

PostGIS Manual

Edited by
Paul Ramsey

PostGIS Manual

Edited by Paul Ramsey

PostGIS is an extension to the PostgreSQL object-relational database system which allows GIS (Geographic Information Systems) objects to be stored in the database. PostGIS includes support for GiST-based R-Tree spatial indexes, and functions for basic analysis of GIS objects.

Table of Contents

1. Introduction.....	7
1.1. Credits	7
1.2. More Information.....	7
2. Installation.....	9
2.1. Requirements	9
2.2. PostGIS	9
2.2.1. Upgrading	11
2.3. JDBC.....	12
2.4. Loader/Dumper	12
3. Frequently Asked Questions	15
4. Using PostGIS.....	21
4.1. GIS Objects	21
4.1.1. Standard versus Canonical Forms.....	21
4.2. Using OpenGIS Standards	23
4.2.1. The SPATIAL_REF_SYS Table	23
4.2.2. The GEOMETRY_COLUMNS Table	25
4.2.3. Creating a Spatial Table	26
4.3. Loading GIS Data	27
4.3.1. Using SQL	27
4.3.2. Using the Loader.....	28
4.4. Retrieving GIS Data.....	29
4.4.1. Using SQL	29
4.4.2. Using the Dumper	32
4.4.3. Using Minnesota Mapserver	33
4.5. Building Indexes	36
4.5.1. GiST Indexes.....	36
4.5.2. Using Indexes.....	37
4.6. Complex Queries	38

4.7. Java Clients (JDBC).....	38
4.8. C Clients (libpq).....	41
4.8.1. Text Cursors	41
4.8.2. Binary Cursors	41
5. PostGIS Reference	43
5.1. OpenGIS Functions.....	43
5.2. Other Functions.....	46

Chapter 1. Introduction

PostGIS is developed by Refrations Research Inc, as a spatial database technology research project. Refrations is a GIS and database consulting company in Victoria, British Columbia, Canada, specializing in data integration and custom software development. We plan on supporting and developing PostGIS to support a range of important GIS functionality, including full OpenGIS support, advanced topological constructs (coverages, surfaces, networks), desktop user interface tools for viewing and editing GIS data, and web-based access tools.

1.1. Credits

Dave Blasby <dblasby@refrations.net>

The principal developer of PostGIS. Dave maintains the server side objects and index support, the server side analytical functions, and the Mapserver connectivity.

Chris Hodgson <chodgson@refrations.net>

Maintains new functions and the 7.2 index bindings.

Paul Ramsey <pramsey@refrations.net>

Maintains the JDBC objects and keeps track of the documentation and packaging.

Jeff Lounsbury <jeffloun@refrations.net>

Maintains the Shape loader/dumper.

1.2. More Information

- The latest software, documentation and news items are available at the PostGIS web site, <http://postgis.refractions.net>.
- More information about the PostgreSQL database server is available at the PostgreSQL main site <http://www.postgresql.org>.
- More information about GiST indexing is available at the GiST development site, <http://www.sai.msu.su/~megeera/postgres/gist>.
- More information about Mapserver internet map server is available at <http://mapserver.gis.umn.edu> (<http://mapserver.gis.umn.edu/>).
- The "Simple Features for Specification for SQL (<http://www.opengis.org/techno/specs/99-049.pdf>)" is available at the OpenGIS Consortium web site: <http://www.opengis.org>.

Chapter 2. Installation

2.1. Requirements

PostGIS has the following requirements for building and usage:

- A complete configured and built PostgreSQL source code tree. PostGIS uses definitions from the PostgreSQL configure/build process to conform to the particular platform you are building on. PostgreSQL is available from <http://www.postgresql.org>.
- GNU C compiler (`gcc`). Some other ANSI C compilers can be used to compile PostGIS, but we find far fewer problems when compiling with `gcc`.
- GNU Make (`gmake` or `make`). For many systems, GNU `make` is the default version of `make`. Check the version by invoking `make -v`. Other versions of `make` may not process the PostGIS `Makefile` properly.
- (Optional) Proj4 reprojection library. The Proj4 library is used to provide coordinate reprojection support within PostGIS. Proj4 is available for download from <http://www.remotesensing.org/proj>.

2.2. PostGIS

The PostGIS module is an extension to the PostgreSQL backend server. As such, PostGIS 0.7 *requires* a full copy of the PostgreSQL source tree in order to compile. The PostgreSQL source code is available at <http://www.postgresql.org>.

PostGIS 0.7 can be built against PostgreSQL 7.1.x or PostgreSQL 7.2.x. Earlier versions of PostgreSQL are *not* supported.

Chapter 2. Installation

1. Before you can compile the postgis server modules, you must compile and install the PostgreSQL package.
2. Retrieve the PostGIS source archive from <http://postgis.refrations.net/postgis-0.7.tar.gz> (<http://postgis.refrations.net/postgis-0.6.tar.gz>). Uncompress and untar the archive in the "contrib" directory of the PostgreSQL source tree.

```
# cd [postgresql source tree]/contrib
# gzip -d -c postgis-0.7.tar.gz | tar xvf -
```

3. Once your PostgreSQL installation is up-to-date, enter the postgis directory, and edit the Makefile.
 - If you are compiling against PostgreSQL 7.2.x, you must set the `USE_PG72` variable to `I`.
 - If want support for coordinate reprojection you must have the Proj4 library installed, and set the `USE_PROJ` variable to `I`.

4. Run the compile and install commands.

```
# make
# make install
```

All files are installed relative to the PostgreSQL install directory, `[prefix]`.

- Libraries are installed `[prefix]/lib/contrib`.
- Important support files such as `postgis.sql` are installed in `[prefix]/share/contrib`.
- Loader and dumber binaries are installed `[prefix]/bin`.

5. PostGIS requires the PL/pgSQL procedural language extension. Before loading the `postgis.sql` file, you must first enable PL/pgSQL. You should use the

`createlang` command. The PostgreSQL 7.1 Programmer's Guide has the details if you want to this manually for some reason.

```
# createlang plpgsql [yourdatabase]
```

6. Now load the PostGIS object and function definitions into your database by loading the `postgis.sql` definitions file.

```
# psql -d [yourdatabase] -f postgis.sql
```

The PostGIS server extensions are now loaded and ready to use.

