Developing Applications with PostGIS
Mapserver User Meeting

Presented By: Dave Blasby (dblasby@refractions.net) &
Chris Hodgson (chodgson@refractions.net)
Refractions Research Inc.
209 – 560 Johnson Street
Victoria, BC, V8W-3C6
pramsey@refractions.net
Phone: (250) 885-0632
Fax: (250) 383-2140

June 7, 2003

1 GOALS

After completing this workshop, you will have gained the skills to:

- create a PostGIS database
- create a geometry table with indexes to use with Mapserver
- insert and manipulate PostGIS geometries using SQL
- display the contents of a PostGIS table using Mapserver
- use spatial operators and functions to select specific geometries from a PostGIS table and display them in Mapserver
- use SQL joins to select data from multiple PostGIS tables for display in Mapserver

2 CREATING A POSTGIS DATABASE

Type the following commands in the command prompt to create a new PostGIS database:

```
createdb [dbname]

createlang plpgsql [dbname]

psql [dbname] < postgis.sql
```

The first line is a standard PostgreSQL command to create a new database. The "createlang" command adds the ability to use functions written in 'plpgsql' to the new database. The final "psql" command runs the SQL interpreter on the postgis.sql file.

This file contains commands that add the new "geometry" data type, the functions and operators that use it, and two extra tables that support the PostGIS system. These two tables are the geometry_columns table, for keeping track of the geometry columns and their constraints, and the spatial_ref_sys table for keeping track of the different spatial referencing systems (cartographic projection parameterizations) that the geometries' coordinates are stored in.

3 CREATING A POSTGIS TABLE

First you will need to connect to your newly created database using the psql SQL command interpreter:

```
psql [dbname]
```

To create a table, type the following into your psql prompt:

```
CREATE TABLE test ( id INTEGER PRIMARY KEY, name VARCHAR(20) NOT
NULL );

SELECT AddGeometryColumn ( '[dbname]', 'test', 'geom', -1,
'GEOMETRY', 2 );
```

The AddGeometryColumn function is a new function added by PostGIS, and requires a little bit of explaining. The function adds a new column to the specified table, and also a new row to the geometry_columns table, in order for the PostGIS system to keep track of it. The parameters are (in order):

- name of the database
- name of the table
- name of the new column to be created
- Spatial Referencing Identifier (SRID) of the new column
  - The SRID number references a row in the spatial_ref_sys table, and all the geometries in one column are required to have the same SRID.
- type of geometry to allow in the column, can be one of:
  - POINT
  - LINESTRING
  - POLYGON
  - MULTIPOINT
  - MULTILINESTRING
  - MULTIPOLYGON
  - GEOMETRY (all types allowed)
- number of dimensions to allow in the column (2 or 3)
  - Actual 3-dimensional geometries cannot be modeled, the geometry must still be legal in two dimensions - three dimensions simply allow a z-coordinate to be stored for each point.

4 CREATING A SPATIAL INDEX

A spatial index is used to speed up bounding-box based queries. This sort of query, for example "select all geometries which are within this bounding box", is used primarily to select the geometries within an area of interest, or within the display area of a map. Since this is by far the most popular query performed in web-based mapping, it is very important that the set of geometries can be selected quickly.

To create an R-Tree spatial index on your newly created PostGIS table, type the following into the psql prompt:

```
CREATE INDEX test_geom_idx ON test USING GIST (geom
GIST_GEOMETRY_OPS);
```

This command can take a long time if there is a significant amount of data in your table, however, since our test table is brand new, it will take no time at all. If you are going to be loading a large amount of data into a table, it is more efficient to insert the geometries first, and build the index afterward. Either way, once your data is loaded, it is a good idea to run the command:

```
VACUUM ANALYZE;
```

This should update the database's cache of statistics and possibly reorganize the structures of some indexes to increase the speed of queries using those indexes.

5 INSERTING DATA INTO POSTGIS

We will insert some data into PostGIS using the SQL command line, in order to help us understand the syntax for describing geometries. PostGIS uses the Well-Known Text (WKT) format for describing geometries – here are some examples of how to insert data into your new table:

```
INSERT INTO test ( id, name, geom )
   VALUES ( 1, 'geom 1', GeometryFromText( 'POINT(1 1)', -1 ) );

INSERT INTO test ( id, name, geom )
   VALUES ( 2, 'geom 2',
      GeometryFromText( 'LINESTRING(1 2, 2 3)', -1 ) );

INSERT INTO test ( id, name, geom )
   VALUES ( 3, 'geom 3',
      GeometryFromText( 'POLYGON((2 1, 3 1, 3 3, 2 1))', -1 ) );
```

The function "GeometryFromText" takes a WKT string and an SRID and returns a geometry. Here are a few more examples of the WKT for different types of geometries:

```
POINT(0 0)

LINESTRING(0 0,1 1,1 2)

POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1))

MULTIPOINT(0 0,1 2)

MULTILINESTRING((0 0,1 1,1 2),(2 3,3 2,5 4))

MULTIPOLYGON(((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)),((-1
-1,-1 -2,-2 -2,-2 -1,-1 -1)))

GEOMETRYCOLLECTION(POINT(2 4),LINESTRING(2 3,3 4))
```

