingle_gid ON townsingle USING btree(gid);
CREATE UNIQUE INDEX idx_townsingle_town ON townsingle USING btree(town);
```

Installing R Packages

Before we continue, we need to expand our R environment by installing **deldir** package which contains voronoi functionality. In order to do so, please follow these steps.

We will be using the deldir package - I listed some other packages here because they looked interesting, but we won't have time to explore those in too much detail in this exercise

- **foreign** - Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase http://cran.us.r-project.org/doc/packages/foreign.pdf
- **GeoXp** - Interactive exploratory spatial data analysis http://cran.us.r-project.org/doc/packages/GeoXp.pdf
- **deldir** - Delaunay Triangulation and Dirichlet (Voronoi) Tesselation. http://cran.us.r-project.org/doc/packages/deldir.pdf
- **gridBase**? - Integration of base and grid graphics
- **gstat**? - geostatistical modelling, prediction and simulation http://www.gstat.org/

1. Open up your RGUI console
2. At the console prompt type the following
   ```
   chooseCRANmirror()
   ```
   This command should pop open a dialog requesting you to pick a mirror. Pick one closest to you and click OK.
3. At the console prompt type
   ```
   utils:::menuInstallPkgs()
   ```
   This command should pop open a dialog listing all the packages available in your selected CRAN mirror.

   If the above doesn't work try

   ```
   available.packages()
   ```

   to get a command line listing of packages available at your chosen CRAN mirror.
4. Hold the control-key down and select the packages listed above and then click ok
   For a list of what other packages are available and what they do check out http://cran.us.r-project.org/src/contrib/PACKAGES.html

   Alternatively for example if you did available.packages(), you can install the packages individually by doing for example< br/>
   ```
   install.packages('deldir')
   ```

   where in this case **deldir** is the name of the package we wish to install.

Exploring Help in R

R offers some neat features as far as help goes that you can access straight from the R GUI console. Below are some useful commands to find out more about a package or drill in at examples.

- **List functions and summary about a package** - *help(package=<package name>)* Example:
  `help(package=deldir)`
- **Search all help text for a particular phrase** - *help.search('<phrase>')* Example:
  `help.search('median')`
- **See list of demos available in all installed packages** - Example:
  `demo(package = .packages(all.available = TRUE))`
- **Load a library and then run a demo in that library**
  library(packagename)
  demo(<demo name>) -
  ```
  Example: libary(gstat)
  demo(block)
  ```

Delaunay Triangulation and Dirichlet Package

In this example we will use a voronoi PL/R implementation provided by Mike Leahy in the PostGIS users newsgroup - http://postgis.refractions.net/pipermail/postgis-users/2007-June/016102.html

The function and voronoi type code is repeated below: Run the below in PgAdmin or psql. **Note for Windows users - if you are copying this - make sure to paste in Notepad and save the file and then open in PgAdmin or psql. This will strip the carriage returns and just leave line breaks. R parser chokes on carriage returns.**

```
--This function uses the deldir library in R to generate voronoi polygons for an input set
of points in a PostGIS table.

--Requirements:
--  R-2.5.0 with deldir-0.0-5 installed
--  PostgreSQL-8.2.x with PostGIS-1.x and PL/R-8.2.0.4 installed

--Usage: select * from voronoi('table','point-column','id-column');

--Where:
--  'table' is the table or query containing the points to be used for generating the
voronoi polygons,
--  'point-column' is a single 'POINT' PostGIS geometry type (each point must be unique)
--  'id-column' is a unique identifying integer for each of the originating points (e.g.,
'gid')

--Output: returns a recordset of the custom type 'voronoi', which contains the id of the
--  originating point, and a polygon geometry

create type voronoi as (id integer, polygon geometry);

create or replace function voronoi(text,text,text) returns setof voronoi as '
    library(deldir)

    # select the point x/y coordinates into a data frame...
    points <- pg.spi.exec(sprintf("select x(%2$s) as x, y(%2$s) as y from %1$s;",arg1,arg2))

    # calculate an approprate buffer distance (~10%):
    buffer = ((abs(max(points$x)-min(points$x))+abs(max(points$y)-min(points$y)))/2)*(0.10)

    # get EWKB for the overall buffer of the convex hull for all points:
    buffer <- pg.spi.exec(sprintf("select buffer(convexhull(geomunion(%2$s)),%3$.6f) as
ewkb from %1$s;",arg1,arg2,buffer))

    # the following use of deldir uses high precision and digits to prevent slivers between
the output polygons, and uses
    # a relatively large bounding box with four dummy points included to ensure that points
in the peripheral areas of the
    # dataset are appropriately enveloped by their corresponding polygons:
    voro = deldir(points$x, points$y, digits=22, frac=0.0000000000000000000000001,list
(ndx=2,ndy=2), rw=c(min(points$x)-abs(min(points$x)-max(points$x)), max(points$x)+abs(min
(points$x)-max(points$x)), min(points$y)-abs(min(points$y)-max(points$y)), max(points$y)+abs
(min(points$y)-max(points$y))))
    tiles = tile.list(voro)
    poly = array()
    id = array()
    p = 1
    for (i in 1:length(tiles)) {
        tile = tiles[[i]]

        curpoly = "POLYGON(("

        for (j in 1:length(tile$x)) {
             curpoly = sprintf("%s %.6f %.6f,",curpoly,tile$x[[j]],tile$y[[j]])
        }
```

```
        curpoly = sprintf("%s %.6f %.6f))",curpoly,tile$x[[1]],tile$y[[1]])

        # this bit will find the original point that corresponds to the current polygon,
along with its id and the SRID used for the
        # point geometry (presumably this is the same for all points)...this will also
filter out the extra polygons created for the
        # four dummy points, as they will not return a result from this query:
        ipoint <- pg.spi.exec(sprintf("select %3$s as id, intersection(''SRID=''||srid(%2
$s)||'';%4$s'',''%5$s'') as polygon from %1$s where intersects(%2$s,''SRID=''||srid(%2
$s)||'';%4$s'');",arg1,arg2,arg3,curpoly,buffer$ewkb[1]))
        if (length(ipoint) > 0)
        {
            poly[[p]] <- ipoint$polygon[1]
            id[[p]] <- ipoint$id[1]
            p = (p + 1)
        }
    }
    return(data.frame(id,poly))
' language 'plr';
```

After you have installed the above Voronoi function. Test it out with the following code

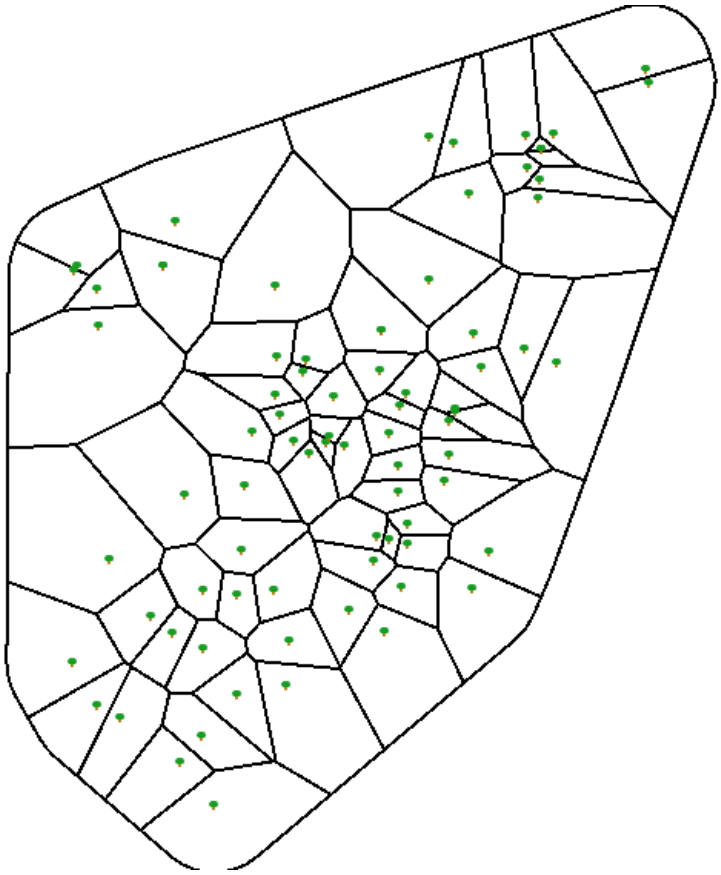
```
CREATE TABLE schools_voroni(gid int);

