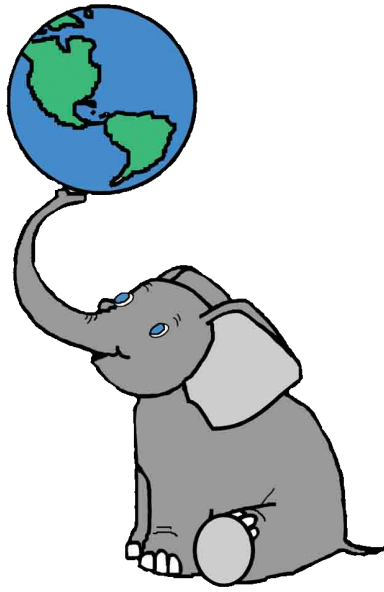


# Introduction to PostGIS

**Prepared By:** Paul Ramsey  
Refractions Research Inc.  
400 - 1207 Douglas Street  
Victoria, BC, V8W-2E7  
[pramsey@refractions.net](mailto:pramsey@refractions.net)  
Phone: (250) 383-3022  
Fax: (250) 383-2140



# Table of Contents

---

<b><u>1</u></b>	<b><u>SUMMARY</u></b>	<b>3</b>
1.1	<a href="#">Requirements</a>	3
1.2	<a href="#">Conventions</a>	3
1.3	<a href="#">Downloads</a>	3
<b><u>2</u></b>	<b><u>POSTGRESQL / POSTGIS SETUP</u></b>	<b>4</b>
2.1	<a href="#">Initialize the Data Area</a>	4
2.2	<a href="#">Start the Database Server</a>	5
2.3	<a href="#">Create a Working Database</a>	6
2.4	<a href="#">Load the PostGIS Extension</a>	7
<b><u>3</u></b>	<b><u>USING POSTGIS</u></b>	<b>8</b>
3.1	<a href="#">Simple Spatial SQL</a>	8
3.2	<a href="#">Loading Shape File Data</a>	9
3.3	<a href="#">Creating Indexes</a>	10
3.4	<a href="#">Using Indexes</a>	11
3.5	<a href="#">Indexes and Query Plans</a>	12
3.6	<a href="#">PostgreSQL Optimization</a>	14
3.7	<a href="#">Spatial Analysis in SQL</a>	15
3.8	<a href="#">Data Integrity</a>	16
3.9	<a href="#">Distance Queries</a>	16
3.10	<a href="#">Spatial Joins</a>	17
3.11	<a href="#">Overlays</a>	18
3.12	<a href="#">Coordinate Projection</a>	19
<b><u>4</u></b>	<b><u>EXERCISES</u></b>	<b>20</b>
4.1	<a href="#">Basic Exercises</a>	20
4.2	<a href="#">Advanced Exercises</a>	21

1

# Summary

---

PostGIS is a spatial database add-on for the PostgreSQL relational database server. It includes support for all of the functions and objects defined in the OpenGIS “Simple Features for SQL” specification. Using the many spatial functions in PostGIS, it is possible to do advanced spatial processing and querying entirely at the SQL command-line.

This workshop will cover

- installation and setup of PostgreSQL,
- installation of the PostGIS extension,
- loading of sample data,
- indexing,
- performance tuning,
- spatial SQL basics, and
- spatial SQL best practices.

## 1 Requirements

This workshop will use the new Windows native PostgreSQL, and a data package of shape files, which should be installed on your machines already. For those following from home, unzip the installation package zip file in C: and continue.

## 2 Conventions

Interactive sessions will be presented in this document in grey boxes, with the text to be entered in **boldface**, and the results in normal text. In general, directions to be performed will be presented in **boldface**.

## 3 Downloads

The presentation materials, data and installation packages used in this workshop can be downloaded from <http://postgis.refractions.net/>.

The PostgreSQL source code is available from <http://www.postgresql.org/>.

The PostGIS source code is available from <http://postgis.refractions.net/>.

The GEOS source code is available from <http://geos.refractions.net/>.

The Proj4 source code is available from <http://proj.maptools.org/>.

The PgAdmin administration tool is available from <http://www.pgadmin.org>.