6 DISPLAYING A POSTGIS TABLE IN MAPSERVER

One relatively easy way to look at the data you have now inserted into your PostGIS table is to use Mapserver. The following layer definition shows how to display the data from your "test" PostGIS table as a layer using Mapserver:

```
LAYER
     CONNECTIONTYPE postgis
     NAME "test"
     CONNECTION "user=theuser password=thepass dbname=thedb
host=theserver"
     DATA "geom FROM test"
     STATUS ON
     TYPE LINE
     CLASS
          COLOR 0 0 0
     END
END
```

The example map-file provided includes a similar layer definition; you should only need to edit the connection string in order to get it to work. In addition, make sure you set your map extents to default to reasonable values in order to see the data you inserted.

Note that point objects won't be displayed if your layer is defined to be of type "line", and polygons also will not be filled – you can change the type to point or polygon in order to see this. Normally, you would define a table to store only one type of geometry – you can do this with the "AddGeometryColumn" function.

7 IMPORTING SHAPEFILES INTO POSTGIS

Next we will create a new PostGIS table by importing data from an ESRI shapefile, using the "shp2pgsql" program:

```
shp2pgsql -D [shapefile] [tablename] [dbname] | psql [dbname]
```

The "shp2pgsql" program takes a shapefile and outputs the appropriate SQL to create a table with the same attributes (columns) as the shapefile, and to insert all of the records from the shapefile into PostGIS. The "-D" option makes shp2pgsql use the PostgreSQL "dump" format for inserting the rows, rather than standard SQL insert statements. The dump format simply loads faster, which is why we are using it. Shp2pgsl can also insert the rows into an existing table with the correct columns (instead of creating a new table), in order to aggregate multiple shapefiles with the same schema into a single table. Note that the column containing the geometry from the shapefile is called "the_geom" in the database table created by shp2pgsql.

To load the sample data, run the following commands from the shell prompt:

```
shp2pgsql –D victoria_elections.shp elections [dbname] | psql
[dbname]

shp2pgsql –D victoria_ocean.shp ocean [dbname] | psql [dbname]

shp2pgsql –D victoria_roads.shp roads [dbname] | psql [dbname]

shp2pgsql –D victoria_hospitals.shp hospitals [dbname] | psql
[dbname]
```

This will create four new tables in your database, named elections, ocean, roads andhospitals. We will be using the data in these tables in the further examples.

8 PREPARING TABLES FOR USE WITH MAPSERVER

Since we are going to be displaying the data in these new tables using Mapserver, we should create spatial indexes on these tables using the following SQL commands:

```
CREATE INDEX elections_geom_idx ON elections
    USING GIST (the_geom GIST_GEOMETRY_OPS);

CREATE INDEX ocean_geom_idx ON ocean
    USING GIST (the_geom GIST_GEOMETRY_OPS);

CREATE INDEX roads_geom_idx ON roads
    USING GIST (the_geom GIST_GEOMETRY_OPS);

CREATE INDEX hospitals_geom_idx ON hospitals
    USING GIST (the_geom GIST_GEOMETRY_OPS);
```

In addition, if we want to do feature queries on these tables, they will go faster if there is a primary key on them:

```
ALTER TABLE elections ADD PRIMARY KEY( gid );

ALTER TABLE ocean ADD PRIMARY KEY( gid );

ALTER TABLE roads ADD PRIMARY KEY( gid );

ALTER TABLE hospitals ADD PRIMARY KEY( gid );
```

It isn't necessary to use the gid column as the primary key – it is just handy, as it comes from the shapefile as a unique identifier. If your data didn't come from a shapefile, you would probably already have a primary key, and there is no reason to change it. If your table doesn't have a unique identifier, and you don't want to add one (it is usually a good idea to have a unique identifier), then you can use the "object identifier" provided by PostgreSQL.

The OID is a sort of "hidden column" that every table has and each row has a unique identifier in it. You can add a unique index on the OID of your table and use the OID to select a feature when doing queries. There will be more on this later…

Your data should now be ready to look at in Mapserver. The example mapfile provided includes layer definitions for all of these layers.

9 AN SQL QUERY – HOW MANY VOTED?

The data we are currently displaying in Mapserver could just as easily (perhaps more so) be displayed directly from the original shapefiles. However, remember that the data is coming out of a live, transactional database, so someone could be making changes as you are displaying the data, and those changes would be visible instantly on your map or through feature queries. The database also gives us a lot more flexibility – here is an example of something that you couldn't do with Mapserver and shapefiles.