SELECT AddGeometryColumn('', 'schools_voroni', 'the_geom', 26986, 'POLYGON',2);

INSERT INTO schools_voroni(gid, the_geom)
    SELECT v.id, v.polygon
        FROM voronoi('(SELECT gid, the_geom
        FROM schools
            WHERE town = ''BOSTON'' AND grades LIKE ''%3%'' AND type = ''PUB'') as bs', 'bs.
the_geom', 'bs.gid') As v

ALTER TABLE schools_voroni
  ADD CONSTRAINT pk_schools_voroni PRIMARY KEY(gid);
```

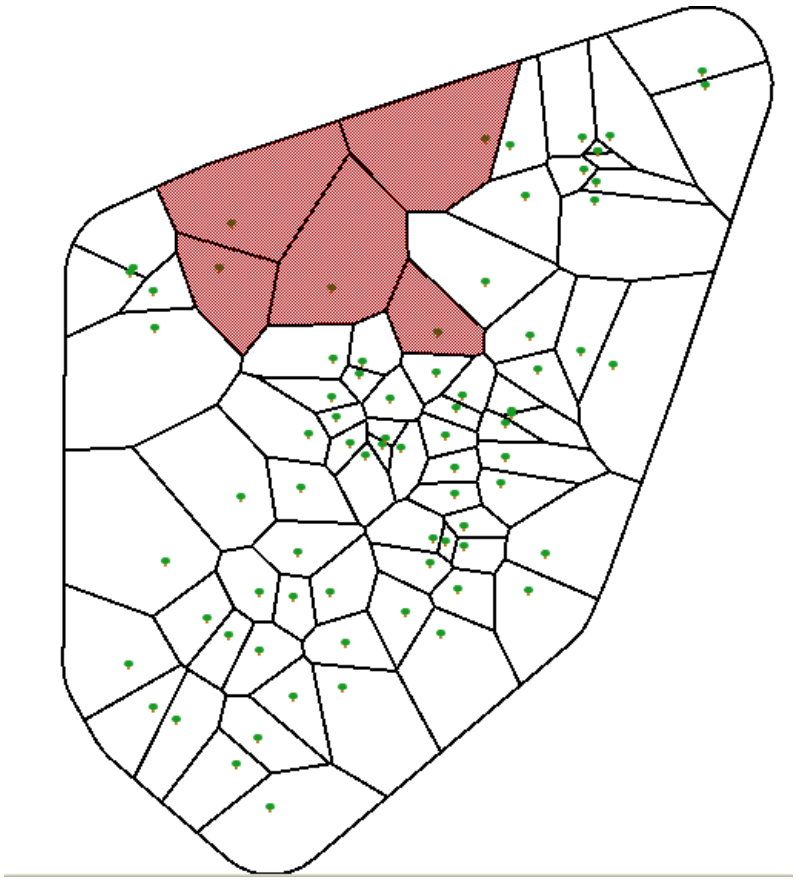
Below is a map of the public elementary schools in Boston overlayed on the Voronoi Polygons of those schools



At this moment, some people may be thinking - Nice polygons, but what is this supposed to tell me about Boston public elementary schools? Well the short simplistic answer is that the Voronoi polygon of a school tells you theoretically what area a school serves, all else being equal. To find out more about this interesting topic, checkout Voronoi links and tidbits. For example we assume people prefer to send their kids to schools near

them. With this information and using census blocks, we can theoretically figure out which areas are underserved based on population of elementary school kids in that voroni polygon area, relative to the number of schools serving that area. So a query something like (NOTE I had only population not elementary school population so this is theoretically flawed already.)

```
 CREATE TABLE vorhighdens As
SELECT v.thegid, sum(pop_2000)/v.schcount As vpop, v.the_geom
FROM (SELECT MAX(gid) as thegid, COUNT(gid) As schcount, the_geom, area(the_geom) as
the_area
FROM schools_voroni
GROUP BY the_geom, area(the_geom)) As v
    INNER JOIN cens2000blocks cb ON v.the_geom && cb.the_geom AND Within(cb.the_geom, v.
the_geom)
GROUP BY v.the_geom, v.schcount, v.thegid
ORDER BY vpop DESC LIMIT 5
```



The darkened areas are where based on my simple model, more elementary schools need to be built. Please NOTE this model is gravely flawed for many reasons

- I have not taken into consideration the size of each school (assumed all are the same size)
- My model is using overall population which in certain areas - like where I live - very few kids live here since its filled with yuppie executives, DINKS, and retired wealthy folk so the population here is probably very skewed.
- I assumed census blocks fit nicely in a voronoi zone which may or may not be true. In theory this is probably not too bad of an assumption
- I assumed all elementary schools are created equal. For example some schools have special needs programs that others do not.

Hopefully there is enough here to demonstrate how a real model would work.

Miscellaneous Tutorials/Cheatsheets/Examples

OGR2OGR Cheatsheet

OGR Tools

The OGR toolkit is a subkit of the FW Tools Toolkit. The FW Tools Toolkit is a toolkit available in both Linux and Windows executable form as well as source code form. It has several command line tools. The ones we find most useful are:

- **OgrInfo** - inspects a GIS datasource and spits out summary data or detailed information about the layers, kinds of geometries found in the file.
- **Ogr2Ogr** - this is a command line tool that converts one Ogr defined data source to another Ogr data source. Ogr supports multiple data formats: ESRI Shapefile, MapInfo Tab file, TIGER, s57, DGN, CSV, DBF, GML, KML, Interlis, SQLite, ODBC, PostGIS/PostgreSQL, MySQL .

These 2 command line tools can be found in the **bin** folder of your FWTools install. To start using these tools

1. Install the FW Tools tool kit.
2. Launch the **FW Tools Shell** - in windows this is found under Start->Programs->FW Tools ..
3. From the shell - cd into your directory that has the data you want to convert

Getting more Help

If you want a comprehensive listing of options offered by **ogr2ogr** or **ogrinfo**, run the following at the FW Tools Shell.