2

## PostgreSQL / PostGIS Setup

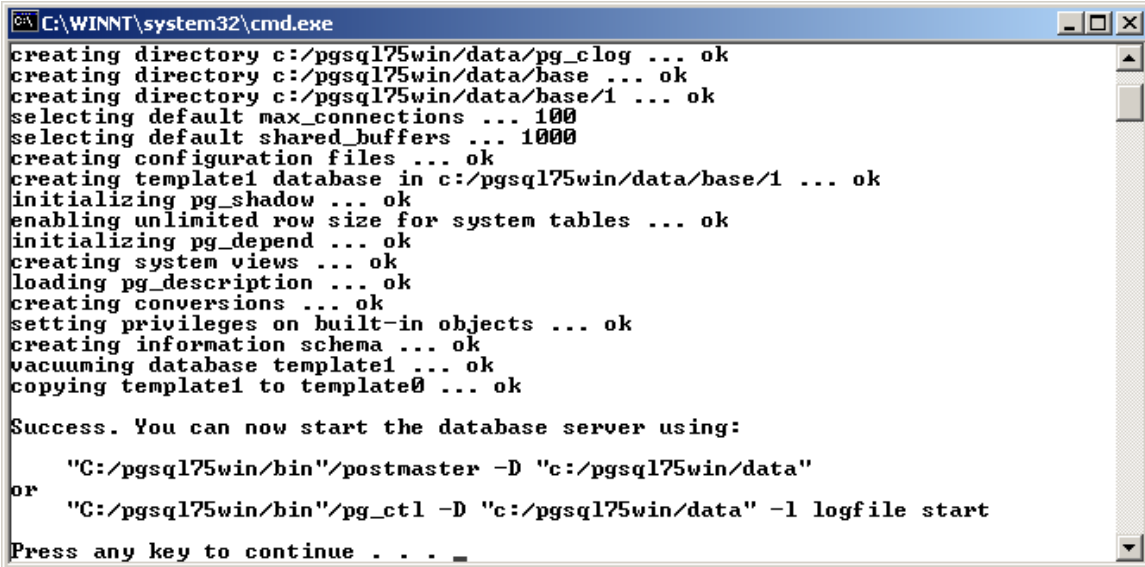
The contents of the PostgreSQL directory are:

\bin	(executables)
\include	(include files)
\lib	(shared DLL libraries)
\share	(extensions)
env.bat	(command prompt with correct environment)
initdb.bat	(initialize \data database directory)
pgstart.bat	(start database server)

### 4 Initialize the Data Area

All PostgreSQL database servers can serve multiple databases out of a single database area. Initializing the database area copies standard system files into the area, suitable for starting a database server on with only template databases to start with.

**Double-click the initdb.bat file to initialize the database directory.**



```
C:\WINNT\system32\cmd.exe
creating directory c:/pgsql75win/data/pg_clog ... ok
creating directory c:/pgsql75win/data/base ... ok
creating directory c:/pgsql75win/data/base/1 ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 1000
creating configuration files ... ok
creating template1 database in c:/pgsql75win/data/base/1 ... ok
initializing pg_shadow ... ok
enabling unlimited row size for system tables ... ok
initializing pg_depend ... ok
creating system views ... ok
loading pg_description ... ok
creating conversions ... ok
setting privileges on built-in objects ... ok
creating information schema ... ok
vacuuming database template1 ... ok
copying template1 to template0 ... ok

Success. You can now start the database server using:

    "C:/pgsql75win/bin"/postmaster -D "c:/pgsql75win/data"
or
    "C:/pgsql75win/bin"/pg_ctl -D "c:/pgsql75win/data" -l logfile start

Press any key to continue . . . _
```

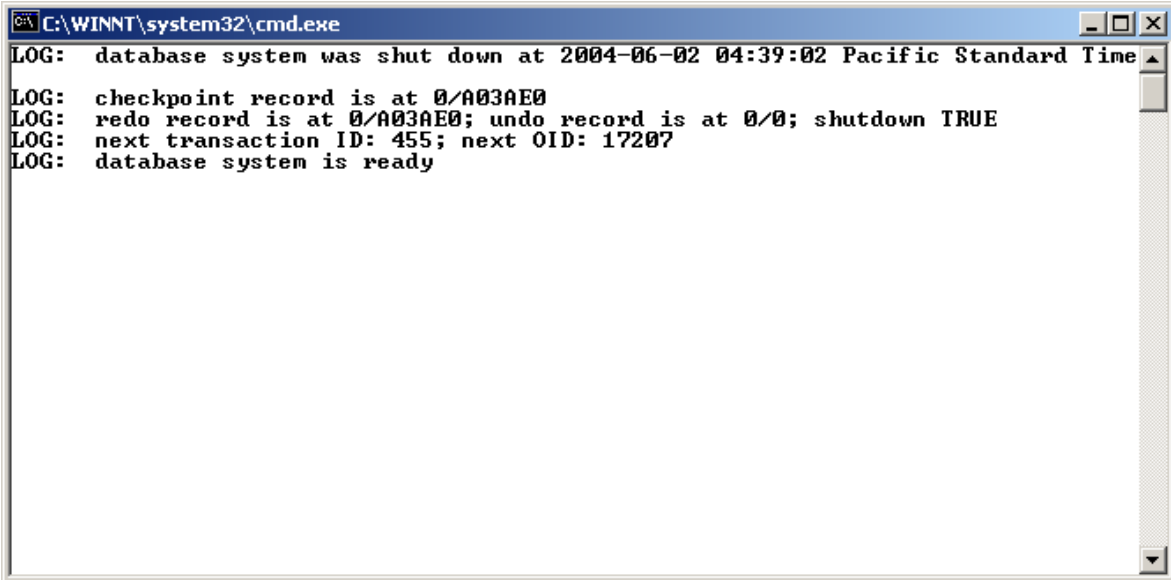
There will be a new \data directory created under your PostgreSQL directory.