7. For a complete set of EPSG coordinate system definition identifiers, you can also load the `spatial_ref_sys.sql` definitions file and populate the `SPATIAL_REF_SYS` table.

```
# psql -d [yourdatabase] -f spatial_ref_sys.sql
```

2.2.1. Upgrading

Upgrading PostGIS can be tricky, because the underlying C libraries which support the object types and geometries may have changed between versions. To avoid problems when upgrading, you will have to dump all the tables in your database, destroy the database, create a new one, execute the new `postgis.sql` file, then upload your database dump:

```
# pg_dump -t "*" -f dumpfile.sql yourdatabase
# dropdb yourdatabase
# createdb yourdatabase
# createlang plpgsql yourdatabase
# psql -f postgis.sql -d yourdatabase
# psql -f dumpfile.sql -d yourdatabase
# vacuumdb -z yourdatabase
```

Note: When upgrading from version 0.5 to 0.6+, all your geometries will be created with an SRID of -1. To create valid OpenGIS geometries, you will have to create a valid SRID in the SPATIAL_REF_SYS table, and then update your geometries to reference the SRID with the following SQL (with the appropriate substitutions):

```
UPDATE TABLE <table> SET <geocolumn> = Set-  
SRID(<geocolumn>, <SRID>);
```

2.3. JDBC

The JDBC extensions provide Java objects corresponding to the internal PostGIS types. These objects can be used to write Java clients which query the PostGIS database and draw or do calculations on the GIS data in PostGIS.

1. Enter the `jdbc` sub-directory of the PostGIS distribution.
2. Edit the `Makefile` to provide the correct paths of your java compiler (`JAVAC`) and interpreter (`JAVA`).
3. Run the `make` command. Copy the `postgis.jar` file to wherever you keep your java libraries.

2.4. Loader/Dumper

The data loader and dumper are built and installed automatically as part of the PostGIS build. To build and install them manually:

```
# cd postgis-0.7/loader  
# make
```

```
# make install
```

The loader is called `shp2pgsql` and converts ESRI Shape files into SQL suitable for loading in PostGIS/PostgreSQL. The dumper is called `pgsql2shp` and converts PostGIS tables into ESRI shape files.

Chapter 2. Installation

Chapter 3. Frequently Asked Questions

1. What kind of geometric objects can I store?

You can store point, line, polygon, multipoint, multiline, multipolygon, and geometrycollections. These are specified in the Open GIS Well Known Text Format (with 3d extensions).

2. How do I insert a GIS object into the database?

First, you need to create a table with a column of type geometry to hold your GIS data. Connect to your database with `psql` and try the following SQL:

```
CREATE TABLE gtest ( ID int4, NAME varchar(20) );
SELECT AddGeometryColumn('dbname', 'gtest', 'geom', -
1, 'LINESTRING', 2);
```

If the geometry column addition fails, you probably have not loaded the PostGIS functions and objects into this database. See the installation instructions.

Then, you can insert a geometry into the table using a SQL insert statement. The GIS object itself is formatted using the OpenGIS Consortium well-known text format:

```
INSERT INTO gtest (ID, NAME, GEOM)
VALUES (1, 'First Geometry', GeometryFrom-
Text('LINESTRING(2 3,4 5,6 5,7 8)', -1));
```

For more information about other GIS objects, see the object reference.

To view your GIS data in the table:

```
SELECT id, name, AsText(geom) AS geom FROM gtest;
```

The return value should look something like this:

```
id | name          | geom
---+-----+-----
```

Chapter 3. Frequently Asked Questions

```
1 | First Geometry | LINESTRING(2 3,4 5,6 5,7 8)
(1 row)
```

3. How do I construct a spatial query?

There are a number of spatial operators available to PostgreSQL, and several of them have been implemented by PostGIS in order to provide indexing support.

In order to do a spatial query with index support, you must use the "overlap operator" (`&&`) which uses the following important simplifying assumption: *all features shall be represented by their bounding boxes.*

We recognize that using bounding boxes to proxy for features is a limiting assumption, but it is an important one in providing spatial indexing capabilities. Commercial spatial databases use the same assumption – bounding boxes are important to most indexing schemes.

The most important spatial operator from a user's perspective is the "overlap operator", which tests whether one feature's bounding box overlaps that of another. An example of a spatial query using `&&` is:

```
SELECT id,name FROM GTEST WHERE GEOM && 'BOX3D(3 4,4 5)::box3d
```

Note that the bounding box used for querying must be explicitly declared as a `box3d` using the `::box3d` casting operation.

4. How do I speed up spatial queries on large tables?

Fast queries on large tables is the *raison d'être* of spatial databases (along with transaction support) so having a good index is important.

To build a spatial index on a table with a `geometry` column, use the "CREATE INDEX" function as follows:

```
CREATE INDEX [indexname] ON [tablename]
  USING GIST ( [geometrycolumn] gist_geometry_ops);
```

The "USING GIST" option tells the server to use a GiST (Generalized Search Tree) index. The reference to gist_geometry_ops tells the server to use a particular set of comparison operators for building the index: the gist_geometry_ops part of the PostGIS extension.

Note: For PostgreSQL version 7.1.x, you can specifically request a lossy index by appending WITH (ISLOSSY) to the index creation command. For PostgreSQL 7.2.x and above all GiST indexes are assumed to be lossy. Lossy indexes use a proxy object (in the spatial case, a bounding box) for building the index.

5. How can I get my search to return things that really are inside the search box, not just overlapping bounding boxes?

The '&&' operator only checks bounding box overlaps, but you can use the truly_inside() function to get only those features which *actually* intersect the search box. For example, by combining the use of '&&' for a fast index search and truly_inside() for an accurate final check of the result set, you can get only those features inside the search box (note that this *only* works for search boxes right now, not any arbitrary geometry):

```
SELECT [COLUMN1], [COLUMN2], AsText ([GEOMETRYCOLUMN])
FROM [TABLE] WHERE [GEOM_COLUMN] && [BOX3d]
AND truly_inside([GEOM_COLUMN], [BOX3d]);
```

6. Why aren't PostgreSQL R-Tree indexes supported?

Early versions of PostGIS used the PostgreSQL R-Tree indexes. However, PostgreSQL R-Trees have been completely discarded since version 0.6, and spatial indexing is provided with an R-Tree-over-GiST scheme.

