PostGIS Manual

PostGIS Manual

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 analysis and processing of GIS objects.

Table of Contents

Chapter 1. Introduction

PostGIS is developed by Refractions Research Inc, as a spatial database technology research project. Refractions 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

* 0.60

* 0.60 Sandro Santilli <strk@refractions.net>

  Coordinates all bug fixing and maintainance effort, integration of new GEOS functionality, and new function enhancements.

* 0.60 Chris Hodgson <chodgson@refractions.net>

  Maintains new functions and the 7.2 index bindings.

* 0.60 Paul Ramsey <pramsey@refractions.net>

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

* 0.60 Jeff Lounsbury <jeffloun@refractions.net>

  Original development of the Shape file loader/dumper.

* 0.60 Dave Blasby <dblasby@gmail.com>

  The original developer of PostGIS. Dave wrote the server side objects, index bindings, and many of the server side analytical functions.

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 GEOS geometry operations library is available at http://geos.refractions.net [http://geos.refractions.net].
- More information about the Proj4 reprojection library is available at http://www.remotesensing.org/proj.
- 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 PostgreSQL GiST development site, http://www.sai.msu.su/~megera/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.
- (Recommended) 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.
- (Recommended) GEOS geometry library. The GEOS library is used to provide geometry tests (Touches(), Contains(), Intersects()) and operations (Buffer(), GeomUnion(), Difference()) within PostGIS. GEOS is available for download from http://geos.refractions.net.

2.2. PostGIS

The PostGIS module is a extension to the PostgreSQL backend server. As such, PostGIS 1.0.0RC4 *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 1.0.0RC4 can be built against PostgreSQL versions 7.2.0 to 7.4.x. Earlier versions of PostgreSQL are *not* supported.

1. Before you can compile the PostGIS server modules, you must compile and install the PostgreSQL package.

   ### Note

   If you plan to use GEOS functionality you might need to explicitly link PostgreSQL against the standard C++ library:

   ```
   LDFLAGS=-lstdc++ ./configure [YOUR OPTIONS HERE]
   ```

   This is a workaround for bogus C++ exceptions interaction with older development tools. If you experience weird problems (backend unexpectedly closed or similar things) try this trick. This will require recompiling your PostgreSQL from scratch, of course.

2. Retrieve the PostGIS source archive from http://postgis.refractions.net/postgis-1.0.0RC4.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-1.0.0RC4.tar.gz | tar xvf -
```

3. Once your PostgreSQL installation is up-to-date, enter the "postgis" directory, and edit the `Makefile`.

   - If want support for coordinate reprojection you must have the Proj4 library installed, set the `USE_PROJ` variable to *1*, and adjust the `PROJ_DIR` variable to point to your Proj4 installation directory.
   - If want to use GEOS functionality you must have the GEOS library installed, set the `USE_GEOS` variable to *1*, and adjust the `GEOS_DIR` variable to point to your GEOS installation directory.

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 `lwpostgis.sql` are installed in `[prefix]/share/contrib`.
   - Loader and dumber binaries are installed in `[prefix]/bin`.

5. PostGIS requires the PL/pgSQL procedural language extension. Before loading the `lwpostgis.sql` file, you must first enable PL/pgSQL. You should use the `createlang` command. The PostgreSQL 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 `lwpostgis.sql` definitions file.

```
# psql -d [yourdatabase] -f lwpostgis.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.

For this purpose PostGIS provides an utility script to restore a dump produced with the pg_dump -Fc command. It is experimental so redirecting its output to a file will help in case of problems. The procedure is as follow:

```
# Create a "custom-format" dump of the database you want
# to upgrade (let's call it "olddb")
$ pg_dump -Fc olddb olddb.dump

# Restore the dump contextually upgrading postgis into
# a new database. The new database doesn't have to exist.
# Let's call it "newdb"
$ sh utils/postgis_restore.pl lwpostgis.sql newdb olddb.dump > restore.log

# Check that all restored dump objects really had to be restored from dump
# and do not conflict with the ones defined in lwpostgis.sql
$ grep ^KEEPING restore.log | less

# If upgrading from PostgreSQL < 7.5 to >= 7.5 you might want to
# drop the attrelid, varattnum and stats columns in the geometry_columns
# table, which are no-more needed. Keeping them won't hurt.
# !!! DROPPING THEM WHEN REALLY NEEDED WILL DO HURT !!!!
$ psql newdb -c "ALTER TABLE geometry_columns DROP attrelid"
$ psql newdb -c "ALTER TABLE geometry_columns DROP varattnum"
$ psql newdb -c "ALTER TABLE geometry_columns DROP stats"

# spatial_ref_sys table is restore from the dump, to ensure your custom
# additions are kept, but the distributed one might contain modification
# so you should backup your entries, drop the table and source the new one.
# If you did make additions we assume you know how to backup them before
# upgrading the table. Replace of it with the new one is done like this:
$ psql newdb
newdb=> drop table spatial_ref_sys;
DROP
newdb=> \i spatial_ref_sys.sql
```

Following is the "old" procedure description. IT SHOULD BE AVOIDED if possible, as it will leave in the database many spurious functions. It is kept in this document as a "backup" in case postgis_restore.pl won't work for you:

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

2.2.2. Common Problems

There are several things to check when your installation or upgrade doesn't go as you expected.

1. It is easiest if you untar the PostGIS distribution into the contrib directory under the PostgreSQL source tree. However, if this is not possible for some reason, you can set the `PGSQL_SRC` environment variable to the path to the PostgreSQL source directory. This will allow you to compile PostGIS, but the **make install** may not work, so be prepared to copy the PostGIS library and executable files to the appropriate locations yourself.

2. Check that you you have installed PostgreSQL 7.2 or newer, and that you are compiling against the same version of the PostgreSQL source as the version of PostgreSQL that is running. Mix-ups can occur when your (Linux) distrubution has already installed PostgreSQL, or you have otherwise installed PostgreSQL before and forgotten about it. PostGIS will only work with PostgreSQL 7.2 or newer, and strange, unexpected error messages will result if you use an older version. To check the version of PostgreSQL which is running, connect to the database using psql and run this query:

   ```
   SELECT version();
   ```

   If you are running an RPM based distribution, you can check for the existence of pre-installed packages using the **rpm** command as follows: **rpm -qa | grep postgresql**

Also check that you have made any necessary changes to the top of the Makefile. This includes:

1. If you want to be able to do coordinate reprojections, you must install the Proj4 library on your system, set the `USE_PROJ` variable to 1 and the `PROJ_DIR` to your installation prefix in the Makefile.

2. If you want to be able to use GEOS functions you must install the GEOS library on your system, and set the `USE_GEOS` to 1 and the `GEOS_DIR` to your installation prefix in the Makefile.