### 5 Start the Database Server

The database server must be kept running throughout the workshop. For a

production database setup, the database server would be started as a service, not in a terminal window. However, having the database in a terminal window allows us to see the database LOG, NOTICE and ERROR messages, which is handy.

**Double-click the pgstart.bat file to start the database server.**



```
C:\WINNT\system32\cmd.exe
LOG:  database system was shut down at 2004-06-02 04:39:02 Pacific Standard Time
LOG:  checkpoint record is at 0/A03AE0
LOG:  redo record is at 0/A03AE0; undo record is at 0/0; shutdown TRUE
LOG:  next transaction ID: 455; next OID: 17207
LOG:  database system is ready
```

You now have a PostgreSQL server running.

**6**

## Create a Working Database

The rest of the steps in this workshop will be run inside the command prompt window.

**Double-click the env.bat file to get a command prompt.**

```
C:\pgsql75win> createdb postgis
CREATE DATABASE

c:\pgsql75win> psql postgis
Welcome to psql 7.5devel, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

Warning: Console codepage (437) differs from windows codepage (1252)
        8-bit characters will not work correctly. See PostgreSQL
        documentation "Installation on Windows" for details.

postgis=#
```

You are now connected to your PostgreSQL database! Ignore the “Warning” message, it is a side effect of using the brand new PostgreSQL Windows version, but will not effect the operation of the software.

Try out the database by creating a test table, inserting some data, selecting it out, and dropping the table.

```
postgis=# create table test ( id integer, name varchar );
CREATE TABLE
postgis=# insert into test values ( 1, 'foo' );
INSERT 17213 1
postgis=# select * from test;
 id | name
----+-----
  1 | foo
(1 row)

postgis=# drop table test;
DROP TABLE
postgis=# \q
```

## 7 Load the PostGIS Extension

The PostGIS extension requires the PL/PgSQL procedural language for some of its functions, so the first step is to create that language.

Then we change to the extensions directory, and load the postgis.sql file, that binds the functions in the postgis library to SQL objects and functions.

Finally, we add the spatial reference system definitions.

Now we are ready to connect to the spatial database and start doing some spatial work.

```
C:\pgsql75win> createlang plpgsql postgis

C:\pgsql75win> cd share\contrib

C:\pgsql75win\share\contrib> psql -f postgis.sql postgis
BEGIN
CREATE FUNCTION
. . . . .
COMMIT

C:\pgsql75win\share\contrib> psql -f spatial_ref_sys.sql postgis
BEGIN
INSERT 19254
. . . . .
COMMIT

C:\pgsql75win\share\contrib> psql postgis
```

## 1 UNIX Note

The PostGIS installation steps are the same for Linux/UNIX as they are for Windows. The PL/PgSQL language must be created, the postgis.sql file loaded, and the spatial reference definitions loaded. The commands are all the same, and are run at the UNIX command line.

## 3



# Using PostGIS

## 8 Simple Spatial SQL

Now we will test creating a table with a geometry column, adding some spatial objects to the table, and running a spatial function against the table contents.

```
postgis=# create table points ( pt geometry, name varchar );
CREATE TABLE
postgis=# insert into points values ( 'POINT(0 0)', 'Origin' );
INSERT 19269 1
postgis=# insert into points values ( 'POINT(5 0)', 'X Axis' );
INSERT 19270 1
postgis=# insert into points values ( 'POINT(0 5)', 'Y Axis' );
INSERT 19271 1
postgis=# select name, AsText(pt), Distance(pt, 'POINT(5 5)')
        from points;
```

name	astext	distance
Origin	POINT(0 0)	7.07106781186548
X Axis	POINT(5 0)	5
Y Axis	POINT(0 5)	5

(3 rows)

```
postgis=# drop table points;
DROP TABLE
```

Note that there are two spatial database functions being used in the example above: `Distance()` and `AsText()`. Both functions expect geometry objects as arguments. The `Distance()` function calculates the minimum Cartesian distance between two spatial objects. The `AsText()` function turns geometry into a simple textual representation, called “Well-Known Text”.