```
ogr2ogr --help
ogrinfo --help
```

Conversions from MapInfo to Other formats

Conversion from MapInfo to ESRI Shape

```
ogr2ogr -f "ESRI Shapefile" mydata.shp mydata.tab
```

Conversion from MapInfo to PostGIS

```
ogr2ogr -f "PostgreSQL" PG:"host=myhost user=myloginname dbname=mydbname password=mypassword"
mytabfile.tab
```

Note: for the above, you can leave out the host if its localhost and user and password if you have your authentication set to trust.

Importing as a different table name

In the below example, we don't want OGR to create a table called **mytable**. We instead want to call the table something different like **newtablename**. To do so we use the **nln** option.

```
ogr2ogr -f "PostgreSQL" PG:"host=myhost user=myloginname dbname=mydbname password=mypassword"
mytabfile.tab -nln newtablename
```

When OGR guesses wrong or fails to guess

Sometimes OGR does not output the right projection, particularly with Units of Feet or data that has no projection info or the projection information can't be easily translated to your system. Sometimes OGR can't match the projection to one in your spatial_ref_sys table so creates a new entry in that table. In these cases you have to tell OGR what the output projection is. You do this with the **-a_srs** flag.

```
ogr2ogr -f "PostgreSQL" -a_srs "EPSG:2249" PG:"host=myhost user=myloginname dbname=mydbname
password=mypassword" mytabfile.tab
```

In the above example I told OGR2OGR to assume the source/output projection is in Massachusetts Mainland US Ft. **Note:** All Spatial Ref Systems can be found in the spatial_ref_sys table of PostGIS or the Data/gcs.csv file of your FW Tools install.

Conversions from PostGIS to Other formats

Conversion from PostGIS to ESRI Shape

The **pgsql2shp** and **shp2pgsql** are usually the best tools for converting back and forth between PostGIS and ESRI for 2 main reasons.

- It has fewer idiosyncracies when converting data
- It has a lot fewer dependencies so can fit on your floppy.

If you really want to use Ogr2Ogr for this kind of conversion, below is the standard way to do it

```
ogr2ogr -f "ESRI Shapefile" mydata.shp PG:"host=myhost user=myloginname dbname=mydbname
password=mypassword" "mytable"
```

Selecting specific fields, sets of data and Geometry

Sometimes you have more than one geometry field in a table, and ESRI shape can only support one geometry field per shape. Also you may only want a subset of data. In these cases, you will need to select the geometry field to use. The most flexible way to do this is to use the -sql command which will take any sql statement.

```
ogr2ogr -f "ESRI Shapefile" mydata.shp PG:"host=myhost user=myloginname dbname=mydbname
password=mypassword" -sql "SELECT name, the_geom FROM neighborhoods"
```

Example snippet converting from PostGIS to KML

```
ogr2ogr -f "KML" neighborhoods.kml PG:"host=myhost user=myloginname dbname=mydbname
password=mypassword" -sql "select gid, name, the_geom from neighborhoods"
```

Exporting multiple tables from PostGIS using Ogr2Ogr

One way in which ogr2ogr excels above using the pgsql2shp tool is that ogr2ogr can export multiple tables at once. This is pretty handy for sharing your postgis data with others who do not have a postgis database.

The code below will export all your postgis tables out into a folder called *mydatadump* in ESRI shape (shp) format.

```
ogr2ogr -f "ESRI Shapefile" mydatadump PG:"host=myhost user=myloginname dbname=mydbname
password=mypassword"
```

The code below will export all your postgis tables out into a folder called *mydatadump* in MapInfo .tab format.

```
ogr2ogr -f "MapInfo File" mydatadump PG:"host=myhost user=myloginname dbname=mydbname
password=mypassword"
```

Now most of the time you probably only want to output a subset of your postgis tables rather than all your tables. This code exports only the neighborhoods and parcels tables to a folder called *mydatadump* in ESRI shapefile format

```
ogr2ogr -f "ESRI Shapefile" mydatadump PG:"host=myhost user=myloginname dbname=mydbname
password=mypassword" neighborhood parcels
```

Conversion from TIGER to other formats

Topologically Integrated Geographic Encoding and Referencing system (TIGER) is the US Census Bureaus proprietary format for exporting US Census geographic and statistical data. Starting in 2007, they will be using ESRI Shapefile (SHP) as there official export format. So this section may be a bit obsolete for the upcoming versions.

To get the files for your location - you can browse their archive at http://www.census.gov/geo/www/tiger/index.html

Reading the meta data using ogrinfo

```
ogrinfo TGR25025.RTI
```

Conversion from Tiger to ESRI shape

Give me all layers in TGR25025

The tiger files contain a set of layers, so unlike the other outputs we have done, we will specify a folder to dump all the layers into

```
ogr2ogr -f "ESRI Shapefile" masuffolk TGR25025.RTI
```

Note: The above outputs all the tiger layers in the TGR25025 set into a folder called masuffolk that resides within our data folder that we have cded to.

Just One Layer

```
ogr2ogr -f "ESRI Shapefile" sufcomp.shp TGR25025.RT1 layer CompleteChain
```

In the above, we are asking for just the CompleteChain layer and to output to a new file called sufcomp.shp. Note it will output shp and the corresponding shx, and prj files.

Conversion from TIGER to MapInfo

The conversion follows a similar path to ESRI Shape

All Layers - Each Layer as a single file

The below will create a folder masuf and output all the layers into that folder and give each a tab file extension

```
ogr2ogr -f "MapInfo File" masuf TGR25025.RT1
```

Single Layer - Single File

```
ogr2ogr -f "MapInfo File" sufcomp.tab TGR25025.RT1 layer CompleteChain
```

Conversion from Tiger to PostGIS

```
ogr2ogr -update -append -f "PostGreSQL" PG:"host=myserver user=myusername dbname=mydbname
password=mypassword" TGR25025.RT1 layer CompleteChain -nln masuf -a_srs "EPSG:4269"
```

Note in the above we needed to put the -update -append option because OGR2OGR will try to create a folder if given a file with no extension, which translates to creating a new database.

We also put in the **-nln masuf** to prevent OGR2OGR from creating the table name as CompleteChain

Lastly we put in EPSG:4269 to tell OGR to just assume Tiger is in NAD 83 long lat. Without this it creates an entry in the spatial_ref_sys table which is equivalent to the already existing well known 4269.

Using OGR to Import Non-spatial Data

While OGR was designed primarily to transform data between different spatial datasources, it is a little known fact that it can be used as well for importing non-spatial datasources such as Dbase files and CSV files.

Importing Dbase file into PostgreSQL using Ogr2Ogr

In the below example, we will demonstrate importing a Dbase file into PostgreSQL using OGR2OGR