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-1.0.0RC4/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 (or queries) into ESRI Shape files.

Chapter 3. Frequently Asked Questions

3.1. 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 XYZ,XYM,XYZM extentions).

3.2. 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(”, ’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’, GeomFromText(’LINESTRING(
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
----+-------------+---------------------------
  1 | First Geometry | LINESTRING(2 3,4 5,6 5,7 8)
(1 row)
```

3.3. How do I construct a spatial query?

The same way you construct any other database query, as an SQL combination of return values, functions, and boolean tests.

For spatial queries, there are two issues that are important to keep in mind while constructing your query: is there a spatial index you can make use of; and, are you doing expensive calculations on a large number of geometries.

In general, you will want to use the "intersects operator" (&&) which tests whether the bounding boxes of features intersect. The reason the && operator is useful is because if a spatial index is available to speed up the test, the && operator will make use of this. This can make queries much much faster.

You will also make use of spatial functions, such as Distance(), Intersects(), Contains() and Within(), among others, to narrow down the results of your search. Most spatial queries include both an indexed test and a spatial function test. The index test serves to limit the number of return tuples to only tuples that *might* meet the condition of interest. The spatial functions are then use to test the condition exactly.

```
SELECT id, the_geom FROM thetable
WHERE
  the_geom && 'POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))'
AND
  Contains(the_geom,'POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))';
```

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

Fast queries on large tables is the *raison d'etre* 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] );
```

The "USING GIST" option tells the server to use a GiST (Generalized Search Tree) index.

Note

GiST indexes are assumed to be lossy. Lossy indexes uses a proxy object (in the spatial case, a bounding box) for building the index.

You should also ensure that the PostgreSQL query planner has enough information about your index to make rational decisions about when to use it. To do this, you have to "gather statistics" on your geometry tables.

For PostgreSQL 8.0.x and greater, just run the **VACUUM ANALYZE** command.

For PostgreSQL 7.4.x and below, run the **SELECT UPDATE_GEOMETRY_STATS()** command.

3.5. 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 "lossy" trick 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.

3.6. 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. However, this will cause most applications based on PostGIS to fail, and it is generally suggested that you do use `AddGeometryColumn()` to create geometry tables.

Mapserver is one application which makes use of the `geometry_columns` meta-data. Specifically, Mapserver can use the SRID of the geometry column to do on-the-fly reprojection of features into the correct map projection.

3.7. What is the best way to find all objects within 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.

The `Expand()` function is a handy way of enlarging a bounding box to allow an index search of a region of interest. The combination of a fast access index clause and a slower accurate distance test provides the best combination of speed and precision for this query.

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

```
SELECT *
FROM GEOTABLE
WHERE
  GEOCOLUMN && Expand(GeomFromText('POINT(1000 1000)',-1),100)
AND
  Distance(GeomFromText('POINT(1000 1000)',-1),GEOCOLUMN) < 100;
```

3.8. 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 4. Using PostGIS

4.1. GIS Objects

The GIS objects supported by PostGIS are a superset of the "Simple Features" defined by the OpenGIS Consortium (OGC). As of version 0.9, PostGIS supports all the objects and functions specified in the OGC "Simple Features for SQL" specification.

PostGIS extends the standard with support for 3DZ,3DM and 4D coordinates.

4.1.1. OpenGIS WKB and WKT

The OpenGIS specification defines two standard ways of expressing spatial objects: the Well-Known Text (WKT) form 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.

Examples of the text representations (WKT) of the spatial objects of the features are as follows:

- 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 3),LINESTRING((2 3,3 4)))

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.

Input/Output of these formats are available using the following interfaces:

```
bytea WKB = asBinary(geometry);
text WKT = asText(geometry);
geometry = GeomFromWKB(bytea WKB, SRID);
geometry = GeometryFromText(text WKT, SRID);
```

For example, a valid insert statement to create and insert an OGC spatial object would be:

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

4.1.2. PostGIS EWKB, EWKT and Canonical Forms

OGC formats only support 2d geometries, and the associated SRID is *never* embedded in the input/output representations.

Postgis extended formats are currently superset of OGC one (every valid WKB/WKT is a valid EWKB/EWKT) but this might vary in the future, specifically if OGC comes out with a new format conflicting with our extensions. Thus you SHOULD NOT rely on this feature!

Postgis EWKB/EWKT add 3dm,3dz,4d coordinates support and embedded SRID information.

Examples of the text representations (EWKT) of the extended spatial objects of the features are as follows:

- POINT(0 0 0) -- XYZ
- SRID=32632;POINT(0 0) -- XY with SRID
- POINTM(0 0 0) -- XYM
- POINT(0 0 0 0) -- XYZM
- SRID=4326;MULTIPOINTM(0 0 0,1 2 1) -- XYM with SRID

- MULTILINESTRING((0 0 0,1 1 0,1 2 1),(2 3 1,3 2 1,5 4 1))
- 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))
- 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)))
- GEOMETRYCOLLECTIONM(POINTM(2 3 9),LINESTRINGM((2 3 4,3 4 5)))

Input/Output of these formats are available using the following interfaces:

```
bytea EWKB = asEWKB(geometry);
text EWKT = asEWKT(geometry);
geometry = GeomFromEWKB(bytea EWKB);
geometry = GeomFromEWKT(text EWKT);
```

For example, a valid insert statement to create and insert a PostGIS spatial object would be:

```
INSERT INTO SPATIALTABLE (
  THE_GEOM,
  THE_NAME
)
VALUES (
  GeomFromEWKT('SRID=312;POINTM(-126.4 45.32 15)'),
  'A Place'
)
```

The "canonical forms" of a PostgreSQL type are the representations you get with a simple query (without any function call) and the one which is guaranteed to be accepted with a simple insert, update or copy. For the postgis 'geometry' type these are:

```
- Output -
binary: EWKB
 ascii: HEXEWKB (EWKB in hex form)

- Input -
binary: EWKB
 ascii: HEXEWKB|EWKT
```

For example this statement reads EWKT and returns HEXEWKB in the process of canonical ascii input/output:

```
=# SELECT 'SRID=4;POINT(0 0)'::geometry;
    geometry
----------------------------------------------------
 0101000020040000000000000000000000000000000000000000
(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:
* 0.60

* 0.60 SRID

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

* 0.60 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.

* 0.60 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.

* 0.60 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/EPSG2WKT.TXT. For a discussion of WKT in general, see the OpenGIS "Coordinate Transformation Services Implementation Specification" at http://www.opengis.org/techno/specs.htm. For information on the European Petroleum Survey Group (EPSG) and their database of spatial reference systems, see http://epsg.org.

* 0.60 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:

* 0.60

* 0.60 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 PostgreSQL schema name is used (`public` is the default).

* 0.60 F_GEOMETRY_COLUMN

The name of the geometry column in the feature table.

* 0.60 COORD_DIMENSION

The spatial dimension (2, 3 or 4 dimensional) of the column.

* 0.60 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`.

* 0.60 TYPE

The type of the spatial object. To restrict the spatial column to a single type, use one of: POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION or corresponding XYM versions POINTM, LINESTRINGM, POLYGONM, MULTIPOINTM, MULTILINESTRINGM, MULTIPOLYGONM, GEOMETRYCOLLECTIONM. 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(<schema_name>, <table_name>,
          <column_name>, <srid>, <type>,
          <dimension>)
  ```