### 1 Examples of Well-Known Text

```
POINT(1 1)
MULTIPOINT(1 1, 3 4, -1 3)
LINESTRING(1 1, 2 2, 3 4)
POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))
MULTIPOLYGON((0 0, 0 1, 1 1, 1 0, 0 0), (5 5, 5 6, 6 6, 6 5, 5 5))
MULTILINESTRING((1 1, 2 2, 3 4),(2 2, 3 3, 4 5))
```

## 9 Loading Shape File Data

Now we will load our example data (**C:\osgis-data\postgis**) into the database.

The sample data is in Shape files, so we will need to convert it into a loadable format using the `shp2pgsql` tool and then load it into the database.

Our data is in projected coordinates, the projection is “BC Albers” and is stored in the `SPATIAL_REF_SYS` table as SRID 42102. When we create the loadable format, we will specify the SRID on the command line, so that the data is correctly referenced to a coordinate system. This will be important later when we try the coordinate reprojection functionality.

We can either create the load format as a file, then load the file with `psql`, or pipe the results of `shp2pgsql` directly into the `psql` terminal monitor. We will use the first option for the `bc_pubs` data, and second option for the rest of the data files.

```
C:\pgsql75win> cd \osgis-data\postgis

C:\osgis-data\postgis> dir *.shp
Directory of C:\osgis-data\postgis>
06/02/2004  01:47p           1,332 bc_hospitals.shp
06/02/2004  12:05p       278,184 bc_municipality.shp
06/02/2004  01:22p           10,576 bc_pubs.shp
06/02/2004  12:17p    19,687,612 bc_roads.shp
03/09/2004  03:28p    8,359,672 bc_voting_areas.shp
                5 File(s)      29,430,800 bytes

C:\osgis-data\postgis> shp2pgsql -s 42102 bc_pubs.shp bc_pubs >
bc_pubs.sql
C:\osgis-data\postgis> psql -f bc_pubs.sql postgis
BEGIN
INSERT 215525 1
. . . . .
COMMIT

C:\osgis-data\postgis> shp2pgsql -s 42102 bc_roads.shp bc_roads |
psql postgis
C:\osgis-data\postgis> shp2pgsql -s 42102 bc_hospitals.shp
bc_hospitals | psql postgis
C:\osgis-data\postgis> shp2pgsql -s 42102 bc_municipality.shp
bc_municipality | psql postgis
C:\osgis-data\postgis> shp2pgsql -s 42102 bc_voting_areas.shp
bc_voting_areas | psql postgis
```

## 10 Creating Spatial Indexes

Indexes are extremely important for large spatial tables, because they allow queries to quickly retrieve the records they need. Since PostGIS is frequently

used for large data sets, learning how to build and (more importantly) how to use indexes is key.

PostGIS indexes are R-Tree indexes, implemented on top of the general GiST (Generalized Search Tree) indexing schema. R-Trees organize spatial data into nesting rectangles for fast searching. (For the seminal paper, see <http://postgis.refractions.net/rtree.pdf>)

```
C:\pgsql75win> psql postgis

postgis=# create index bc_roads_gidx on bc_roads using gist
( the_geom gist_geometry_ops );
CREATE INDEX

postgis=# create index bc_pubs_gidx on bc_pubs using gist
( the_geom gist_geometry_ops );
CREATE INDEX

postgis=# create index bc_voting_areas_gidx on bc_voting_areas using
gist ( the_geom gist_geometry_ops );
CREATE INDEX

postgis=# create index bc_municipality_gidx on bc_municipality using
gist ( the_geom gist_geometry_ops );
CREATE INDEX

postgis=# create index bc_hospitals_gidx on bc_hospitals using gist
( the_geom gist_geometry_ops );
CREATE INDEX

postgis=# \d bc_roads
```

Run a table describe on `bc_roads` (`\d bc_roads`) and note the index summary at the bottom of the description. There should be two indexes, the primary key index and the new “gist” index you just created.

Now, clean up your database and update the index selectivity statistics (we will explain these in more detail in a couple sections).

```
postgis=# VACUUM ANALYZE;
```

11

## Using Spatial Indexes

It is important to remember that spatial indexes are not used automatically for every spatial comparison or operator. In fact, because of the “rectangular” nature of the R-Tree index, spatial indexes are only good for bounding box comparisons.