```
ogr2ogr -f "PostgreSQL" PG:"host=myserver user=myusername dbname=mydbname password=mypassword"
sometable.dbf -nln "sometable"
```

download

Using OpenLayers: Part 1

What is Open Layers?

Open Layers is an opensource client-side JavaScript/AJAX framework for overlaying various mapping services. It supports various mapping apis such as Google, Yahoo, Microsoft Virtual Earth, OGC WMS, OGC WFS, KaMap, Text layers, and Markers to name a few. The nice thing about it being a pure client-side implementation is that you can drive it with any server language such as ASP.NET, PHP, PERL and for simple maps, embed directly into a plain html file. There is minimal requirement from the webserver if you are using publicly available or subscription layers.

In the next couple of sections, we'll test drive OpenLayers by overlaying various layers from Boston.

Downloading the classes and setting up the Base page

We will be using the 2.2 API which you can download from http://openlayers.org/download/OpenLayers-2.2.zip

1. Download the file from the above URL.
2. Extract the folder copy out the build/OpenLayers.js and the img and theme folders into a new folder called ol22
3. Create a blank file called example1.htm file which sits in the same directory as you ol22 directory so you will have ol22, example1.htm

Copy the contents of http://www.bostongis.com/demos/OpenLayers/example1.htm to your blank file.

Dissecting the Pieces of the Application

In the above example which you can see from the link above, we have drawn a Microsoft Virtual Earth Map layer, and 3 WMS layers - Boston Neighborhood Boundary, Boston Mainstreets Boundary, and Boston Water Taxi stops. Two of our WMS layers come from a UMN Mapserver dishing out two layers at once and our Water Taxi layer is being dished out by Mass GIS Geoserver - http://lyceum.massgis.state.ma.us/wiki/doku.php?id=history:home .

Adding the necessary script references

First for any 3rd party mapping services you will be using, you need to include the libraries in addition to the OpenLayers library file and these should be included before your OpenLayers include. The code to do that is on line 11 and 12 of the htm file and look like this.

```
11:  <script src='http://dev.virtualearth.net/mapcontrol/v3/mapcontrol.js'></script>
12:  <script src="ol22/OpenLayers.js"></script>
```

Alternatively, you can use the latest release OpenLayers.js directly from the OpenLayers site by replace line 12 with

```
12:  <script src="http://openlayers.org/api/OpenLayers.js"></script>
```

Creating the Open Layers map Object and adding layers

Next we create a blank div with id=map that has the dimensions we want and position that where we want on the page. Our map will live there.

```
16:    <div id="map" style="width: 400px; height: 400px"></div>
```

Next we write the javascript code to create the map and load into our div and add the layers

```
18:    var map = new OpenLayers.Map(document.getElementById("map"));
19:    var dndwmsurl =  "http://dndmaps.cityofboston.gov/mapserv/scripts/mapserv410/mapserv410.exe?
map=\\mapserv\\dndwms\\dndbasepg.map&"
20:
21:    map.addLayer(new OpenLayers.Layer.VirtualEarth("VE"));
22:
23:     /**Note we don't have to specify an SRS, Service or Request for WMS layers below.
24:        OpenLayer will ask for projection based on base our base layer, EPSG:4326, Service: WMS,
Request: GetMap.
25:        We chose image/gif because IE6 and below doesn't natively support transparency for png
without a hack.  **/
```

```
26:    wmstaxi = new OpenLayers.Layer.WMS("MASSGIS Boston Taxi Stops", "http://giswebservices.
massgis.state.ma.us/geoserver/wms",
27:            {layers: "massgis:GISDATA.WATERTAXISTOPS_PT", transparent: "true", format: "image/
gif"},
28:                {tileSize: new OpenLayers.Size(400,400), buffer: 1 })
29:    map.addLayer(wmstaxi)
30:
31:    var wmsbos = new OpenLayers.Layer.WMS("Boston Neigborhoods and Mainstreets",dndwmsurl,
32:                {layers: "neighborhoods,mainstreets", transparent:"true",format: "image/gif"},
33:                    {tileSize: new OpenLayers.Size(400,400), buffer: 1 });
34:
35:    map.addLayer(wmsbos);
```

A couple of notes about the above code that may not be entirely obvious

- In the UMN MapServer WMS we have extra slashes for the map file because our map location has \. E.g its \\dndwms instead of just \. This is to escape out the \. Future versions of Open Layers will do this automatically.
- For the WMS layers we specified a tileSize and buffer. This is to prevent Open Layers from slicing up the tiles on the server too granulary - otherwise it will ask for a lot more tiles per call. The layers we are overlaying are fairly light so don't need many calls.

Centering the map and adding the layer switcher

The layer switcher is the little + sign on the right side that shows you the layers active and allows you to deactivate a layer or reactivate a layer.

```
37:    var boston = new OpenLayers.LonLat(-71.0891380310059,42.3123226165771);
38:    map.setCenter(boston, 10);
39:    map.addControl(new OpenLayers.Control.LayerSwitcher());
```

download

Using OpenLayers: Part 2

In this tutorial we will just show some example snippets of using OpenLayers that we have found most useful. We will be assuming Open Layers 2.4 and above.

Initializing Layers Off

In our previous example, you will notice that all the layers we have added are turned on by default. In order to turn the layers off, we can pass in the additional parameter of **visibility**. Like the below line 28. Note that visibility applies to all classes of layers, not just WMS layers.

```
26:    wmstaxi = new OpenLayers.Layer.WMS("MASSGIS Boston Taxi Stops", "http://giswebservices.
massgis.state.ma.us/geoserver/wms",
27:            {layers: "massgis:GISDATA.WATERTAXISTOPS_PT", transparent: "true", format: "image/
gif"},
28:                {tileSize: new OpenLayers.Size(400,400), buffer: 1,  visibility: false })
```

The above effect can be seen http://www.bostongis.com/demos/OpenLayers/example2.htm.

If you want to turn off a layer after initialization, then you would use the **setVisibility** method of a layer. For example - **mylayer.setVisibility (false)**. This would be useful say if you did not want to use the built-in layer switcher, but instead for example have sets of layers display and not display based on others.

Forcing a layer to be a Base Layer

Within an OpenLayers map, you can have only one layer selected as Base. This layer will control the projection of other layers among other things. In the layer switcher, you will see base layer options marked at the top in a category section called Base Layer. Each option will have a radio button instead of a checkbox to prevent from selecting more than one. By default the first Base Layer added to the map becomes the active base layer.

Certain layers have to be base layers because they are not transparent and there are other things you can't control about them like projection and so forth. For example commercial layers such as Google, Yahoo, and Virtual Earth are just assumed to be base layers so you don't have to explicitly state them to be and actually can't force them not to be.

If you have a WMS layer for example, it can be used as a base layer or an overlay layer. If you don't explicitly state it is a base layer, it will be assumed to be an overlay layer. So how do you force baseness you ask, simple by setting the **isBaseLayer** property of the layer as shown below.