  Or, using current schema:

  ```
  AddGeometryColumn(<table_name>,
          <column_name>, <srid>, <type>,
          <dimension>)
  ```

  Example1: **SELECT AddGeometryColumn('public', 'roads_geom', 'geom', 423, 'LINESTRING', 2)**

  Example2: **SELECT AddGeometryColumn('roads_geom', 'geom', 423, 'LINESTRING', 2)**

Here is an example of SQL used to create a table and add a spatial column (assuming 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', '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', '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,GeomFromText('LINESTRING(191232 243118,191108 243242)',-1),'Jeff Rd');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES↵
(2,GeomFromText('LINESTRING(189141 244158,189265 244817)',-1),'Geordie Rd');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES↵
(3,GeomFromText('LINESTRING(192783 228138,192612 229814)',-1),'Paul St');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES↵
(4,GeomFromText('LINESTRING(189412 252431,189631 259122)',-1),'Graeme Ave');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES↵
(5,GeomFromText('LINESTRING(190131 224148,190871 228134)',-1),'Phil Tce');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES↵
(6,GeomFromText('LINESTRING(198231 263418,198213 268322)',-1),'Dave Cres');
COMMIT;
```

The data file can be piped into PostgreSQL very easily using the "psql" SQL 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:
* 0.60

* 0.60 -d

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

* 0.60 -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.

* 0.60 -c

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

* 0.60 -D

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

* 0.60 -s <SRID>

  Creates and populates the geometry tables with the specified SRID.

* 0.60 -k

  Keep idendifiers case (column, schema and attributes). Note that attributes in Shapefile are all UPPERCASE.

* 0.60 -i

  Coerce all integers to standard 32-bit integers, do not create 64-bit bigints, even if the DBF header signature appears to warrant it.

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

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

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

```
# shp2pgsql shaperoads myschema.roadstable | 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:
* 0.60

* 0.60 &&
  This operator tells whether the bounding box of one geometry intersects the bounding box of another.

* 0.60 ~=
  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).

* 0.60 =
  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 "GeomFromText()" function. So, for example:

```
SELECT
  ID, NAME
FROM ROADS_GEOM
WHERE
  GEOM ~= GeomFromText('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 && GeomFromText('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 && GeomFromText('BOX3D(191232 243117,191232 243119)'::box3d,-1);
```

Note the use of the SRID, to specify the projection of the BOX3D. The value -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 (possibly defined by a query) into a shape file. The basic syntax is:

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

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

The commandline options are:
* 0.60

* 0.60 -f <filename>

Write the output to a particular filename.

* 0.60 -h <host>

  The database host to connect to.

* 0.60 -p <port>

  The port to connect to on the database host.

* 0.60 -P <password>

  The password to use when connecting to the database.

* 0.60 -u <user>

  The username to use when connecting to the database.

* 0.60 -g <geometry column>

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

* 0.60 -b

  Use a binary cursor. This will make the operation faster, but will not work if any NON-geometry attribute in the table lacks a cast to text.

* 0.60 -r

  Raw mode. Do not drop the `gid` field, or escape column names.

* 0.60 -d

  For backward compatibility: write a 3-dimensional shape file when dumping from old (pre-1.0.0) postgis databases (the default is to write a 2-dimensional shape file in that case). Starting from postgis-1.0.0+, dimensions are fully encoded.

4.5. Building Indexes

Indexes are what make using a spatial database for large data sets 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 generic 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 [table_name] [column_name];

-- This is only needed for PostgreSQL 7.4 installations and below
SELECT UPDATE_GEOMETRY_STATS([table_name], [column_name]);
```

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 process of being built.

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 statistics are gathered about the number and distributions of values in a table, to provide the query planner with better information to make decisions around index usage. For PostgreSQL 7.4 installations and below this is done by running **update_geometry_stats([table_name, column_name])** (compute distribution) and **VACUUM ANALYZE [table_name] [column_name]** (compute number of values). Starting with PostgreSQL 8.0 running **VACUUM ANALYZE** will do both operations. 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 ENABLE_SEQSCAN=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`.

- If you find the planner wrong about the cost of sequencial vs index scans try reducing the value of random_page_cost in postgresql.conf or using SET random_page_cost=#. Default value for the parameter is 4, try setting it to 1 or 2. Decrementing the value makes the planner more inclined of using Index scans.

4.6. Complex Queries

The *raison d'etre* 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.

4.6.1. Taking Advantage of Indexes

When constructing a query it is important to remember that only the bounding-box-based operators such as && can take advatage of the GiST spatial index. Functions such as `distance()` cannot use the index to optimize their operation. For example, the following query would be quite slow on a large table:

```
SELECT the_geom FROM geom_table
WHERE distance( the_geom, GeomFromText( 'POINT(100000 200000)', -1 ) ) < 100
```

This query is selecting all the geometries in geom_table which are within 100 units of the point (100000, 200000). It will be slow because it is calculating the distance between each point in the table and our specified point, ie. one `distance()` calculation for each row in the table. We can avoid this by using the && operator to reduce the number of distance calculations required:

```
SELECT the_geom FROM geom_table
WHERE the_geom && 'BOX3D(90900 190900, 100100 200100)'::box3d
 AND distance( the_geom, GeomFromText( 'POINT(100000 200000)', -1 ) ) < 100
```

This query selects the same geometries, but it does it in a more efficient way. Assuming there is a GiST index on the_geom, the query planner will recognize that it can use the index to reduce the number of rows before calculating the result of the `distance()` function. Notice that the `BOX3D` geometry which is used in the && operation is a 200 unit square box centered on the original point - this is our "query box". The && operator uses the index to quickly reduce the result set down to only those geometries which have bounding boxes that overlap the "query box". Assuming that our query box is much smaller than the extents of the entire geometry table, this will drastically reduce the number of distance calculations that need to be done.

4.6.2. Examples of Spatial SQL

The examples in this section will make use of two tables, a table of linear roads, and a table of polygonal municipality boundaries. The table definitions for the `bc_roads` table is:

```
 Column   |     Type      |  Description
------------+------------------+------------------
gid      | integer        | Unique ID
name     | character varying | Road Name
the_geom   | geometry       | Location Geometry (Linestring)
```

The table definition for the bc_municipality table is:

```
 Column   |     Type      |  Description
-----------+------------------+------------------
gid     | integer        | Unique ID
code    | integer        | Unique ID
name    | character varying | City / Town Name
the_geom  | geometry       | Location Geometry (Polygon)
```

4.6.2.4.6.2.1.1. What is the total length of all roads, expressed in kilometers?

You can answer this question with a very simple piece of SQL:

```
postgis=# SELECT sum(length(the_geom))/1000 AS km_roads FROM bc_roads;
    km_roads
------------------
 70842.1243039643
(1 row)
```

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

This query combines an attribute condition (on the municipality name) with a spatial calculation (of the area):

```
postgis=# SELECT area(the_geom)/10000 AS hectares FROM bc_municipality
        WHERE name = 'PRINCE GEORGE';
    hectares
------------------
 32657.9103824927
(1 row)
```

4.6.2.4.6.2.1.3. What is the largest municipality in the province, by area?

This query brings a spatial measurement into the query condition. There are several ways of approaching this problem, but the most efficient is below:

```
postgis=# 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)
```

Note that in order to answer this query we have to calculate the area of every polygon. If we were doing this a lot it would make sense to add an area column to the table that we could separately index for performance. By ordering the results in a descending direction, and them using the PostgreSQL "LIMIT" command we can easily pick off the largest value without using an aggregate function like max().

4.6.2.4.6.2.1.4. What is the length of roads fully contained within each municipality?

This is an example of a "spatial join", because we are bringing together data from two tables (doing a join) but using a spatial interaction condition ("contained") as the join condition rather than the usual relational approach of joining on a common key:

```
postgis=# SELECT m.name, sum(length(r.the_geom))/1000 as roads_km
        FROM bc_roads AS r,bc_municipality AS m
        WHERE r.the_geom && m.the_geom
        AND contains(m.the_geom,r.the_geom)
        GROUP BY m.name
        ORDER BY roads_km;

          name          |    roads_km
----------------------------+------------------
 SURREY                 | 1539.47553551242
 VANCOUVER               | 1450.33093486576
 LANGLEY DISTRICT          | 833.793392535662
 BURNABY                | 773.769091404338
 PRINCE GEORGE            |  694.37554369147
 ...
```

This query takes a while, because every road in the table is summarized into the final result (about 250K roads for our particular example table). For smaller overlays (several thousand records on several hundred) the response can be very fast.

4.6.2.4.6.2.1.5. Create a new table with all the roads within the city of Prince George.

This is an example of an "overlay", which takes in two tables and outputs a new table that consists of spatially clipped or cut resultants. Unlike the "spatial join" demonstrated above, this query actually creates new geometries. An overlay is like a turbo-charged spatial join, and is useful for more exact analysis work:

```
postgis=# CREATE TABLE pg_roads as
        SELECT intersection(r.the_geom, m.the_geom) AS intersection_geom,
             length(r.the_geom) AS rd_orig_length,
             r.*
        FROM bc_roads AS r, bc_municipality AS m
        WHERE r.the_geom && m.the_geom
        AND intersects(r.the_geom, m.the_geom)
        AND m.name = 'PRINCE GEORGE';
```

4.6.2.4.6.2.1.6. 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)
```

4.6.2.4.6.2.1.7. What is the largest municipality polygon that has a hole?

```
postgis=# SELECT gid, name, area(the_geom) AS area
       FROM bc_municipality
       WHERE nrings(the_geom) > 1
       ORDER BY area DESC LIMIT 1;
 gid |    name    |     area
-----+-------------+-----------------
  12 | SPALLUMCHEEN | 257374619.430216
(1 row)
```

4.7. Using Mapserver

The Minnesota Mapserver is an internet web-mapping server which conforms to the OpenGIS Web Mapping Server 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.

4.7.1. Basic Usage

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 "widehighways"
  # 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:
* 0.60

* 0.60 CONNECTIONTYPE

  For PostGIS layers, this is always "postgis".

* 0.60 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> host-name=<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.

* 0.60 DATA

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

* 0.60 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 );
```

4.7.2. Frequently Asked Questions

4.7.2.4.7.2.1.1. When I use an `EXPRESSION` in my map file, the condition never returns as true, even though I know the values exist in my table.

Unlike shape files, PostGIS field names have to be referenced in EXPRESSIONS using *lower case*.

```
EXPRESSION ([numlanes] >= 6)
```

4.7.2.4.7.2.1.2. The FILTER I use for my Shape files is not working for my PostGIS table of the same data.

Unlike shape files, filters for PostGIS layers use SQL syntax (they are appended to the SQL statement the PostGIS connector generates for drawing layers in Mapserver).

```
FILTER "type = 'highway' and numlanes >= 4"
```

4.7.2.4.7.2.1.3. My PostGIS layer draws much slower than my Shape file layer, is this normal?

In general, expect PostGIS layers to be 10% slower than equivalent Shape files layers, due to the extra overhead involved in database connections, data transformations and data transit between the database and Mapserver.

If you are finding substantial draw performance problems, it is likely that you have not build a spatial index on your table.

```
postgis# CREATE INDEX geotable_gix ON geotable USING GIST ( geocolumn );
postgis# SELECT update_geometry_stats();  -- For PGSQL < 8.0
postgis# VACUUM ANALYZE;              -- For PGSQL >= 8.0
```

4.7.2.4.7.2.1.4. My PostGIS layer draws fine, but queries are really slow. What is wrong?

For queries to be fast, you must have a unique key for your spatial table and you must have an index on that unique key.

You can specify what unique key for mapserver to use with the `USING UNIQUE` clause in your `DATA` line:

```
DATA "the_geom FROM geotable USING UNIQUE gid"
```

If your table does not have an explicit unique column, you can "fake" a unique column by using the PostgreSQL row "oid" for your unique column. "oid" is the default unique column if you do not declare one, so enhancing your query speed is a matter of building an index on your spatial table oid value.