This is why all spatial databases implement a “two phase” form of spatial processing.

- The first phase is the indexed bounding box search, which runs on the whole table.
- The second phase is the accurate spatial processing test, which runs on just the subset returned by the first phase.

In PostGIS, the first phase indexed search is activated by using the “&&” operator. “&&” is a symbol with a particular meaning. Just like “=” means “equals”, “&&” means “bounding boxes overlap”. After a little while, using the “&&” operator will become second nature.

Let’s compare the performance of a query that uses a two-phase index strategy and a query that does not.

First, time the non-indexed query (this looks for the roads that cross a supplied linestring, the example linestring is constructed so that only one road is returned):

```
postgis=# select gid, name from bc_roads where crosses(the_geom,
GeomFromText('LINESTRING(1220446 477473,1220417 477559)', 42102));
gid | name
-----+-----
64555 | Kitchener St
(1 row)
```

Now, time the two-phase strategy, that includes an index search as well as the crossing test:

```
postgis=# select gid, name from bc_roads where the_geom &&
GeomFromText('LINESTRING(1220446 477473,1220417 477559)', 42102) and
crosses(the_geom, GeomFromText('LINESTRING(1220446 477473,1220417
477559)', 42102));
gid | name
-----+-----
64555 | Kitchener St
(1 row)
```

You will have to be pretty fast with your stopwatch to time the second query.



## Indexes and Query Plans

Databases are fancy engines for speeding up random access to large chunks of data. Large chunks of data have to be stored on disk, and disk access is (relatively speaking) very, very slow. At the core of databases are algorithms tuned to get as much data as possible with as few disk accesses as possible.

Query plans are the rules used by databases to convert a piece of SQL into a strategy for reading the data. In PostgreSQL, you can see the estimated query plan for any SQL query by prepending “EXPLAIN” before the query. You can see the actual observed performance by prepending “EXPLAIN ANALYZE” before the query.

**Hit the Up Arrow to recall a previous query.**  
**Hit Ctrl-A to move the cursor to the front of the query.**  
**Add the EXPLAIN clause and hit enter.**

```
postgis=# explain select gid, name from bc_roads where
crosses(the_geom, GeomFromText('LINESTRING(1220446 477473,1220417
477559)', 42102));
          QUERY PLAN
-----
Seq Scan on bc_roads  (cost=0.00..7741.71 rows=54393 width=17)
  Filter: crosses(the_geom, 'SRID=42102;LINESTRING(1220446
477473,1220417 477559)::geometry)
(2 rows)
```

No “&&” operator, so no index scan. The query has to test every row in the table.

```
postgis=# explain select gid, name from bc_roads where the_geom &&
GeomFromText('LINESTRING(1220446 477473,1220417 477559)', 42102) and
crosses(the_geom, GeomFromText('LINESTRING(1220446 477473,1220417
477559)', 42102));
          QUERY PLAN
-----
Index Scan using bc_roads_gidx on bc_roads  (cost=0.00..6.02 rows=1
width=17)
  Index Cond: (the_geom && 'SRID=42102;LINESTRING(1220446
477473,1220417 477559)::geometry)
  Filter: crosses(the_geom, 'SRID=42102;LINESTRING(1220446
477473,1220417 477559)::geometry)
(3 rows)
```

With the index scan, the query only has to text a few rows using the crosses filter.



## When Query Plans Go Bad

A database can only use one index at a time to retrieve data.

So, if you are making a query using a filter that specifies two indexed columns, the database will attempt to pick the index with the “greatest selectivity”. That is, the index that returns the fewest rows. It does this by using statistics gathered from the table and indexes about the makeup of the data.

As of PostgreSQL version 7.5, the spatial selectivity analysis for PostGIS indexes is integrated into the main selectivity system of PostgreSQL. So, to ensure that your statistics are kept up to date as you change your data, you just have to run the “ANALYZE” command occasionally. You must compile PostGIS with USE\_STATS enabled to have this functionality (this is the default).

Prior to PostgreSQL 7.4, the selectivity analysis had to be run separately, with a custom SQL function: “SELECT UPDATE\_GEOMETRY\_STATS();”

Without selectivity, PostGIS is tuned to force the query planner to always use the spatial index. This can be a terrible strategy, if the index filter is very large – if, for example, your query is “give me all the streets named ‘Bob’ in the south half of the province”. Using the index on street name will be much more selective (assuming there are not many streets named Bob in the province) than the spatial index (there are hundreds of thousands of streets in the south half of the province).