```
          map.addLayer(new OpenLayers.Layer.WMS.Untiled("MassGIS Boundary", "http://
giswebservices.massgis.state.ma.us/geoserver/wms",
{'layers': "massgis:GISDATA.BOUNDARY_ARC",'transparent':"true",'format': "image/png"},
{'isBaseLayer':true}));
```

To Tile or Not to Tile

There are currently two kinds of layers in which you have the option of tiling and not tiling. These are WMS layers and Mapserver layers. Each of these are exposed as separate layer classes **WMS**, **WMS.Untiled**, **Mapserver**, **Mapserver.Untiled**. In what circumstances would you want to tile and other circumstances where you would not want to tile. To first answer this question, its useful to think about what happens when you tile.

When maps are tiled, the map viewport is broken out into several quandrants - which tries to approximate what you have set for TileSize, but often falls short. So for each load of the screen, multiple requests sometimes in the order of 60 per layer paint is requested. You can imagine this can be a considerable strain on a weak server and especially silly when what you are asking for is a light boundary file of say Boston. In the untiled case, only one request is made per layer, but the request is usually on the order of 4 times larger than the view port to allow for smoother panning effects at the cost of a little slower load time. Outlined below are the cases where we've found tiling or untiling useful. Some are a matter of taste and some are a matter of practicality.

Use Untiled in the following scenarios

- Fairly light-weight (in file size) that span huge areas
- Processor constrained WMS server
- If you find it annoying that half of your map paints while the other half is loading
- High band-width server and high band-width clients
- Your map images return embedded scales.

Use tiled during the following scenarios

- Fairly heavy geometries (in file size) that span small areas
- Super fast WMS server or server with tile caching built in
- Low band-width clients
- Relatively long pauses of a completely blank map area that suddenly loads all at once annoys you.

An example of an untiled layer is shown in our visibility discussion. An example of an untiled is shown in our Base Layer discussion.

PostGIS Code Snippets

Extent Expand Buffer Distance: PostGIS - ST_Extent, Expand, ST_Buffer, ST_Distance

Extent Expand Buffer Distance ST_DWithin

In this quick exercise, we will explore the following PostGIS OGC functions: Extent, Expand, Buffer, Distance

Extent

Extent is an aggregate function - meaning that it is used just as you would use SQL sum, average, min, and max often times in conjunction with group by to consolidate a set of rows into one. Pre-1.2.2 It returned a 2-dimensional bounding box object (BBOX2D) that encompasses the set of geometries you are consolidating.

Unlike most functions in PostGIS, it returns a postgis BBOX object instead of a geometry object.
In some cases, it may be necessary to cast the resulting value to a PostGIS geometry for example if you need to do operations that work on projections etc. Since a bounding box object does not contain projection information, it is best to use the setSRID function as opposed to simply casting it to a geometry if you need projection information. SetSRID will automagically convert a BBOX to a geometry and then stuff in SRID info specified in the function call.

Extent3d

Extent has a sibling called **Extent3d** which is also an aggregate function and is exactly like Extent except it returns a 3-dimensional bounding box (BBOX3D).

ST_Extent

Starting around version 1.2.2, Extent and Extent3d will be deprecated in favor of **ST_Extent** . *ST_Extent* will return a BOX3D object.

Expand (< 1.3.1), ST_Expand (1.2.2 +)

Expand returns a geometry object that is a box encompassing a given geometry. Unlike extent, it is not an aggregate function. It is often used in conjunction with the distance function to do proximity searches because it is less expensive than the distance function alone.

The reason why expand combined with distance is much faster than distance alone is that it can utilize gist indexes since it does compares between boxes so therefore reduces the set of geometries that the distance function needs to check.

Note in versions of PostGIS after 1.2, there exists a new function called **ST_DWithin** which utilizes indexes, simpler to write than the expand, &&, distance combination.

ST_Distance will be the preferred name in version 1.2.2 and up

The following statements return equivalent values, but the one using Expand is much faster especially when geometries are indexed and commonly used attributes are indexed.

Find all buildings located within 100 meters of Roslindale
Using distance and Expand: Time with indexed geometries: 14.5 seconds - returns 8432 records

```
               SELECT b.the_geom_nad83m
FROM neighborhoods n, buildings b
WHERE n.name = 'Roslindale' and expand(n.thegeom_meter, 100) && b.thegeom_meter
        and distance(n.thegeom_meter, b.thegeom_meter) < 100
```

Using distance alone: Time with indexed geometries: 8.7 minutes - returns 8432 records

```
               SELECT b.the_geom_nad83m
FROM neighborhoods n, buildings b
WHERE n.name = 'Roslindale'
        and distance(n.thegeom_meter, b.thegeom_meter) < 100
```

ST_DWithin (1.3.1 and above)

We will write the above using the new ST_DWithin to demonstrate how much easier it is.

Using ST_DWithin: Time with indexed geometries: 14.5 seconds - returns 8432 records

```
SELECT b.the_geom_nad83m
        FROM neighborhoods n, buildings b
                WHERE n.name = 'Roslindale' and ST_DWithin(n.thegeom_meter, b.thegeom_meter, 100)
```

Within (< 1.3.1), ST_Within (1.3.1 and above)

ST_Within(A,B) returns true if the geometry A is within B. There is an important distinction between Within and ST_Within and that is the ST_Within does an implicit A&&B call to utilize indexes where as Within and _ST_Within do not.

1.3.1 and above do

```
SELECT b.the_geom_nad83m
        FROM neighborhoods n, buildings b
                WHERE n.name = 'Roslindale' and ST_Within(b.thegeom_meter, n.thegeom_meter)
```

Pre 1.3.1 do

```
SELECT b.the_geom_nad83m
        FROM neighborhoods n, buildings b
                WHERE n.name = 'Roslindale' and b.thegeom_meter && n.thegeom_meter AND Within(b.
thegeom_meter, n.thegeom_meter)
```

ST_Buffer (+1.2.2), Buffer (< 1.2.2)

Buffer returns a geometry object that is the radial expansion of a geometry expanded by the specified number of units. Calculations are in units of the Spatial Reference System of this Geometry. The optional third parameter sets the number of segments used to approximate a quarter circle (defaults to 8) if third argument is not provided.
This is a much more involved process than the expand function because it needs to look at every point of a geometry whereas the expand function only looks at the bounding box of a geometry.

Aliases: ST_Buffer (MM-SQL)

Correcting Invalid Geometries with Buffer

Buffer can also be used to correct invalid geometries by smoothing out self-intersections. It doesn't work for all invalid geometries, but works for some. For example the below code will correct invalid neighborhood geometries that can be corrected. Note in here I am also combining the use with MULTI since in this case buffer will return a polygon and our table geometries are stored as multi-polygons. If your column geometry is a **POLYGON** rather than a MULTIPOLYGON then you can leave out the **MULTI** part.