```
postgis# CREATE INDEX geotable_oid_idx ON geotable (oid);
```

4.7.3. Advanced Usage

The `USING` pseudo-SQL clause is used to add some information to help mapserver understand the results of more complex queries. More specifically, when either a view or a subselect is used as the source table (the thing to the right of "FROM" in a `DATA` definition) it is more difficult for mapserver to automatically determine a unique identifier for each row and also the SRID for the table. The `USING` clause can provide mapserver with these two pieces of information as follows:

```
DATA "the_geom FROM (SELECT table1.the_geom AS the_geom, table1.oid AS oid,↩
table2.data AS data
 FROM table1 LEFT JOIN table2 ON table1.id = table2.id) AS new_table USING↩
UNIQUE oid USING SRID=-1"
```

* 0.60

* 0.60 USING UNIQUE <uniqueid>

Mapserver requires a unique id for each row in order to identify the row when doing map queries. Normally, it would use the oid as the unique identifier, but views and subselects don't automatically have an oid column. If you want to use Mapserver's query functionality, you need to add a unique column to your view or subselect, and declare it with `USING UNIQUE`. For example, you could explicitly select one of the table's oid values for this purpose, or any other column which is guaranteed to be unique for the result set.

The `USING` statement can also be useful even for simple `DATA` statements, if you are doing map queries. It was previously recommended to add an index on the oid column of tables used in query-able layers, in order to speed up the performance of map queries. However, with the `USING` clause, it is possible to tell mapserver to use your table's primary key as the identifier for map queries, and then it is no longer necessary to have an additional index.

Note

"Querying a Map" is the action of clicking on a map to ask for information about the map features in that location.

Don’t confuse "map queries" with the SQL query in a `DATA` definition.

* 0.60 USING SRID=<srid>

PostGIS needs to know which spatial referencing system is being used by the geometries in order to return the correct data back to mapserver. Normally it is possible to find this information in the "geometry_columns" table in the PostGIS database, however, this is not possible for tables which are created on the fly such as subselects and views. So the `USING SRID=` option allows the correct SRID to be specified in the `DATA` definition.

Warning

The parser for Mapserver PostGIS layers is fairly primitive, and is case sensitive in a few areas. Be careful to ensure that all SQL keywords and all your `USING` clauses are in upper case, and that your `USING UNIQUE` clause precedes your `USING SRID` clause.

4.7.4. Examples

Lets start with a simple example and work our way up. Consider the following Mapserver layer definition:

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

This layer will display all the road geometries in the roads table as black lines.

Now lets say we want to show only the highways until we get zoomed in to at least a 1:100000 scale - the next two layers will acheive this effect:

```
LAYER
 CONNECTION "user=theuser password=thepass dbname=thedb host=theserver"
 DATA "the_geom FROM roads"
 MINSCALE 100000
 STATUS ON
 TYPE LINE
 FILTER "road_type = 'highway'"
 CLASS
  COLOR 0 0 0
 END
END

LAYER
 CONNECTION "user=theuser password=thepass dbname=thedb host=theserver"
 DATA "the_geom FROM roads"
 MAXSCALE 100000
 STATUS ON
 TYPE LINE
 CLASSITEM road_type
 CLASS
  EXPRESSION "highway"
  SIZE 2
  COLOR 255 0 0
 END
 CLASS
  COLOR 0 0 0
 END
END
```

The first layer is used when the scale is greater than 1:100000, and displays only the roads of type "highway" as black lines. The `FILTER` option causes only roads of type "highway" to be displayed.

The second layer is used when the scale is less than 1:100000, and will display highways as double-thick red lines, and other roads as regular black lines.

So, we have done a couple of interesting things using only mapserver functionality, but our `DATA` SQL statement has remained simple. Suppose that the name of the road is stored in another table (for whatever reason) and we need to do a join to get it and label our roads.

```
LAYER
 CONNECTION "user=theuser password=thepass dbname=thedb host=theserver"
 DATA "the_geom FROM (SELECT roads.oid AS oid, roads.the_geom AS the_geom,↩
road_names.name as name
  FROM roads LEFT JOIN road_names ON roads.road_name_id =↩
road_names.road_name_id) AS named_roads
  USING UNIQUE oid USING SRID=-1"
 MAXSCALE 20000
 STATUS ON
 TYPE ANNOTATION
 LABELITEM name
 CLASS
  LABEL
   ANGLE auto
   SIZE 8
   COLOR 0 192 0
   TYPE truetype
   FONT arial
  END
 END
END
```

This annotation layer adds green labels to all the roads when the scale gets down to 1:20000 or less. It also demonstrates how to use an SQL join in a `DATA` definition.

4.8. 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 pgsql-specific connection *↩
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 object 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.

4.9. C Clients (libpq)

...

4.9.1. Text Cursors

...

4.9.2. Binary Cursors

...

Chapter 5. Performance tips

5.1. Small tables of large geometries

5.1.1. Problem description

Current PostgreSQL versions (including 8.0) suffer from a query optimizer weakness regarding TOAST tables. TOAST tables are a kind of "extension room" used to store large (in the sense of data size) values that do not fit into normal data pages (like long texts, images or complex geometries with lots of vertices), see http://www.postgresql.org/docs/8.0/static/storage-toast.html for more information).

The problem appears if you happen to have a table with rather large geometries, but not too much rows of them (like a table containing the boundaries of all european countries in high resolution). Then the table itsself is small, but it uses lots of TOAST space. In our example case, the table itsself had about 80 rows and used only 3 data pages, but the TOAST table used 8225 pages.

Now issue a query where you use the geometry operator && to search for a bounding box that matches only very few of those rows. Now the query optimizer sees that the table has only 3 pages and 80 rows. He estimates that a sequential scan on such a small table is much faster than using an index. And so he decides to ignore the GIST index. Usually, this estimation is correct. But in our case, the && operator has to fetch every geometry from disk to compare the bounding boxes, thus reading all TOAST pages, too.

To see whether your suffer from this bug, use the "EXPLAIN ANALYZE" postgresql command. For more information and the technical details, you can read the thread on the postgres performance mailing list: http://archives.postgresql.org/pgsql-performance/2005-02/msg00030.php

5.1.2. Workarounds

The PostgreSQL people are trying to solve this issue by making the query estimation TOAST-aware. For now, here are two workarounds:

The first workaround is to force the query planner to use the index. Send "SET enable_seqscan TO off;" to the server before issuing the query. This basically forces the query planner to avoid sequential scans whenever possible. So it uses the GIST index as usual. But this flag has to be set on every connection, and it causes the query planner to make misestimations in other cases, so you should "SET enable_seqscan TO on;" after the query.

The second workaround is to make the sequential scan as fast as the query planner thinks. This can be achieved by creating an additional column that "caches" the bbox, and matching against this. In our example, the commands are like:

```
SELECT addGeometryColumn('myschema','mytable','bbox','4326','GEOMETRY','2');

UPDATE mytable set bbox = Envelope(Force_2d(the_geom));
```

Now change your query to use the && operator against bbox instead of geom_column, like:

```
SELECT geom_column FROM mytable WHERE bbox && SetSrid('BOX3D(0 0,1 1)'::box3d,4326);
```

Of yourse, if you change or add rows to mytable, you have to keep the bbox "in sync". The most transparent way to do this would be triggers, but you also can modify your application to keep the bbox column current or run the UPDATE query above after every modification.

Chapter 6. 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.

6.1. OpenGIS Functions

6.1.1. Management Functions

* 0.60

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

  Syntax: AddGeometryColumn(<schema_name>, <table_name>, <column_name>, <srid>, <type>, <dimension>). Adds a geometry column to an existing table of attributes. The `schema_name` is the name of the table schema (unused for pre-schema PostgreSQL installations). 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'.

* 0.60 DropGeometryColumn(varchar, varchar, varchar)

  Syntax: DropGeometryColumn(<schema_name>, <table_name>, <column_name>). Remove a geometry column from a spatial table. Note that schema_name will need to match the f_schema_name field of the table's row in the geometry_columns table.

* 0.60 SetSRID(geometry)

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

6.1.2. Geometry Relationship Functions

* 0.60

* 0.60 Distance(geometry,geometry)

  Return the cartesian distance between two geometries in projected units.

* 0.60 Equals(geometry,geometry)

  Returns 1 (TRUE) if this Geometry is "spatially equal" to anotherGeometry. Use this for a 'better' answer than '='. equals ('LINESTRING(0 0, 10 10)','LINESTRING(0 0, 5 5, 10 10)') is true.

  Performed by the GEOS module

  OGC SPEC s2.1.1.2

* 0.60 Disjoint(geometry,geometry)

Returns 1 (TRUE) if this Geometry is "spatially disjoint" from anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 //s2.1.13.3 - a.Relate(b, 'FF*FF****')

* 0.60 Intersects(geometry,geometry)

Returns 1 (TRUE) if this Geometry "spatially intersects" anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 //s2.1.13.3 - Intersects(g1, g2) --> Not (Disjoint(g1, g2))

* 0.60 Touches(geometry,geometry)

Returns 1 (TRUE) if this Geometry "spatially touches" anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 // s2.1.13.3- a.Touches(b) -> (I(a) intersection I(b) = {empty set}) and (a intersection b) not empty

* 0.60 Crosses(geometry,geometry)

Returns 1 (TRUE) if this Geometry "spatially crosses" anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 // s2.1.13.3 - a.Relate(b, 'T*T******')

* 0.60 Within(geometry,geometry)

Returns 1 (TRUE) if this Geometry is "spatially within" anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 // s2.1.13.3 - a.Relate(b, 'T*F**F***')

* 0.60 Overlaps(geometry,geometry)

Returns 1 (TRUE) if this Geometry is "spatially overlapping" anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 // s2.1.13.3

* 0.60 Contains(geometry,geometry)

Returns 1 (TRUE) if this Geometry is "spatially contains" anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 // s2.1.13.3 - same as within(geometry,geometry)

* 0.60 Intersects(geometry,geometry)

Returns 1 (TRUE) if this Geometry is "spatially intersects" anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 // s2.1.13.3 - NOT disjoint(geometry,geometry)

* 0.60 Relate(geometry,geometry, intersectionPatternMatrix)

Returns 1 (TRUE) if this Geometry is spatially related to anotherGeometry, by testing for intersections between the Interior, Boundary and Exterior of the two geometries as specified by the values in the intersectionPatternMatrix.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is the "allowable" version that returns a boolean, not an integer.

OGC SPEC s2.1.1.2 // s2.1.13.3

* 0.60 Relate(geometry,geometry)

returns the DE-9IM (dimensionally extended nine-intersection matrix)

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

not in OGC spec, but implied. see s2.1.13.2

6.1.3. Geometry Processing Functions

* 0.60

* 0.60 Centroid(geometry)

Returns the centroid of the geometry as a point.

Computation will be more accurate if performed by the GEOS module (enabled at compile time).

* 0.60 Area(geometry)

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

* 0.60 Length(geometry)

The length of this Curve in its associated spatial reference.

synonym for length2d()

OGC SPEC 2.1.5.1

* 0.60 PointOnSurface(geometry)

Return a Point guaranteed to lie on the surface

Implemented using GEOS

OGC SPEC 3.2.14.2 and 3.2.18.2 -

* 0.60 Boundary(geometry)

Returns the closure of the combinatorial boundary of this Geometry. The combinatorial boundary is defined as described in section 3.12.3.2 of the OGC SPEC. Because the result of this function is a closure, and hence topologically closed, the resulting boundary can be represented using representational geometry primitives as discussed in the OGC SPEC, section 3.12.2.

Performed by the GEOS module

OGC SPEC s2.1.1.1

* 0.60 Buffer(geometry,double,[integer])

Returns a geometry that represents all points whose distance from this Geometry is less than or equal to distance. Calculations are in the Spatial Reference System of this Geometry. The optional third parameter sets the number of segment used to approximate a quarter circle (defaults to 8).

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.3

* 0.60 ConvexHull(geometry)

Returns a geometry that represents the convex hull of this Geometry.

Performed by the GEOS module

OGC SPEC s2.1.1.3

* 0.60 Intersection(geometry,geometry)

Returns a geometry that represents the point set intersection of this Geometry with anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.3

* 0.60 SymDifference(geometry,geometry)

Returns a geometry that represents the point set symmetric difference of this Geometry with anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.3

* 0.60 Difference(geometry,geometry)

Returns a geometry that represents the point set symmetric difference of this Geometry with anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.3

* 0.60 GeomUnion(geometry,geometry)

Returns a geometry that represents the point set union of this Geometry with anotherGeometry.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

NOTE: this is renamed from "union" because union is an SQL reserved word

OGC SPEC s2.1.1.3

* 0.60 GeomUnion(geometry set)

Returns a geometry that represents the point set union of this all Geometries in given set.

Performed by the GEOS module

Do not call with a GeometryCollection in the argument set

Not explicitly defined in OGC SPEC

* 0.60 MemGeomUnion(geometry set)

Same as the above, only memory-friendly (uses less memory and more processor time).

6.1.4. Geometry Accessors

* 0.60

* 0.60 AsText(geometry)

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

OGC SPEC s2.1.1.1

* 0.60 AsBinary(geometry)

Returns the geometry in the OGC "well-known-binary" format, 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.

OGC SPEC s2.1.1.1 - also see asBinary(<geometry>,'XDR') and asBinary(<geometry>,'NDR')

* 0.60 SRID(geometry)

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

OGC SPEC s2.1.1.1

* 0.60 Dimension(geometry)

The inherent dimension of this Geometry object, which must be less than or equal to the coordinate dimension. OGC SPEC s2.1.1.1 - returns 0 for points, 1 for lines, 2 for polygons, and the largest dimension of the components of a GEOMETRYCOLLECTION.