Our tests have shown search speed for native R-Tree and GiST to be comparable. Native PostgreSQL R-Trees have two limitations which make them undesirable for use with GIS features (note that these limitations are due to the current PostgreSQL native R-Tree implementation, not the R-Tree concept in general):

- R-Tree indexes in PostgreSQL cannot handle features which are larger than 8K in size. GiST indexes can, using the lossytrick of substituting the bounding box for the feature itself.
- R-Tree indexes in PostgreSQL are not null safe", so building an index on a geometry column which contains null geometries will fail.

7. Why should I use the AddGeometryColumn() function and all the other OpenGIS stuff?

If you do not want to use the OpenGIS support functions, you do not have to. Simply create tables as in older versions, defining your geometry columns in the CREATE statement. All your geometries will have SRIDs of -1, and the OpenGIS meta-data tables will *not* be filled in properly. For most current applications, this will not matter.

However, in the future it is likely that client software will use the meta-data tables to interrogate the database about available layers and projections before rendering data. An obvious early example is the Mapserver internet mapping software, which could be altered to interrogate the SPATIAL_REF_SYS table for projection information on the layers it is rendering.

For these reasons it is probably wise to learn and use the OpenGIS concepts from early on.

8. What is the best way to find all objects with a radius of another object?

To use the database most efficiently, it is best to do radius queries which combine the radius test with a bounding box test: the bounding box test uses the spatial index, giving fast access to a subset of data which the radius test is then applied to.

For example, to find all objects with 100 meters of POINT(1000 1000) the following query would work:

```
SELECT *
FROM GEOTABLE
WHERE
  GEOM && GeometryFromText('BOX3D(900 900,1100 1100)',-1)
AND
Distance(GeometryFromText('POINT(1000 1000)',-1),GEOM) < 100;
```

9. How do I perform a coordinate reprojection as part of a query?

To perform a reprojection, both the source and destination coordinate systems must be defined in the SPATIAL_REF_SYS table, and the geometries being reprojected must already have an SRID set on them. Once that is done, a reprojection is as simple as referring to the desired destination SRID.

```
SELECT Transform(GEOM,4269) FROM GEOTABLE;
```

Chapter 3. Frequently Asked Questions

Chapter 4. Using PostGIS

4.1. GIS Objects

The GIS objects supported by PostGIS are the Simple Features defined by the OpenGIS Consortium (OGC). Note that PostGIS currently supports the features and the representation APIs, but not the various comparison and convolution operators given in the OGC Simple Features for SQL specification.

Examples of the text representations of the features are as follows:

- POINT(0 0 0)
- LINESTRING(0 0,1 1,1 2)
- POLYGON((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0))
- MULTIPOINT(0 0 0,1 2 1)
- MULTILINESTRING((0 0 0,1 1 0,1 2 1),(2 3 1,3 2 1,5 4 1))
- MULTIPOLYGON(((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0)),((-1 -1 0,-1 -2 0,-2 -2 0,-2 -1 0,-1 -1 0)))
- GEOMETRYCOLLECTION(POINT(2 3 9),LINESTRING((2 3 4,3 4 5)))

Note that in the examples above there are features with both 2-dimensional and 3-dimensional coordinates. PostGIS supports both 2d and 3d coordinates – if you describe a feature with 2D coordinates when you insert it, the database will return that feature to you with 2D coordinates when you extract it. See the sections on the `force_2d()` and `force_3d()` functions for information on converting features to a particular coordinate dimension representation.

4.1.1. Standard versus Canonical Forms

The OpenGIS specification defines two standard ways of expressing spatial objects: the

Well-Known Text (WKT) form (shown in the previous section) and the Well-Known Binary (WKB) form. Both WKT and WKB include information about the type of the object and the coordinates which form the object.

However, the OpenGIS specification also requires that the internal storage format of spatial objects include a spatial referencing system identifier (SRID). The SRID is required when creating spatial objects for insertion into the database. For example, a valid insert statement to create and insert a spatial object would be:

```
INSERT INTO SPATIALTABLE ( THE_GEOM, THE_NAME )
VALUES (
    GeometryFromText('POINT(-126.4 45.32)', 312),
    'A Place'
)
```

Note that the `GeometryFromText` function requires an SRID number.

The "canonical form" of the spatial objects in PostgreSQL is a text representation which includes all the information necessary to construct the object. Unlike the OpenGIS standard forms, it includes the type, coordinate, and SRID information. The canonical form is the default form returned from a `SELECT` query. The example below shows the difference between the OGC standard and PostGIS canonical forms:

```
db=> SELECT AsText(geom) AS OGCGeom FROM thetable;
OGCGeom
-----
LINESTRING(-123.741378393049 48.9124018962261,-
123.741587115639 48.9123981907507)
(1 row)
```

```
db=> SELECT geom AS PostGISGeom FROM thetable;
PostGISGeom
-----
SRID=123;LINESTRING(-123.741378393049 48.9124018962261,-
123.741587115639 48.9123981907507)
(1 row)
```

4.2. Using OpenGIS Standards

The OpenGIS Simple Features Specification for SQL defines standard GIS object types, the functions required to manipulate them, and a set of meta-data tables. In order to ensure that meta-data remain consistent, operations such as creating and removing a spatial column are carried out through special procedures defined by OpenGIS.

There are two OpenGIS meta-data tables: `SPATIAL_REF_SYS` and `GEOMETRY_COLUMNS`. The `SPATIAL_REF_SYS` table holds the numeric IDs and textual descriptions of coordinate systems used in the spatial database.

4.2.1. The `SPATIAL_REF_SYS` Table

The `SPATIAL_REF_SYS` table definition is as follows:

```
CREATE TABLE SPATIAL_REF_SYS (  
    SRID INTEGER NOT NULL PRIMARY KEY,  
    AUTH_NAME VARCHAR(256),  
    AUTH_SRID INTEGER,  
    SRTEXT VARCHAR(2048),  
    PROJ4TEXT VARCHAR(2048)  
)
```

The `SPATIAL_REF_SYS` columns are as follows:

`SRID`

An integer value that uniquely identifies the Spatial Referencing System within the database.