If the spatial index is (incorrectly) chosen, the database will have to sequence scan every road from the south of the province to see if it is named ‘Bob’, instead of sequence scanning every road named ‘Bob’ to see if it is in the south half of the province.

**13**



## PostgreSQL Optimization

PostgreSQL is shipped by the development team with a conservative default configuration. The intent is to ensure that PostgreSQL runs without modification on as many common machines as possible. That means if your machine is uncommonly good (with a large amount of memory, for example) the default configuration is probably too conservative.

True database optimization, particularly of databases under transactional load, is a complex business, and there is no one size fits all solution. However, simply boosting a few PostgreSQL memory use parameters can increase performance.

The database configuration file is in the database's `\data` area, and is named `postgresql.conf`.

Open `postgresql.conf` in any text editor. The following lines offer quick boosts if you have the hardware:

```
shared_buffers = 1000          # min 16, 8KB each
```

Increase the shared buffers as much as possible, but not so much that you exceed the amount of physical memory available for you on the computer. After about a few hundred MB, there are diminishing returns. Since each buffer is 8Kb, you will have to do a little math to calculate your ideal buffer size.

For example, if you want 120MB of shared buffers, you will want to set a value of  $120 * 1024 / 8 = 15360$ .

Shared buffers are (surprise) shared between all PostgreSQL back-ends, so you do not have to worry about how many back-ends you spawn.

```
work_mem = 1024                # min 64, size in KB  
maintenance_work_mem = 16384  # min 1024, size in KB
```

Working memory is used for sorting and grouping. It is used per-backend, so you have to guesstimate how many back-ends you will have running at a time. The default value is only 1MB.

Maintenance memory is used for vacuuming the database. It is also used per-backend, but you are unlikely to have more than one backend vacuuming at a time (probably). The default value is 16MB, which might be enough, depending on how often you vacuum.

**NOTE:** Some optimizations, like `shared_buffers` have to be synchronized with operating system kernel tuning. For example, the default number of shared buffers allowed by Linux is fairly low, and that number needs to be increased before a tuned-up PostgreSQL will be able to run. See <http://www.postgresql.org/docs/7.4/static/kernel-resources.html>.



## Spatial Analysis in SQL

A surprising number of traditional GIS analysis questions can be answered using a spatial database and SQL. GIS analysis is generally about filtering spatial objects with conditions, and summarizing the results – and that is exactly what databases are really good at. GIS analysis also tests interactions between spatially similar features, and uses the interactions to answer questions – with spatial indexes and spatial functions, databases can do that too!

### 3 Exercises

These exercises will be easier if you first peruse the data dictionary and functions list for information about the data columns and available functions. You can enter these exercises in order, or for a challenge, cover up your page with another piece of paper and try to figure out the answer yourself before looking at the SQL.

“What is the total length of all roads in the province, in kilometers?”

```
postgres=# select sum(length(the_geom))/1000 as km_roads from
bc_roads;
 km_roads
-----
70842.1243039643
(1 row)
```

”How large is the city of Prince George, in hectares?”

```
postgres=# select area(the_geom)/10000 as hectares from
bc_municipality where name = 'PRINCE GEORGE';
 hectares
-----
32657.9103824927
(1 row)
```

”What is the largest municipality in the province, by area?”

```
postgres=# select name, area(the_geom)/10000 as hectares from
bc_municipality order by hectares desc limit 1;
 name | hectares
-----+-----
TUMBLER RIDGE | 155020.02556131
(1 row)
```

The last one is particularly tricky. There are several ways to do it, including a two step process that finds the maximum area, then finds the municipality that has that area. The suggested way uses the PostgreSQL “LIMIT” statement and a reverse ordering to pull just the top area.



## Data Integrity

PostGIS spatial functions require that input geometries obey the “Simple Features for SQL” specification for geometry construction. That means LINESTRINGS cannot self-intersect, POLYGONS cannot have their holes outside their boundaries, and so on. Some of the specifications are quite strict, and some input geometries may not conform to them.

The `isvalid()` function is used to test that geometries conform to the specification. This query counts the number of invalid geometries in the voting areas table:

```
postgis=# select count(*) from bc_voting_areas where not
isvalid(the_geom);
count
-----
      0
(1 row)
```

## 16 Distance Queries

It is easy to write inefficient queries for distances, because it is not always obvious how to use the index in a distance query. Here is a short example of a distance query using an index.

“How many BC Unity Party supporters live within 2 kilometers of the Tabor Arms pub in Prince George?”