```
select dimension('GEOMETRYCOLLECTION(LINESTRING(1
1,0 0),POINT(0 0)');
dimension
-----------
1
```

* 0.60 Envelope(geometry)

Returns a POLYGON representing the bounding box of the geometry.

OGC SPEC s2.1.1.1 - The minimum bounding box for this Geometry, returned as a Geometry. The polygon is defined by the corner points of the bounding box ((MINX, MINY), (MAXX, MINY), (MAXX, MAXY), (MINX, MAXY), (MINX, MINY)).

NOTE:PostGIS will add a Zmin/Zmax coordinate as well.

* 0.60 IsEmpty(geometry)

Returns 1 (TRUE) if this Geometry is the empty geometry . If true, then this Geometry represents the empty point set - i.e. GEOMETRYCOLLECTION(EMPTY).

OGC SPEC s2.1.1.1

* 0.60 IsSimple(geometry)

Returns 1 (TRUE) if this Geometry has no anomalous geometric points, such as self intersection or self tangency.

Performed by the GEOS module

OGC SPEC s2.1.1.1

* 0.60 IsClosed(geometry)

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

* 0.60 IsRing(geometry)

Returns 1 (TRUE) if this Curve is closed (StartPoint ( ) = EndPoint ( )) and this Curve is simple (does not pass through the same point more than once).

performed by GEOS

OGC spec 2.1.5.1

* 0.60 NumGeometries(geometry)

If geometry is a GEOMETRYCOLLECTION (or MULTI*) return the number of geometries, otherwise return NULL.

* 0.60 GeometryN(geometry,int)

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

1 is 1st geometry

* 0.60 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.

* 0.60 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.

* 0.60 ExteriorRing(geometry)

Return the exterior ring of the polygon geometry. Return NULL if the geometry is not a polygon.

* 0.60 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.

* 0.60 InteriorRingN(geometry,integer)

Return the N'th interior ring of the polygon geometry. Return NULL if the geometry is not a polygon or the given N is out of range (1-based).

* 0.60 EndPoint(geometry)

Returns the last point of the LineString geometry as a point.

* 0.60 StartPoint(geometry)

Returns the first point of the LineString geometry as a point.

* 0.60 GeometryType(geometry)

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

OGC SPEC s2.1.1.1 - Returns the name of the instantiable subtype of Geometry of which this Geometry instance is a member. The name of the instantiable subtype of Geometry is returned as a string.

* 0.60 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.

* 0.60 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.

* 0.60 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.

6.1.5. Geometry Constructors

* 0.60

* 0.60 GeomFromText(text,[<srid>])

Makes a Geometry from WKT with the given SRID.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

* 0.60 PointFromText(text,[<srid>])

Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

Throws an error if the WKT is not a Point

* 0.60 LineFromText(text,[<srid>])

Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

Throws an error if the WKT is not a Line

* 0.60 LinestringFromText(text,[<srid>])

  Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

  from the conformance suite

  Throws an error if the WKT is not a Line

* 0.60 PolyFromText(text,[<srid>])

  Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

  Throws an error if the WKT is not a Polygon

* 0.60 PolygonFromText(text,[<srid>])

  Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

  from the conformance suite

  Throws an error if the WKT is not a Polygon

* 0.60 MPointFromText(text,[<srid>])

  Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

  Throws an error if the WKT is not a MULTIPOINT

* 0.60 MLineFromText(text,[<srid>])

  Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

  Throws an error if the WKT is not a MULTILINESTRING

* 0.60 MPolyFromText(text,[<srid>])

  Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

  Throws an error if the WKT is not a MULTIPOLYGON

* 0.60 GeomCollFromText(text,[<srid>])

  Makes a Geometry from WKT with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

  Throws an error if the WKT is not a GEOMETRYCOLLECTION

* 0.60 GeomFromWKB(bytea,[<srid>])

  Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

* 0.60 GeomFromWKB(bytea,[<srid>])

  Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

* 0.60 PointFromWKB(bytea,[<srid>])

  Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

  throws an error if WKB is not a POINT

* 0.60 LineFromWKB(bytea,[<srid>])

  Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

  OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

  throws an error if WKB is not a LINESTRING

* 0.60 LinestringFromWKB(bytea,[<srid>])

  Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

  from the conformance suite

  throws an error if WKB is not a LINESTRING

* 0.60 PolyFromWKB(bytea,[<srid>])

Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a POLYGON

* 0.60 PolygonFromWKB(bytea,[<srid>])

Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

from the conformance suite

throws an error if WKB is not a POLYGON

* 0.60 MPointFromWKB(bytea,[<srid>])

Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a MULTIPOINT

* 0.60 MLineFromWKB(bytea,[<srid>])

Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a MULTILINESTRING

* 0.60 MPolyFromWKB(bytea,[<srid>])

Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a MULTIPOLYGON

* 0.60 GeomCollFromWKB(bytea,[<srid>])

Makes a Geometry from WKB with the given SRID. If SRID is not give, it defaults to -1.

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a GEOMETRYCOLLECTION

6.2. Postgis Extensions

6.2.1. Management Functions

* 0.60

* 0.60 DropGeometryTable([<schema_name>], <table_name>)

  Drops a table and all its references in geometry_columns. Note: uses current_schema() on schema-aware pgsql installations if schema is not provided.

* 0.60 UpdateGeometrySRID([<schema_name>], <table_name>, <column_name>, <srid>)

  Update the SRID of all features in a geometry column updating constraints and reference in geometry_columns. Note: uses current_schema() on schema-aware pgsql installations if schema is not provided.

* 0.60 update_geometry_stats([<table_name>, <column_name>])

  Update statistics about spatial tables for use by the query planner. You will also need to run "VACUUM ANALYZE [table_name] [column_name]" for the statistics gathering process to be complete. NOTE: starting with PostgreSQL 8.0 statistics gathering is automatically performed running "VACUUM ANALYZE".

* 0.60 postgis_version()

  Returns the version number of the PostGIS functions installed in this database (deprecated, use postgis_full_version() instead).

* 0.60 postgis_lib_version()

  Returns the version number of the PostGIS library.

* 0.60 postgis_lib_build_date()

  Returns build date of the PostGIS library.

* 0.60 postgis_script_build_date()

  Returns build date of the PostGIS scripts.

* 0.60 postgis_scripts_installed()

  Returns the version number of the lwpostgis.sql script installed in this database.

* 0.60 postgis_scripts_released()

  Returns the version number of the lwpostgis.sql script released with the installed postgis lib.

* 0.60 postgis_geos_version()

  Returns the version number of the GEOS library, or NULL if GEOS support is not enabled.

* 0.60 postgis_proj_version()

  Returns the version number of the PROJ4 library, or NULL if PROJ4 support is not enabled.

* 0.60 postgis_uses_stats()

  Returns true if STATS usage has been enabled, false otherwise.

* 0.60 postgis_full_version()

  Reports full postgis version and build configuration infos.

6.2.2. Operators

* 0.60

* 0.60 A &< B

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

* 0.60 A &> B

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

* 0.60 A << B

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

* 0.60 A >> B

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

* 0.60 A &<| B

  The "&<|" operator returns true if A's bounding box overlaps or is below B's bounding box.

* 0.60 A |&> B

  The "|&>" operator returns true if A's bounding box overlaps or is above B's bounding box.

* 0.60 A <<| B

  The "<<|" operator returns true if A's bounding box is strictly below B's bounding box.

* 0.60 A |>> B

  The "|>>" operator returns true if A's bounding box is strictly above B's bounding box.

* 0.60 A ~= B

  The "~=" operator is the "same as" 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.

* 0.60 A @ B

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

* 0.60 A ~ B

  The "~" operator returns true if A's bounding box completely contains B's bounding box.

* 0.60 A && B

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

6.2.3. Measurement Functions

* 0.60

* 0.60 area2d(geometry)

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

* 0.60 distance_sphere(point, point)

Returns linear distance in meters between two lat/lon points. Uses a spherical earth and radius of 6370986 meters. Faster than distance_spheroid(), but less accurate. Only implemented for points.

* 0.60 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.

* 0.60 length2d(geometry)

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

* 0.60 length3d(geometry)

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

* 0.60 length_spheroid(geometry,spheroid)

Calculates the length of of a geometry on an elipsoid. This is useful if the coordinates of the geometry are in latitude/longitude and a length is desired without reprojection. The elipsoid 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;
```