`AUTH_NAME`

The name of the standard or standards body that is being cited for this reference system. For example, "EPSG" would be a valid `AUTH_NAME`.

AUTH_SRID

The ID of the Spatial Reference System as defined by the Authority cited in the AUTH_NAME. In the case of EPSG, this is where the EPSG projection code would go.

SRTEXT

The Well-Known Text representation of the Spatial Reference System. An example of a WKT SRS representation is:

```
PROJCS["NAD83 / UTM Zone 10N",  
  GEOGCS["NAD83",  
    DATUM["North_American_Datum_1983",  
      SPHEROID["GRS 1980",6378137,298.257222101]  
    ],  
    PRIMEM["Greenwich",0],  
    UNIT["degree",0.0174532925199433]  
  ],  
  PROJECTION["Transverse_Mercator"],  
  PARAMETER["latitude_of_origin",0],  
  PARAMETER["central_meridian",-123],  
  PARAMETER["scale_factor",0.9996],  
  PARAMETER["false_easting",500000],  
  PARAMETER["false_northing",0],  
  UNIT["metre",1]  
]
```

For a listing of EPSG projection codes and their corresponding WKT representations, see <http://www.opengis.org/techno/interop/EPG2WKT.TXT>. For a discussion of WKT in general, see the OpenGIS "Coordinate Transformation Services Implementation Specification" at <http://www.opengis.org/techno/specs.htm>.

PROJ4TEXT

PostGIS uses the Proj4 library to provide coordinate transformation capabilities. The PROJ4TEXT column contains the Proj4 coordinate definition string for a particular SRID. For example:

```
+proj=utm +zone=10 +ellps=clrk66 +datum=NAD27 +units=m
```

For more information about, see the Proj4 web site at <http://www.remotesensing.org/proj>. The `spatial_ref_sys.sql` file contains both SRTEXT and PROJ4TEXT definitions for all EPSG projections.

4.2.2. The GEOMETRY_COLUMNS Table

The GEOMETRY_COLUMNS table definition is as follows:

```
CREATE TABLE GEOMETRY_COLUMNS (  
    F_TABLE_CATALOG VARCHAR(256) NOT NULL,  
    F_TABLE_SCHEMA VARCHAR(256) NOT NULL,  
    F_TABLE_NAME VARCHAR(256) NOT NULL,  
    F_GEOMETRY_COLUMN VARCHAR(256) NOT NULL,  
    COORD_DIMENSION INTEGER NOT NULL,  
    SRID INTEGER NOT NULL,  
    TYPE VARCHAR(30) NOT NULL  
)
```

The columns are as follows:

F_TABLE_CATALOG, F_TABLE_SCHEMA, F_TABLE_NAME

The fully qualified name of the feature table containing the geometry column. Note that the terms "catalog" and "schema" are Oracle-ish. There is not PostgreSQL analogue of "catalog" so that column is left blank – for schema the database name is used.

F_GEOMETRY_COLUMN

The name of the geometry column in the feature table.

COORD_DIMENSION

The spatial dimension (2 or 3 dimensional) of the column.

SRID

The ID of the spatial reference system used for the coordinate geometry in this table. It is a foreign key reference to the SPATIAL_REF_SYS.

TYPE

The type of the spatial object. To restrict the spatial column to a single type, use one of: POINT, LINESTRING, POLYGON, MULTPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION. For heterogeneous (mixed-type) collections, you can use GEOMETRY as the type.

Note: This attribute is (probably) not part of the OpenGIS specification, but is required for ensuring type homogeneity.

4.2.3. Creating a Spatial Table

Creating a table with spatial data is done in two stages:

- Create a normal non-spatial table.

For example: `CREATE TABLE ROADS_GEOM (ID int4, NAME varchar(25))`

- Add a spatial column to the table using the OpenGIS `AddGeometryColumn` function. The syntax is: `AddGeometryColumn(<db_name>, <table_name>, <column_name>, <sruid>, <type>, <dimension>)`.

For example: `SELECT AddGeometryColumn('roads_db', 'roads_geom', 'geom', 423, 'LINESTRING', 2)`

Here is an example of SQL used to create a table and add a spatial column (assuming the db is 'parks_db' and that an SRID of 128 exists already):

```
CREATE TABLE PARKS ( PARK_ID int4, PARK_NAME varchar(128), PARK_DATE date, PARK_TYPE varchar(2) );
SELECT AddGeometryColumn('parks_db', 'parks', 'park_geom', 128, 'MULTIPOLYGON', 2 );
```

Here is another example, using the generic geometry type and the undefined SRID value of -1:

```
CREATE TABLE ROADS ( ROAD_ID int4, ROAD_NAME varchar(128) );
SELECT AddGeometryColumn( 'roads_db', 'roads', 'roads_geom', -1, 'GEOMETRY', 3 );
```

4.3. Loading GIS Data

Once you have created a spatial table, you are ready to upload GIS data to the database. Currently, there are two ways to get data into a PostGIS/PostgreSQL database: using formatted SQL statements or using the Shape file loader/dumper.

4.3.1. Using SQL

If you can convert your data to a text representation, then using formatted SQL might be the easiest way to get your data into PostGIS. As with Oracle and other SQL

databases, data can be bulk loaded by piping a large text file full of SQL "INSERT" statements into the SQL terminal monitor.

A data upload file (`roads.sql` for example) might look like this:

```
BEGIN;
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (1,GeometryFrom-
Text('LINESTRING(191232 243118,191108 243242)',-1),'Jeff Rd');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (2,GeometryFrom-
Text('LINESTRING(189141 244158,189265 244817)',-
1),'Geordie Rd');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (3,GeometryFrom-
Text('LINESTRING(192783 228138,192612 229814)',-1),'Paul St');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (4,GeometryFrom-
Text('LINESTRING(189412 252431,189631 259122)',-
1),'Graeme Ave');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (5,GeometryFrom-
Text('LINESTRING(190131 224148,190871 228134)',-1),'Phil Tce');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (6,GeometryFrom-
Text('LINESTRING(198231 263418,198213 268322)',-1),'Dave Cres');
COMMIT;
```

The data file can be piped into PostgreSQL very easily using the `psql` terminal monitor:

```
psql -d [database] -f roads.sql
```

4.3.2. Using the Loader

The `shp2pgsql` data loader converts ESRI Shape files into SQL suitable for insertion into a PostGIS/PostgreSQL database. The loader has several operating modes distinguished by command line flags:

`-d`

Drops the database table before creating a new table with the data in the Shape file.