First, we find out where the Tabor Arms is:

```
postgis=# select astext(the_geom) from bc_pubs where name ilike
'Tabor Arms%';
          astext
-----
POINT(1209385.41168654 996204.96991804)
(1 row)
```

Now, we use that location to pull all the voting areas within 2 kilometers and sum up the Unity Party votes:

```
postgis=# select sum(upbc) as unity_voters from bc_voting_areas where
the_geom && setsrid(expand('POINT(1209385 996204)::geometry, 2000),
42102) and distance(the_geom, geomfromtext('POINT(1209385 996204)',
42102)) < 2000;
unity_voters
-----
          421
(1 row)
```

## 17 Spatial Joins

A standard table join puts two tables together into one output result based on a common key. A spatial join puts two tables together into one output result based on a spatial relationship.

We will find the safest pubs in British Columbia. Presumably, if we are close to a hospital, things can only get so bad.

“Find all pubs located within 250 meters of a hospital.”

For clarity, we will do this query without an index clause, since the tables are so small.

```
postgres=# select h.name, p.name from bc_hospitals h,  
bc_pubs p where distance(h.the_geom, p.the_geom) < 250;
```

Just as with a standard table join, values from each input table are associated and returned side-by-side.

Using indexes, spatial joins can be used to do very large scale data merging tasks.

For example, the results of the 2000 election are usually summarized by riding – that is how members are elected to the legislature, after all. But what if we want the results summarized by municipality, instead?

“Summarize the 2000 provincial election results by municipality.”

```
postgres=# select m.name, sum(v.ndp) as ndp, sum(v.lib) as liberal,  
sum(v.gp) as green, sum(v.upbc) as unity, sum(v.vtotal) as total from  
bc_voting_areas v, bc_municipality m where v.the_geom && m.the_geom  
and intersects(v.the_geom, m.the_geom) group by m.name order by  
m.name;
```

name	ndp	liberal	green	unity	total
100 MILE HOUSE	398	959	0	70	1527
ABBOTSFORD	1507	9547	27	575	12726
ALERT BAY	366	77	26	0	500
ANMORE	247	1299	0	0	1644
ARMSTRONG	433	1231	199	406	2389
ASHCROFT	217	570	0	39	925
BELCARRA	93	426	37	0	588
BURNABY	15500	31615	8276	1218	58316
BURNS LAKE	370	919	104	92	1589
CACHE CREEK	76	323	0	27	489
CAMPBELL RIVER	2814	6904	1064	0	11191
. . . . .					



## Overlays

Overlays are a standard GIS technique for analyzing the relationship between two layers. Particularly where attribute are to be scaled by area of interaction between the layers, overlays are an important tool.

In SQL terms, an overlay is just a spatial join with an intersection operation. For each polygon in table A, find all polygons in table B that interact with it. Intersect A with all potential B's and copy the resultants into a new table, with the attributes of both A and B, and the original areas of both A and B. From there, attributes can be scaled appropriately, summarized, etc. Using sub-selects, temporary tables are not even needed – the entire overlay-and-summarize operation can be embedded in one SQL statement.

Here is a small example overlay that creates a new table of voting areas clipped by the Prince George municipal boundary:

```
postgis=# create table pg_voting_areas as select
intersection(v.the_geom, m.the_geom) as intersection_geom,
area(v.the_geom) as va_area, v.*, m.name from bc_voting_areas v,
bc_municipality m where v.the_geom && m.the_geom and
intersects(v.the_geom, m.the_geom) and m.name = 'PRINCE GEORGE';
SELECT

postgis=# select sum(area(intersection_geom)) from pg_voting_areas;
      sum
-----
326579103.824927
(1 row)

postgis=# select area(the_geom) from bc_municipality where name =
'PRINCE GEORGE';
      area
-----
326579103.824927
(1 row)
```

Note that the area of the sum of the resultants equals the area of the clipping feature, always a good sign.



## Coordinate Projection

PostGIS supports coordinate reprojection inside the database.

Every geometry in PostGIS has a “spatial referencing identifier” or “SRID” attached to it. The SRID indicates the spatial reference system the geometry coordinates are in. So, for example, all the geometries in our examples so far have been in the British Columbia Albers reference system.