The "elections_themed" layer in the example mapfile has the following DATA definition:

```
DATA "the_geom from
  (SELECT gid, the_geom, vtotal::REAL / vregist::REAL
   AS percent FROM elections)
as foo using srid=-1 using unique gid"
```

This sub-select statement allows us to generate a table on the fly, from which to display the features/geometries. The new table we are creating, in this case, has a column called "percent" which is the percentage of registered voters (vregist) who actually voted (vtotal). This number wasn't in the original table, but it is of interest, and we want to theme our map of the electoral districts using this number. Notice that there are several class definitions that have expressions that use [percent] to refer back to this newly calculated value.

Usually, Mapserver automatically determines the SRID of the table being displayed using the "geometry_columns" table and it uses the OID of each feature to identify it for feature querying. However, because we are generating this new table "on-the-fly", we need to provide some extra information about it. The "using SRID=-1" clause tells Mapserver that the SRID of this dynamically generated table (or view) is –1. This isn't particularly important because we aren't doing any coordinate re-projection, however it would be important if we were using several different coordinate systems in the mapfile and in the database. Regardless, it must be supplied.

In addition, the "using unique gid" clause tells Mapserver that the gid column is the one to use to uniquely identify a feature. Since our table is a relatively simple derivative of the original elections table, both of the "using" clauses are easy to provide. The phrase "as foo" simply names our dynamic table "foo" for the purposes of the rest of the query – but we never use it again. This is just an additional, required bit of syntax.

The result of this layer definition is a map themed by percentage of active voters; the darkest areas have the lowest percentage of voters, and the lightest areas have the most voters. This is more exciting if you remember that all of these numbers could be updated and displayed on the map in real-time.

10 A SPATIAL FUNCTION "HOW FAR IS THE NEAREST HOSPITAL?"

The previous query does a simple mathematical calculation to generate values for the new table. This time, we will use a spatial function to calculate the new table. Let's color-code the roads based on how far they are from the nearest hospital. Here is the data definition that will do it:

```
DATA "the_geom from
   (SELECT
       roads.the_geom AS the_geom,
       roads.gid AS gid,
       min(distance(roads.the_geom, hospitals.the_geom)) AS dist
    FROM roads, hospitals
    GROUP BY roads.oid, roads.the_geom)
as foo using srid=-1 using unique gid"
```

The "distance()" function takes two geometries as arguments and calculates the distance between them. The query calculates the distance between each road segment and all of the hospitals, and keeps the minimum distance for each road segment – calling it "dist". The "using" clauses are again very simple, but required.

11 AN SQL JOIN ROAD NETWORK LABELING

Now we are going to show how to do a simple join to display values from multiple tables. First we need to imagine that for some reason (probably normalization) some of the data in our roads table is actually stored in another table – the roads_attr table. First we need to create this table:

```
CREATE TABLE roads_attr AS
   SELECT gid, street, fromleft, toleft, fromright, toright,
type
   FROM roads;
CREATE INDEX roads_attr_gid_idx ON roads_attr( gid );
```

The roads_attr table now has all of the attribute information for each road feature, while the roads table has only the geometry and the gid (we will imagine). In order to display labels on our roads, we need to join the roads table to the roads_attr table. Here is the DATA definition that does it:

```
DATA "the_geom from
  (SELECT roads.gid as gid,
          roads.the_geom as the_geom,
          roads_attr.street
   FROM roads LEFT JOIN roads_attr ON roads.gid =
roads_attr.gid)
as foo using SRID=-1, using unique gid"
```

This statement gets the labels - "street" - from the roads_attr table, and the geometries - "the_geom" - from the roads table. Mapserver uses the geometry to decide where to place the labels. This definition is used in the "road_labels" layer of the example mapfile.

Notice that each road segment is labeled, not each road. Labeling all of the segments often causes unnecessary repetitions, and the repeated labels often cause some of the roads not to be labeled at all (due to label collisions with other, sometimes repeated, labels). The solution follows.

12 SOME TRICKINESS
NICER ROAD NETWORK LABELING

This last example uses neither a "spatial" function, nor a "normal" function. The "collect" function provided in PostGIS allows you to clump a bunch of geometries together into a single geometry. For example, if you have a bunch of lines, you can use "collect" to group them together into a multi-line. This is similar to the built-in SQL "sum" function for numeric types. Here is the data definition for nicer road labels:

```
DATA "the_geom from
     (SELECT street, collect(the_geom) as the_geom
      FROM roads
      GROUP BY street)
as foo using SRID=-1 using unique street"
```

This collects all of the geometries with the same road name into a single geometry, ensuring that it is only labeled once. This DATA definition is used in the "road_labels_grouped" layer of the example mapfile. We are ignoring the roads_attr table for this example because it would over-complicate things (but it could still be done – this is left as "an exercise for the reader").

13 FOR MORE INFORMATION

If you are interested in learning more about PostGIS, the best place to visit is our web site:

http://postgis.refractions.net

You may also be interested in joining the PostGIS Users mailing list:

postgis_users@postgis.refractions.net

We'd also like to hear your feedback on the workshop – contact chodgson@refractions.net or dblasby@refractions.net to let us know what you liked and what else you'd like to see in future workshops.

See you at the next MUM!