-a

Appends data from the Shape file into the database table. Note that to use this option to load multiple files, the files must have the same attributes and same data types.

-c

Creates a new table and populates it from the Shape file. *This is the default mode.*

-D

Creates a new table and populates it from the Shape file. This uses the PostgreSQL dumpformat for the output data and is much faster to load than the default "insertSQL format. Use this for very large data sets.

-s <SRID>

Creates and populates the geometry tables with the specified SRID.

An example session using the loader to create an input file and uploading it might look like this:

```
# shp2pgsql shaperoads roadstable roadsdb > roads.sql
# psql -d roadsdb -f roads.sql
```

A conversion and upload can be done all in one step using UNIX pipes:

```
# shp2pgsql shaperoads roadstable roadsdb | psql -d roadsdb
```

4.4. Retrieving GIS Data

Data can be extracted from the database using either SQL or the Shape file loader/dumper. In the section on SQL we will discuss some of the operators available to do comparisons and queries on spatial tables.

4.4.1. Using SQL

The most straightforward means of pulling data out of the database is to use a SQL select query and dump the resulting columns into a parsable text file:

```
db=# SELECT id, AsText(geom) AS geom, name FROM ROADS_GEOM;
id | geom | name
---+-----+-----
 1 | LINESTRING(191232 243118,191108 243242) | Jeff Rd
 2 | LINESTRING(189141 244158,189265 244817) | Geordie Rd
 3 | LINESTRING(192783 228138,192612 229814) | Paul St
 4 | LINESTRING(189412 252431,189631 259122) | Graeme Ave
 5 | LINESTRING(190131 224148,190871 228134) | Phil Tce
 6 | LINESTRING(198231 263418,198213 268322) | Dave Cres
 7 | LINESTRING(218421 284121,224123 241231) | Chris Way
(6 rows)
```

However, there will be times when some kind of restriction is necessary to cut down the number of fields returned. In the case of attribute-based restrictions, just use the same SQL syntax as normal with a non-spatial table. In the case of spatial restrictions, the following operators are available/useful:

&&

This operator tells whether the bounding box of one geometry overlaps the bounding box of another.

~=

This operators tests whether two geometries are geometrically identical. For example, if 'POLYGON((0 0,1 1,1 0,0 0))' is the same as 'POLYGON((0 0,1 1,1 0,0 0))' (it is).

=

This operator is a little more naive, it only tests whether the bounding boxes of to geometries are the same.

Next, you can use these operators in queries. Note that when specifying geometries and boxes on the SQL command line, you must explicitly turn the string representations into geometries by using the `GeometryFromText()` function. So, for example:

```
SELECT
  ID, NAME
FROM ROADS_GEOM
WHERE
  GEOM ~= GeometryFrom-
Text('LINESTRING(191232 243118,191108 243242)',-1);
```

The above query would return the single record from the `ROADS_GEOM` table in which the geometry was equal to that value.

When using the `&&` operator, you can specify either a `BOX3D` as the comparison feature or a `GEOMETRY`. When you specify a `GEOMETRY`, however, its bounding box will be used for the comparison.

```
SELECT
  ID, NAME
FROM ROADS_GEOM
WHERE
  GEOM && GeometryFrom-
Text('POLYGON((191232 243117,191232 243119,191234 243117,191232 243117))',-
1);
```

The above query will use the bounding box of the polygon for comparison purposes.

The most common spatial query will probably be a frame-based query, used by client software, like data browsers and web mappers, to grab a map frame worth of data for display. Using a `BOX3D` object for the frame, such a query looks like this:

```
SELECT
  AsText(GEOM) AS GEOM
FROM ROADS_GEOM
WHERE
  GEOM && GeometryFrom-
Text('BOX3D(191232 243117,191232 243119)')::box3d,-1);
```

Note the use of the SRID, to specify the projection of the BOX3D. The -1 is used to indicate no specified SRID.

4.4.2. Using the Dumper

The `pgsql2shp` table dumper connects directly to the database and converts a table into a shape file. The basic syntax is:

```
pgsql2shp [<options>] <database> <table>
```

The commandline options are:

`-d`

Write a 3-dimensional shape file. The default is to write a 2-dimensional shape file.

`-f <filename>`

Write the output to a particular filename.

`-h <host>`

The database host to connect to.

`-p <port>`

The port to connect to on the database host.

`-P <password>`

The password to use when connecting to the database.

`-u <user>`

The username to use when connecting to the database.

-g <geometry column>

In the case of tables with multiple geometry columns, the geometry column to use when writing the shape file.

4.4.3. Using Minnesota Mapserver

The Minnesota Mapserver is an internet web-mapping server. The latest versions conform to the OpenGIS Web Map Specification.

- The Mapserver homepage is at <http://mapserver.gis.umn.edu>.
- The OpenGIS Web Map Specification is at <http://www.opengis.org/techno/specs/01-047r2.pdf>.

To use PostGIS with Mapserver, you will need to know about how to configure Mapserver, which is beyond the scope of this documentation. This section will cover specific PostGIS issues and configuration details.

To use PostGIS with Mapserver, you will need:

- Version 0.6 or newer of PostGIS.
- Version 3.5 or newer of Mapserver.

Mapserver accesses PostGIS/PostgreSQL data like any other PostgreSQL client – using `libpq`. This means that Mapserver can be installed on any machine with network access to the PostGIS server, as long as the system has the `libpq` PostgreSQL client libraries.