You can view the SRID of geometries using the `sruid()` function:

```
postgis=# select sruid(the_geom) from bc_roads limit 1;
```

The SRID of 42102 corresponds to a particular entry in the `SPATIAL_REF_SYS` table. Because PostGIS uses the PROJ4 library for reprojection support, the `SPATIAL_REF_SYS` table includes a `PROJ4TEXT` column that gives the coordinate system definition in PROJ4 parameters:

```
postgis=# select proj4text from spatial_ref_sys where sruid=42102;
proj4text
-----
+proj=aea +ellps=GRS80 +datum=NAD83 +lat_0=45.0 +lon_0=-126.0
+lat_1=58.5 +lat_2=50.0 +x_0=1000000 +y_0=0
(1 row)
```

Coordinate reprojection is done using the `transform()` function, referencing an SRID that exists in the `SPATIAL_REF_SYS` table. For example, in the panel below we will view a geometry in the stored coordinates, then reproject it to geographic coordinates using the `transform()` function:

```
postgis=# select astext(the_geom) from bc_roads limit 1;
astext
-----
MULTILINESTRING((1004687.04355194 594291.053764096,1004729.74799931
594258.821943696,1004808.0184134 594223.285878035,1004864.93630072
594204.422638658,1004900.50302171 594200.005856311))
(1 row)

postgis=# select astext(transform(the_geom,4326)) from bc_roads limit
1;
astext
-----
MULTILINESTRING((-125.9341 50.3640700000001,-125.9335 50.36378,
-125.9324 50.36346,-125.9316 50.36329,-125.9311 50.36325))
(1 row)
```

Note the longitude/latitude coordinates in the transformed geometry.

4

## Exercises

---

### 20 Basic Exercises

“What is the perimeter of the municipality of Vancouver?”

```
postgis=# select perimeter(the_geom) from bc_municipality where name
= 'VANCOUVER';
      perimeter
-----
57321.7782018048
(1 row)
```

”What is the total area of all voting areas in hectares?”

```
postgis=# select sum(area(the_geom))/10000 as hectares from
bc_voting_areas;
      hectares
-----
107000936.980171
(1 row)
```

“What is the total area of all voting areas with more than 100 voters in them?”

```
postgis=# select sum(area(the_geom))/10000 as hectares from
bc_voting_areas where vtotal > 100;
      hectares
-----
41426649.2880661
(1 row)
```

”What is the length in kilometers of all roads named ‘Douglas St’?”

```
postgis=# select sum(length(the_geom))/1000 as kilometers from
bc_roads where name = 'Douglas St';
      kilometers
-----
19.8560819878386
(1 row)
```

### 21

## Advanced Exercises

”What is the length in kilometers of ‘Douglas St’ in Victoria?”

```
postgis=# select sum(length(r.the_geom))/1000 as kilometers from
bc_roads r, bc_municipality m where r.the_geom && m.the_geom and
r.name = 'Douglas St' and m.name = 'VICTORIA';
 kilometers
-----
 4.89151904172838
(1 row)
```

”What two pubs have the most Green Party supporters within 500 meters of them?”

```
postgis=# select p.name, p.city, sum(v.gp) as greens from bc_pubs p,
bc_voting_areas v where v.the_geom && setsrid(expand(p.the_geom,
500), 42102) and distance(v.the_geom, p.the_geom) < 500 group by
p.name, p.city order by greens desc limit 2;
 name | city | greens
-----+-----+-----
 Bimini's | Vancouver | 1407
 Darby D. Dawes | Vancouver | 1104
(2 rows)
```

”What is the latitude of the most southerly hospital in the province?”

```
postgis=# select y(transform(the_geom,4326)) as latitude from
bc_hospitals order by latitude asc limit 1;
 latitude
-----
 48.4657953714625
(1 row)
```

”What were the percentage NDP and Liberal vote within the city limits of Prince George in the 2000 provincial election?”

```
postgis=# select 100*sum(v.ndp)/sum(v.vtotal) as ndp,
100*sum(v.lib)/sum(v.vtotal) as liberal from bc_voting_areas v,
bc_municipality m where v.the_geom && m.the_geom and
intersects(v.the_geom, m.the_geom) and m.name = 'PRINCE GEORGE';
 ndp | liberal
-----+-----
 16 | 59
(1 row)
```

“What is the largest voting area polygon that has a hole?”

```
postgis=# select gid, id, area(the_geom) as area from bc_voting_areas
where nrings(the_geom) > 1 order by area desc limit 1;
gid | id | area
-----+-----+-----
3531 | NOC 60 | 32779957497.4976
(1 row)
```

“How many NDP voters live within 50 meters of ‘Simcoe St’ in Victoria?”

```
postgis=# select sum(v.ndp) as ndp from bc_voting_areas v,
bc_municipality m, bc_roads r where m.the_geom && r.the_geom and
r.name = 'Simcoe St' and m.name = 'VICTORIA' and distance(r.the_geom,
v.the_geom) < 50 and m.the_geom && v.the_geom;

ndp
-----
2558
(1 row)
```