```
        UPDATE neighborhoods
                SET the_geom = multi(buffer(the_geom, 0.0))
                WHERE isvalid(the_geom) = false AND isvalid(buffer(the_geom, 0.0)) = true
```

Pictorial View of Buffer, Expand, Extent

In this section we provide a graphical representation of what the operations Buffer, Expand, Extent look like when applied to geometries.

Legend

Original

Extent

Expand

Buffer

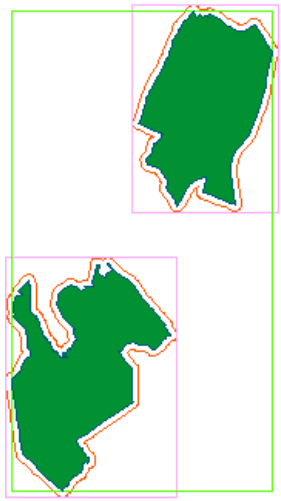
Corresponding Queries
Original:

```
SELECT the_geom, name
FROM neighborhoods
        WHERE name IN('Hyde Park','Roxbury')
```

Expand: Draws a box that extends out 500 units from bounding box of each geometry

```
SELECT expand(the_geom, 500) as geom, name
FROM neighborhoods
WHERE name IN('Hyde Park', 'Roxbury')
```

Extent: Draws a single bounding box around the set of geometries



```
SELECT extent(the_geom) as thebbox, name
FROM neighborhoods
WHERE name IN('Hyde Park', 'Roxbury')
```

Buffer: Extends each geometry out by 500 units.

```
SELECT buffer(the_geom,500) as geom, name
FROM neighborhoods
WHERE name IN('Hyde Park', 'Roxbury')
```

PostGIS GeomFromText

GeomFromText: Loading a Geometry in Well-Known-text (WKT)) into PostGIS

The GeomFromText OGC function is used to convert a Well Known Text (WKT) version of a geometry into a PostGIS geometry. **Aliases:** GeometryFromText
SQL-MM Spec Equivalents ST_GeomFromText, ST_WKTToSQL ()

Example inserting a linestring

This example demonstrates using a WKT representation of a Street in NAD 83 LongLat format and inserting it into a PostGIS geometry field.

```
INSERT INTO streets(street_name, the_geom)
SELECT 'some street',
GeomFromText('LINESTRING(-70.729212 42.373848,-70.67569 42.375098)',4269)
```

The GeomFromText function or equivalent exists in other spatial databases. For example in MySQL 5 and above, the function is called the same and uses the same syntax as above. In Oracle 10g Spatial and Locator, you would use the **SDO_GEOMETRY** function. So the above in Oracle would look something like.

```
INSERT INTO streets(street_name, the_geom)
SELECT 'some street',
SDO_GEOMETRY('LINESTRING(-70.729212 42.373848,-70.67569 42.375098)',SDO_CS.MAP_EPSG_SRID_TO_ORACLE
(4269)))
FROM DUAL
```

Note in the above code we are using an Oracle function called SDO_CS.MAP_EPSG_SRID_TO_ORACLE(), because Oracle Spatial Reference System IDs (SRID) are usually not the same as the commonly used European Petroleum Survey Group (EPSG)standard codes. PostGIS and MySQL SRIDs are generally the same as the EPSG IDs so that call is not necessary.

Example inserting a multipolygon

```
INSERT INTO boston_buildings(name, the_geom)
VALUES('some name', GeomFromText('MULTIPOLYGON(((235670.354215375
894016.780856,235668.324215375 894025.050856,235681.154215375
 894028.210856,235683.184215375 894019.940856,235670.354215375 894016.780856)))', 2805) )
```

Intersects Intersection: PostGIS - ST_Intersects, ST_Intersection

ST_Intersects, Intersects

ST_Intersects is a function that takes two geometries and returns true if any part of those geometries is shared between the 2. In PostGIS versions before 1.3 you would use the following syntax to utilize indexes

```
SELECT a.somfield, b.somefield2
FROM a INNER JOIN b ON
       (a.the_geom && b.the_geom AND intersects(a.the_geom, b.the_geom))
```

In versions from 1.3 forward, the && indexable operator is automatically included in the definition of ST_Intersects so you can simply write

```
SELECT a.somfield, b.somefield2
FROM a INNER JOIN b ON ST_Intersects(a.the_geom, b.the_geom)
```

ST_Intersection, Intersection

The functions ST_Intersection and Intersection are compliments to ST_Intersects. What they return is that portion of geometry A and geometry B that is shared between the two geometries. If the two geometries do not intersect, then what is returned is an empty GEOMETRYCOLLECTION object.

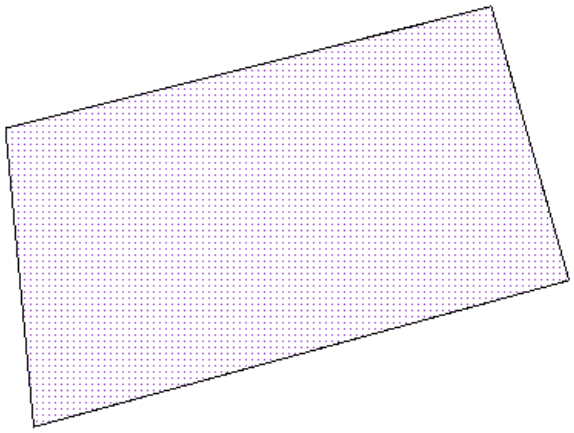
NOTE: PostGIS versions 1.2.2 and up you should use ST_Intersection as Intersection is deprecated. The two functions are more or less equivalent though.

Pictorial View of ST_Intersection

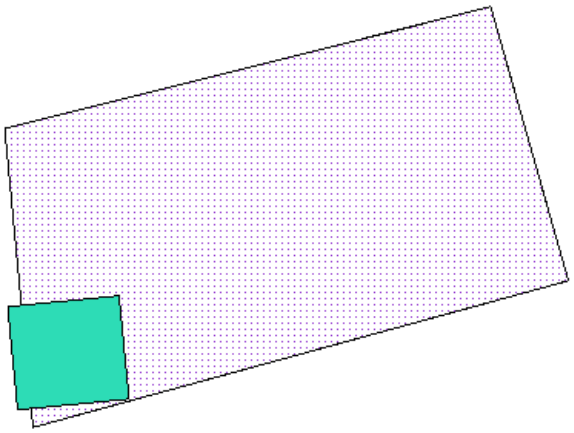
In this section we provide a graphical representation of what the Intersection looks like. We will use an example of a building and a parcel where the building does not completely sit inside the parcel.