1. Compile and install Mapserver, with whatever options you desire, including the `-with-postgis` configuration option.
2. In your Mapserver map file, add a PostGIS layer. For example:

```
LAYER
  CONNECTIONTYPE postgis
```

```
NAME 'sidehighways'
# Connect to a remote spatial database
CONNECTION "user=dbuser dbname=gisdatabase host=bigserver"
# Get the lines from the 'geom' column of the 'roads' table
DATA geom from roads"
STATUS ON
TYPE LINE
# Of the lines in the extents, only render the wide highways
FILTER type = 'highway' and numlanes >= 4"
CLASS
  # Make the superhighways brighter and 2 pixels wide
  EXPRESSION ([numlanes] >= 6)
  COLOR 255 22 22
  SYMBOL solid"
  SIZE 2
END
CLASS
  # All the rest are darker and only 1 pixel wide
  EXPRESSION ([numlanes] < 6)
  COLOR 205 92 82
END
END
```

In the example above, the PostGIS-specific directives are as follows:

CONNECTIONTYPE

For PostGIS layers, this is always `postgis`".

CONNECTION

The database connection is governed by the a 'connection string' which is a standard set of keys and values like this (with the default values in <>):

```
user=<username> password=<password> dbname=<username>
hostname=<server> port=<5432>
```

An empty connection string is still valid, and any of the key/value pairs can be omitted. At a minimum you will generally supply the database name and username to connect with.

DATA

The form of this parameter is "<column> from <tablename>" where the column is the spatial column to be rendered to the map.

FILTER

The filter must be a valid SQL string corresponding to the logic normally following the `WHERE` keyword in a SQL query. So, for example, to render only roads with 6 or more lanes, use a filter of `num_lanes >= 6`.

3. In your spatial database, ensure you have spatial (GiST) indexes built for any the layers you will be drawing.

```
CREATE INDEX [indexname]
  ON [tablename]
  USING GIST ( [geometrycolumn] GIST_GEOMETRY_OPS );
```

4. If you will be querying your layers using Mapserver you will also need an `oid` index".

Mapserver requires unique identifiers for each spatial record when doing queries, and the PostGIS module of Mapserver uses the PostgreSQL `oid` value to provide these unique identifiers. A side-effect of this is that in order to do fast random access of records during queries, an index on the `oid` is needed.

To build an `oid` index", use the following SQL:

```
CREATE INDEX [indexname] ON [tablename] ( oid );
```

4.5. Building Indexes

Indexes are what make using a spatial database for large databases possible. Without indexing, any search for a feature would require a sequential scan of every record in the database. Indexing speeds up searching by organizing the data into a search tree which can be quickly traversed to find a particular record. PostgreSQL supports three kinds of indexes by default: B-Tree indexes, R-Tree indexes, and GiST indexes.

- B-Trees are used for data which can be sorted along one axis; for example, numbers, letters, dates. GIS data cannot be rationally sorted along one axis (which is greater, (0,0) or (0,1) or (1,0)?) so B-Tree indexing is of no use for us.
- R-Trees break up data into rectangles, and sub-rectangles, and sub-sub rectangles, etc. R-Trees are used by some spatial databases to index GIS data, but the PostgreSQL R-Tree implementation is not as robust as the GiST implementation.
- GiST (Generalized Search Trees) indexes break up data into things to one side", things which overlap", things which are inside and can be used on a wide range of data-types, including GIS data. PostGIS uses an R-Tree index implemented on top of GiST to index GIS data.

4.5.1. GiST Indexes

GiST stands for Generalized Search Tree and is a generalized form of indexing. In addition to GIS indexing, GiST is used to speed up searches on all kinds of irregular data structures (integer arrays, spectral data, etc) which are not amenable to normal B-Tree indexing.

Once a GIS data table exceeds a few thousand rows, you will want to build an index to speed up spatial searches of the data (unless all your searches are based on attributes, in which case you'll want to build a normal index on the attribute fields).

The syntax for building a GiST index on a geometry column is as follows:

```
CREATE INDEX [indexname] ON [tablename]
  USING GIST ( [geometryfield] GIST_GEOMETRY_OPS );
```

Building a spatial index is a computationally intensive exercise: on tables of around 1 million rows, on a 300MHz Solaris machine, we have found building a GiST index takes about 1 hour. After building an index, it is important to force PostgreSQL to collect table statistics, which are used to optimize query plans:

```
VACUUM ANALYZE ;
```

GiST indexes have two advantages over R-Tree indexes in PostgreSQL. Firstly, GiST indexes are null safe", meaning they can index columns which include null values. Secondly, GiST indexes support the concept of lossiness which is important when dealing with GIS objects larger than the PostgreSQL 8K page size. Lossiness allows PostgreSQL to store only the "important part of an object in an index – in the case of GIS objects, just the bounding box. GIS objects larger than 8K will cause R-Tree indexes to fail in the build phase.

4.5.2. Using Indexes

Ordinarily, indexes invisibly speed up data access: once the index is built, the query planner transparently decides when to use index information to speed up a query plan. Unfortunately, the PostgreSQL query planner does not optimize the use of GiST indexes well, so sometimes searches which should use a spatial index instead default to a sequence scan of the whole table.

If you find your spatial indexes are not being used (or your attribute indexes, for that matter) there are a couple things you can do:

- Firstly, make sure you run the "VACUUM ANALYZE [tablename]" command on the tables you are having problems with. "VACUUM ANALYZE" gathers statistics about the number and distributions of values in a table, to provide the query planner with better information to make decisions around index usage. You should regularly vacuum your databases anyways – many PostgreSQL DBAs have "VACUUM" run as an off-peak cron job on a regular basis.
- If vacuuming does not work, you can force the planner to use the index information

by using the `SET =OFF` command. You should only use this command sparingly, and only on spatially indexed queries: generally speaking, the planner knows better than you do about when to use normal B-Tree indexes. Once you have run your query, you should consider setting `"ENABLE_SEQSCAN` back on, so that other queries will utilize the planner as normal.

Note: As of version 0.6, it should not be necessary to force the planner to use the index with `"ENABLE_SEQSCAN`.

4.6. Complex Queries

The *raison d'être* of spatial database functionality is performing queries inside the database which would ordinarily require desktop GIS functionality. Using PostGIS effectively requires knowing what spatial functions are available, and ensuring that appropriate indexes are in place to provide good performance.

More to come...

4.7. Java Clients (JDBC)

Java clients can access PostGIS geometry objects in the PostgreSQL database either directly as text representations or using the JDBC extension objects bundled with PostGIS. In order to use the extension objects, the `postgis.jar` file must be in your `CLASSPATH` along with the `postgresql.jar` JDBC driver package.

```
import java.sql.*;  
import java.util.*;  
import java.lang.*;
```

```

import org.postgis.*;

public class JavaGIS {
    public static void main(String[] args)
    {
        java.sql.Connection conn;
        try
        {
            /*
             * Load the JDBC driver and establish a connection.
             */
            Class.forName("org.postgresql.Driver");
            String url = "jdbc:postgresql://localhost:5432/database";
            conn = DriverManager.getConnection(url, "postgres", "");

            /*
             * Add the geometry types to the connection. Note that you
             * must cast the connection to the postgres-specific connec-
             * tion * implementation before calling the addDataType() method.
             */
            ((org.postgresql.Connection)conn).addDataType("geometry", "org.postgis.PGgeometry");
            ((org.postgresql.Connection)conn).addDataType("box3d", "org.postgis.PGbox3d");

            /*
             * Create a statement and execute a select query.
             */
            Statement s = conn.createStatement();
            ResultSet r = s.executeQuery("select AsText(geom) as geom,id from geomtable");
            while( r.next() )
            {
                /*
                 * Retrieve the geometry as an ob-
                 * ject then cast it to the geometry type.
                 * Print things out.
                */
            }
        }
    }
}

```

```
        */
        PGgeometry geom = (PGgeometry)r.getObject(1);
        int id = r.getInt(2);
        System.out.println("Row " + id + ":");
        System.out.println(geom.toString());
    }
    s.close();
    conn.close();
}
catch( Exception e )
{
    e.printStackTrace();
}
}
```

The PGgeometry object is a wrapper object which contains a specific topological geometry object (subclasses of the abstract class Geometry") depending on the type: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon.

```
PGgeometry geom = (PGgeometry)r.getObject(1);
if( geom.getType() = Geometry.POLYGON )
{
    Polygon pl = (Polygon)geom.getGeometry();
    for( int r = 0; r < pl.numRings(); r++ )
    {
        LinearRing rng = pl.getRing(r);
        System.out.println("Ring: " + r);
        for( int p = 0; p < rng.numPoints(); p++ )
        {
            Point pt = rng.getPoint(p);
            System.out.println("Point: " + p);
            System.out.println(pt.toString());
        }
    }
}
```

The JavaDoc for the extension objects provides a reference for the various data accessor functions in the geometric objects.

4.8. C Clients (libpq)

...

4.8.1. Text Cursors

...

4.8.2. Binary Cursors

...

Chapter 4. Using PostGIS

Chapter 5. PostGIS Reference

The functions given below are the ones which a user of PostGIS is likely to need. There are other functions which are required support functions to the PostGIS objects which are not of use to a general user.

5.1. OpenGIS Functions

AddGeometryColumn(*varchar*, *varchar*, *varchar*, *integer*, *varchar*, *integer*)

Syntax: AddGeometryColumn(<db_name>, <table_name>, <column_name>, <srid>, <type>, <dimension>). Adds a geometry column to an existing table of attributes. The *dbname* is the name of the database instance. The *srid* must be an integer value reference to an entry in the SPATIAL_REF_SYS table. The *type* must be an uppercase string corresponding to the geometry type, eg, 'POLYGON' or 'MULTILINESTRING'.

DropGeometryColumn(*varchar*, *varchar*, *varchar*)

Syntax: DropGeometryColumn(<db_name>,<table_name>,<column_name>).
Remove a geometry column from a spatial table.

AsBinary(*geometry*)

Returns the geometry in the OGC well-known-binaryformat, using the endian encoding of the server on which the database is running. This is useful in binary cursors to pull data out of the database without converting it to a string representation.

Dimension(*geometry*)

Returns '2' if the geometry is two dimensional and '3' if the geometry is three dimensional.

Envelope(geometry)

Returns a POLYGON representing the bounding box of the geometry.

GeometryType(geometry)

Returns the type of the geometry as a string. Eg: 'LINESTRING', 'POLYGON', 'MULTIPOINT', etc.

X(geometry)

Find and return the X coordinate of the first point in the geometry. Return NULL if there is no point in the geometry.

Y(geometry)

Find and return the Y coordinate of the first point in the geometry. Return NULL if there is no point in the geometry.

Z(geometry)

Find and return the Z coordinate of the first point in the geometry. Return NULL if there is no point in the geometry.

NumPoints(geometry)

Find and return the number of points in the first linestring in the geometry. Return NULL if there is no linestring in the geometry.

PointN(geometry,integer)

Return the N'th point in the first linestring in the geometry. Return NULL if there is no linestring in the geometry.

ExteriorRing(geometry)

Return the exterior ring of the first polygon in the geometry. Return NULL if there is no polygon in the geometry.

NumInteriorRings(geometry)

Return the number of interior rings of the first polygon in the geometry. Return NULL if there is no polygon in the geometry.

InteriorRingN(geometry,integer)

Return the N'th interior ring of the first polygon in the geometry. Return NULL if there is no polygon in the geometry.

IsClosed(geometry)

Returns true if the geometry start and end points are coincident.

NumGeometries(geometry)

If geometry is a GEOMETRYCOLLECTION return the number of geometries, otherwise return NULL.

GeometryN(geometry,int)

Return the N'th geometry if the geometry is a GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING or MULTIPOLYGON. Otherwise, return NULL.

Distance(geometry,geometry)

Return the cartesian distance between two geometries in projected units.

AsText(geometry)

Return the Well-Known Text representation of the geometry. For example:
POLYGON(0 0,0 1,1 1,1 0,0 0)

SRID(geometry)

Returns the integer SRID number of the spatial reference system of the geometry.

GeometryFromText(varchar, integer)

Syntax: GeometryFromText(<geometry>,<SRID>) Convert a Well-Known Text representation of a geometry into a geometry object.

GeomFromText(varchar, integer)

As above. A synonym for GeometryFromText.

SetSRID(geometry)

Set the SRID on a geometry to a particular integer value. Useful in constructing bounding boxes for queries.

EndPoint(geometry)

Returns the last point of the geometry as a point.

StartPoint(geometry)

Returns the first point of the geometry as a point.

Centroid(geometry)

Returns the centroid of the geometry as a point.