```
SELECT b.the_geom As bgeom, p.the_geom As pgeom,
               ST_Intersection(b.the_geom, p.the_geom) As intersect_bp
       FROM buildings b INNER JOIN parcels p ON ST_Intersection(b,p)
       WHERE ST_Overlaps(b.the_geom, p.the_geom)
       LIMIT 1;
```

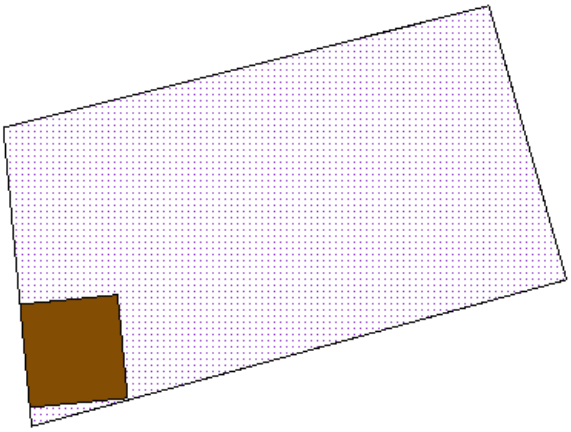
Parcel Geometry: pgeom:



Building Geometry Overlayed on Top Of Parcel: bgeom: The building is in green



The geometry returned by ST_Intersection(b.the_geom, p.the_geom) overlayed on top of Parcel: (intersection is in brown)



PostGIS MakeLine ST_MakeLine Examples

ST_Makeline is an aggregate function that takes a sequence of points and strings them together to make a line. In versions of PostGIS prior to 1.2.2, this was called *MakeLine*. For prior versions - replace ST_MakeLine with MakeLine.

ST_MakeLine Example: Create a line string of Boston Marathon practice route for each day

In this example we have a table of gps point snapshots for our marathon practice for each day broken out by time. We want to convert the points for each day into a single record containing the line path of our run for that day.

NOTE: For this example the trip_datetime field is of type timestamp so records the date and time the gps position was recorded. **CAST (trip_datetime As date)** (this is the ANSI sql standard) or PostgreSQL specific short-hand **trip_datetime::date** strips off the time part so we are just left with the day.

We do a subselect with an order by to force the points to be in order of time so that the points of the line follow the path of our run across time.

```
SELECT St_MakeLine(the_point) as the_route, bp.trip_date
FROM (SELECT the_point, CAST(trip_datetime As date) as trip_date
                FROM boston_marathon_practice
                        ORDER BY trip_datetime) bp
GROUP BY bp.trip_date;
```

Convert Parcel points into line segments

In the previous example since our grouping field is in the same order as the datetime, we only needed to order by one field in our subselect. If your groupings are not in the same order as your line order, then you need to do additional orders for each field you plan to group by. If you don't your resulting points may get shuffled. In the below simplified example we have parcel centroids and we want to make line segments that join them such that for each unique street and zip we have one line segment. For extra measure we want our final set to include the start number range and end number for this segment.

This is a failry simplistic example. In reality you would probably need to do a little bit more because street segments have same names in Boston even within the same zip. We are also assuming the line segment drawing should follow the order of the street numbers and our street numbers are numeric.

```
SELECT p.street, p.zip, ST_MakeLine(p.the_point) As streetsegment,
          MIN(st_num) as st_num_start, MAX(st_num) As st_num_end
FROM (SELECT street, zip, centroid(p.the_geom) as the_point, st_num
          FROM parcels ORDER BY zip, street, st_num) p
GROUP BY p.zip, p.street
```

PostGIS MakePoint

Creating Point Geometries with MakePoint

There are numerous ways of creating point geometries in PostGIS. We have covered these ways in other snippets. Check out the fulling links for other examples.

- GeomFromText
- PointFromText

MakePoint is perhaps in terms of speed the fastest way of creating a PostGIS geometry and on tests I have done, can be as much as 5 to 10 times faster. The downside of MakePoint is that it is not defined in the OGC spec so its not quite as portable across other spatial databases as is GeomFromText and PointFromText.

MakePoint is used in the form MakePoint(**x**, **y**) which for pretty much all spatial reference systems means MakePoint(**longitude**, **latitude**)

Transformable (Reprojectable) and Non-Transformable Geometries

MakePoint used alone creates a nontransformable geometry (one that has an SRID = -1, so because of that, MakePoint is usually used in conjunction with **SetSRID** to create a point geometry with spatial reference information. Examples below and the EWKT output to demonstrate the differences.

Example below can not be used in a transformation to transform to another Spatial Reference System

```
SELECT MakePoint(-71.09405383923, 42.3151215523721) as the_point, AsEWKT(MakePoint(-
71.09405383923, 42.3151215523721)) as ewkt_rep
```

Note - the ewkt_rep output looks exactly like the WKT output - e.g. no spatial reference information so can not be passed into a transformation call - ***POINT(-71.09405383923 42.3151215523721)***

Emparting Spatial Information to MakePoint Geometry

```
SELECT SETSRID(MakePoint(-71.09405383923, 42.3151215523721),4326) as the_point, AsEWKT(SETSRID
(MakePoint(-71.09405383923, 42.3151215523721),4326)) as ewkt_rep
```

Note - the ewkt_rep output looks like the regular WKT output but has extra information about the spatial refence system the point is defined in
SRID=4326;POINT(-71.09405383923 42.3151215523721)

PointFromText, LineFromText, ST_PointFromText, ST_LineFromText OGC functions - PostGIS

Loading Well-Known-Text (WKT) using PointFromText, LineFromText, MPointFromText, MLineFromText, MPolyFromText, PolyFromText

Syntax: PointFromText(*wkt representation*, *SRID*)

PointFromText(*wkt representation*) Note the versions that don't take an SRID return unprojectable geometries so should probably be avoided

Note: Beginning with 1.2.1+ - the names without the ST_ prefix are preferred over the non-ST prefixed ones to conform to newer OGC standards. In future releases the non-ST names will be deprecated and eventually removed from PostGIS.

ST_PointFromText (previously known as PointFromText) and ST_LineFromText (previously known as LineFromText) and the other <Geometry Type>FromText functions are very similar to the all-purpose GeomFromText, except that PointFromText only converts Point WKT geometry representations to a geometry object and returns null if another kind of Geometry is passed in. LineFromText similarly only converts WKT representations of Lines to geometry and returns null if another WKT representation is passed in.

If you look at the current underlying implementation of these functions in PostGIS (1.2.1 and below), you will observe that they in fact call GeomFromText, but if the geometry type of the generated geometry is not a Point or a LineString respectively, then the functions return null.

The M*FromText, ST_M*FromText functions create Multi geometry types from the WKT representation of those types.

Since these function do have some added overhead (not much but some), it makes sense to just use GeomFromText unless you have a mixed bag of WKT geometries coming in and only wish to selectively insert say the Points or the Lines or if just for plain readability to make it clear that you are indeed inserting Points or lines or MultiLineStrings etc.

```
UPDATE points_of_interest SET thepoint_lonlat = PointFromText('POINT(' || longitude || ' ' ||
latitude || ')',4326)
```

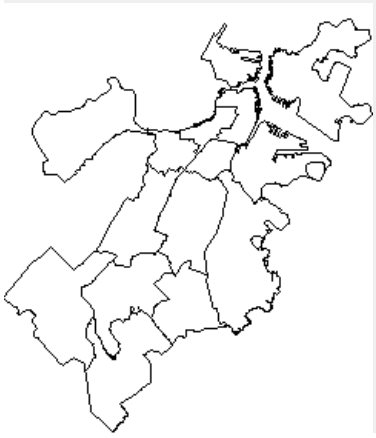
PostGIS Simplify

Simplify: Reduce the weight of a geometry

Simplify reduces the complexity and weight of a geometry using the Douglas-Peucker algorithm. It in general reduces the number of vertices in a polygon, multipolygon, line, or multi-line geometry. It takes 2 arguments, the geometry and the tolerance. To demonstrate we will apply the simplify function on Boston neighborhoods that are in SRID 2249 (NAD83 / Massachusetts Mainland (ftUS)) and compare the sizes of the shape files if we were to dump these out as ESRI shape files. We will also show pictorially how simplify changes the geometries.

Original: Size of Shp - 95 kb

```
SELECT the_geom
        FROM neighborhoods
```



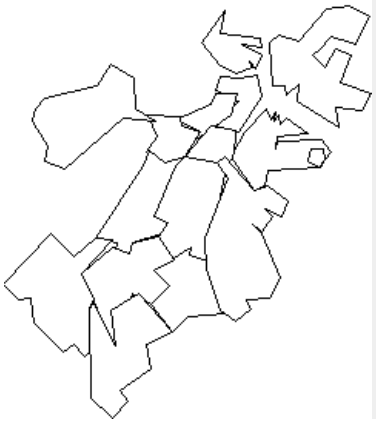
Simplify 500: Size of Shp - 7 kb

```
SELECT simplify(the_geom,500) as simpgeom
        FROM neighborhoods
```



Simplify 1000: Size of Shp - 5 kb

```
SELECT simplify(the_geom,1000) as simpgeom
        FROM neighborhoods
```



Note: When simplifying longlat data, you often get very strange results with simplify. It is best to first transform data from longlat to some other coordinate system, then simplify and then transform back to longlat. So something like

```
SELECT transform(simplify(transform(the_geom, 2249), 500),4326) from neighborhoods
```

Summary (pre 1.3.1 name), ST_Summary (+1.3.1)

ST_Summary(geometry) gives you a brief summary of a geometry telling you how many simple geometries, rings and type of geometry it is.

```
select ST_Summary(the_geom) from neighborhoods;
```

Output of the above ran from psql looks like this

```
 MultiPolygon[BS] with 1 elements
   Polygon[] with 2 rings
    ring 0 has 649 points
    ring 1 has 10 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 182 points

 MultiPolygon[BS] with 2 elements
   Polygon[] with 1 rings
    ring 0 has 319 points
   Polygon[] with 1 rings
    ring 0 has 4 points

 MultiPolygon[BS] with 2 elements
   Polygon[] with 1 rings
    ring 0 has 376 points
   Polygon[] with 1 rings
    ring 0 has 68 points

 MultiPolygon[BS] with 3 elements
   Polygon[] with 1 rings
    ring 0 has 215 points
   Polygon[] with 2 rings
    ring 0 has 4 points
    ring 1 has 23 points
   Polygon[] with 1 rings
    ring 0 has 45 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 71 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 2 rings
    ring 0 has 124 points
    ring 1 has 5 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 199 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 197 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 168 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 400 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 355 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 176 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 1 rings
    ring 0 has 392 points

 MultiPolygon[BS] with 1 elements
   Polygon[] with 2 rings
    ring 0 has 1965 points
    ring 1 has 29 points
```

PostGIS: ST_Translate

ST_Translate, Translate

The ST_Translate function takes any geometry (linestring, multiline etc) returns a new geometry that is the original geometry moved by a vector defined by X,Y,Z. Note the units of measurement are always in the units of the spatial reference system of the geometry argument. There are two forms of it ST_Translate. ST_Translate(geometry, X, Y, Z) and ST_Translate(geometry, X,Y).

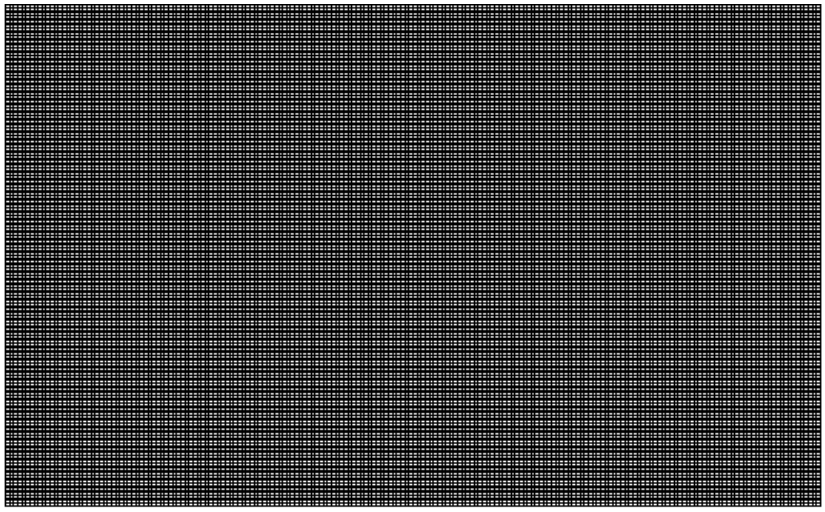
In the below example we use the translate function to create a grid that divides the extent of Boston into into 200x200 rectangles (40,000). By doing the following

- Create a reference box starting at the origin of our extent of Massachusetts that is of dimension 1485x911 meters - in quasi OGC notation - BOX(xorigin yorigin, (xorigin + 1485) (yorigin + 911)) -> BOX(33863.73046875 777606.3125,35348.73046875 778517.3125)
- Next take this box and use it as a paint brush to paint across and then down by translating it hor.n*1485, ver.n*911

```
CREATE TABLE throwaway_grid(gid SERIAL PRIMARY KEY);
SELECT AddGeometryColumn('public', 'throwaway_grid', 'the_geom', 26986, 'POLYGON', 2);

INSERT INTO throwaway_grid(the_geom)
SELECT ST_translate(ref.boxrep, hor.n*width, ver.n*height) As slice
FROM
(SELECT ST_xmin(ST_Extent(the_geom)) As xstart, ST_xmin(ST_Extent(the_geom)) as ystart, ST_SetSRID
(CAST('BOX(33863.73046875 777606.3125,35348.73046875 778517.3125)' as box2d), 26986) as boxrep,
ceiling((ST_xmax(ST_Extent(the_geom)) - ST_xmin(ST_Extent(the_geom)))/200) as width,
    ceiling((ST_ymax(ST_Extent(the_geom)) - ST_ymin(ST_Extent(the_geom)))/200) as height
FROM towns) As ref, generate_series(1,200) as hor(n), generate_series(1,200) as ver(n);

CREATE INDEX idx_throwaway_grid_the_geom ON throwaway_grid USING gist(the_geom);
```



The resulting image of this looks like this.
Take a look at Map Dicing to see an example of how you can use this in conjunction with ST_Interseciton to dice up maps into smaller pieces.

The translate function can be used in conjunction with other functions such as Rotate to do interesting things like rotate a geometry about a point or create ellipses such as described here

Boston GIS Copyright 2007 Paragon Corporation

This Document is available under the GNU Free Documentation License 1.2 http://www.gnu.org/copyleft/fdl.html & for download at the BostonGIS site http://www.bostongis.com