5.2. Other Functions

A <& B

The "<&" operator returns true if A's bounding box overlaps or is to the right of B's bounding box.

A &> B

The "&>" operator returns true if A's bounding box overlaps or is to the left of B's bounding box.

`A << B`

The "`<<`" operator returns true if A's bounding box is strictly to the right of B's bounding box.

`A >> B`

The "`>>`" operator returns true if A's bounding box is strictly to the left of B's bounding box.

`A ~= B`

The "`~='`" operator is the same as `asö` operator. It tests actual geometric equality of two features. So if A and B are the same feature, vertex-by-vertex, the operator returns true.

`A ~ B`

The "`~`" operator returns true if A's bounding box is completely contained by B's bounding box.

`A && B`

The "`&&`" operator is the `ö`verlaps operator. If A's bounding box overlaps B's bounding box the operator returns true.

`area2d(geometry)`

Returns the area of the geometry if it is a polygon or multi-polygon.

`asbinary(geometry,'NDR')`

Returns the geometry in the OGC well-known-binaryformat, using little-endian encoding. This is useful in binary cursors to pull data out of the database without converting it to a string representation.

`asbinary(geometry,'XDR')`

Returns the geometry in the OGC well-known-binaryformat, using big-endian encoding. This is useful in binary cursors to pull data out of the database without

converting it to a string representation.

`box3d(geometry)`

Returns a BOX3D representing the maximum extents of the geometry.

`collect(geometry)`

This function returns a GEOMETRYCOLLECTION object from a set of geometries. The `collect()` function is an aggregate function in the terminology of PostgreSQL. That means that it operates on lists of data, in the same way the `sum()` and `mean()` functions do. For example, `SELECT COLLECT(GEOM) FROM GEOMTABLE GROUP BY ATTRCOLUMN` will return a separate GEOMETRYCOLLECTION for each distinct value of ATTRCOLUMN.

`distance_spheroid(point, point, spheroid)`

Returns linear distance between two lat/lon points given a particular spheroid. See the explanation of spheroids given for `length_spheroid()`. Currently only implemented for points.

`extent(geometry)`

The `extent()` function is an aggregate function in the terminology of PostgreSQL. That means that it operates on lists of data, in the same way the `sum()` and `mean()` functions do. For example, `SELECT EXTENT(GEOM) FROM GEOMTABLE` will return a BOX3D giving the maximum extent of all features in the table. Similarly, `SELECT EXTENT(GEOM) FROM GEOMTABLE GROUP BY CATEGORY` will return one extent result for each category.

`find_srid(varchar,varchar,varchar)`

The syntax is `find_srid(<db/schema>, <table>, <column>)` and the function returns the integer SRID of the specified column by searching through the GEOMETRY_COLUMNS table. If the geometry column has not been properly added with the `AddGeometryColumns()` function, this function will not work either.

`force_collection(geometry)`

Converts the geometry into a GEOMETRYCOLLECTION. This is useful for simplifying the WKB representation.

`force_2d(geometry)`

Forces the geometries into a "2-dimensional" mode so that all output representations will only have the X and Y coordinates. This is useful for force OGC-compliant output (since OGC only specifies 2-D geometries).

`force_3d(geometry)`

Forces the geometries into a "3-dimensional" mode so that all output representations will have the X, Y and Z coordinates.

`length2d(geometry)`

Returns the 2-dimensional length of the geometry if it is a linestring or multi-linestring.

`length3d(geometry)`

Returns the 3-dimensional length of the geometry if it is a linestring or multi-linestring.

`length_spheroid(geometry,spheroid)`

Calculates the length of a geometry on an ellipsoid. This is useful if the coordinates of the geometry are in latitude/longitude and a length is desired without reprojection. The ellipsoid is a separate database type and can be constructed as follows:

```
SPHEROID[<NAME>,<SEMI-MAJOR AXIS>,<INVERSE FLATTENING>]
```

Eg:

```
SPHEROID[GRS_1980",6378137,298.257222101]
```

An example calculation might look like this:

```
SELECT
```

```
length_spheroid(  
    geometry_column,  
    'SPHEROID[GRS_1980",6378137,298.257222101]'  
)  
from geometry_table;
```

`length3d_spheroid(geometry,spheroid)`

Calculates the length of a geometry on an ellipsoid, taking the elevation into account. This is just like `length_spheroid` except vertical coordinates (expressed in the same units as the spheroid axes) are used to calculate the extra distance vertical displacement adds.

`max_distance(linestring,linestring)`

Returns the largest distance between two line strings.

`mem_size(geometry)`

Returns the amount of space (in bytes) the geometry takes.

`npoints(geometry)`

Returns the number of points in the geometry.

`nrings(geometry)`

If the geometry is a polygon or multi-polygon returns the number of rings.

`numb_sub_objects(geometry)`

Returns the number of objects stored in the geometry. This is useful for MULTI-geometries and GEOMETRYCOLLECTIONs.

`perimeter2d(geometry)`

Returns the 2-dimensional perimeter of the geometry, if it is a polygon or multi-polygon.

`perimeter3d(geometry)`

Returns the 3-dimensional perimeter of the geometry, if it is a polygon or multi-polygon.

`point_inside_circle(geometry,float,float,float)`

The syntax for this functions is

`point_inside_circle(<geometry>,<circle_center_x>,<circle_center_y>,<radius>).`

Returns the true if the geometry is a point and is inside the circle. Returns false otherwise.

`postgis_version()`

Returns the version number of the PostGIS functions installed in this database.

`summary(geometry)`

Returns a text summary of the contents of the geometry.

`transform(geometry,integer)`

Returns a new geometry with its coordinates transformed to the SRID referenced by the integer parameter. The destination SRID must exist in the SPATIAL_REF_SYS table.

`translate(geometry,float8,float8,float8)`

Translates the geometry to a new location using the numeric parameters as offsets. Ie: `translate(geom,X,Y,Z).`

`truly_inside(geometryA,geometryB)`

Returns true if any part of B is within the bounding box of A.

`xmin(box3d) ymin(box3d) zmin(box3d)`

Returns the requested minima of a bounding box.

`xmax(box3d) ymax(box3d) zmax(box3d)`

Returns the requested maxima of a bounding box.