* 0.60 length3d_spheroid(geometry,spheroid)

Calculates the length of of a geometry on an elipsoid, 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.

* 0.60 distance(geometry, geometry)

Returns the smaller distance between two geometries.

* 0.60 max_distance(linestring,linestring)

Returns the largest distance between two line strings.

* 0.60 perimeter(geometry)

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

* 0.60 perimeter2d(geometry)

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

* 0.60 perimeter3d(geometry)

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

6.2.4. Geometry Outputs

* 0.60

* 0.60 AsBinary(geometry,{'NDR'|'XDR'})

Returns the geometry in the OGC "well-known-binary" format as a bytea, using little-endian (NDR) or big-endian (XDR) encoding. This is useful in binary cursors to pull data out of the database without converting it to a string representation.

* 0.60 AsEWKT(geometry)

Returns a Geometry in EWKT format (as text).

* 0.60 AsEWKB(geometry, {'NDR'|'XDR'})

Returns a Geometry in EWKB format (as bytea) using either little-endian (NDR) or big-endian (XDR) encoding.

* 0.60 AsSVG(geometry, [rel], [precision])

Return the geometry as an SVG path data. Use 1 as second argument to have the path data implemented in terms of relative moves, the default (or 0) uses absolute moves. Third argument may be used to reduce the maximum number of decimal digits used in output (defaults to 15). Point geometries will be rendered as cx/cy when 'rel' arg is 0, x/y when 'rel' is 1.

* 0.60 AsGML(geometry, [precision])

Return the geometry as a GML element. Second argument may be used to reduce the maximum number of significant digits used in output (defaults to 15).

6.2.5. Geometry Constructors

* 0.60

* 0.60 GeomFromEWKT(text)

Makes a Geometry from EWKT.

* 0.60 GeomFromEWKB(bytea)

Makes a Geometry from EWKB.

* 0.60 MakePoint(<x>, <y>, [<z>], [<m>])

Creates a 2d,3dz or 4d point geometry.

* 0.60 MakePointM(<x>, <y>, <m>)

Creates a 3dm point geometry.

* 0.60 MakeBox2D(<LL>, <UR>)

Creates a BOX2D defined by the given point geometries.

* 0.60 MakeBox3D(<LLB>, <URT>)

Creates a BOX3D defined by the given point geometries.

* 0.60 MakeLine(geometry set)

Creates a Linestring from a set of point geometries. You might want to use a subselect to order points before feeding them to this aggregate.

* 0.60 MakeLine(geometry, geometry)

Creates a Linestring from the two given point geometries.

* 0.60 LineFromMultiPoint(multipoint)

Creates a LineString from a MultiPoint geometry.

* 0.60 AddPoint(linestring, point, [<position>])

Adds a point to a LineString at position <pos>. Third parameter can be omitted or set to -1 for appending.

* 0.60 MakePolygon(linestring, [linestring[]])

Creates a Polygon formed by the given shell and array of holes. You can construct a geometry array using Accum. Input geometries must be closed LINESTRINGS (see IsClosed and GeometryType).

* 0.60 Polygonize(geometry set)

Aggregate. Creates a GeometryCollection containing possible polygons formed from the costituent linework of a set of geometries. Only available when compiled against GEOS >= 2.1.0.

* 0.60 Collect(geometry set)

This function returns a GEOMETRYCOLLECTION or a MULTI object from a set of geometries. The collect() function is an "aggregate" function in the terminology of PostgreSQL. That means that it operators 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.

* 0.60 Collect(geometry, geometry)

This function returns a geometry being a collection of two input geometries. Output type can be a MULTI* or a GEOMETRYCOLLECTION.

* 0.60 Dump(geometry)

This is a set-returning function (SRF). It returns a set of geometry_dump rows, formed by a geometry (geom) and an array of integers (path). When the input geometry is a simple type (POINT,LINESTRING,POLYGON) a single record will be returned with an empty path array and the input geometry as geom. When the input geometry is a collection or multi it will return a record for each of the collection components, and the path will express the position of the component inside the collection.

NOTE: this function is not available for builds against PostgreSQL 7.2.x

6.2.6. Geometry Editors

* 0.60

* 0.60 AddBBOX(geometry)

Add bounding box to the geometry. This would make bounding box based queries faster, but will increase the size of the geometry.

* 0.60 DropBBOX(geometry)

Drop the bounding box cache from the geometry. This reduces geometry size, but makes bounding-box based queries slower.

* 0.60 Force_collection(geometry)

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

* 0.60 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).

* 0.60 Force_3dz(geometry), Force_3d(geometry)

Forces the geometries into XYZ mode.

* 0.60 Force_3dm(geometry)

Forces the geometries into XYM mode.

* 0.60 Force_4d(geometry)

Forces the geometries into XYZM mode.

* 0.60 Multi(geometry)

Returns the geometry as a MULTI* geometry. If the geometry is already a MULTI*, it is returned unchanged.

* 0.60 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.

* 0.60 Translate(geometry,float8,float8,float8)

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

* 0.60 Reverse(geometry)

Returns the geometry with vertex order reversed.

* 0.60 ForceRHR(geometry)

Force polygons of the collection to obey Right-Hand-Rule.

* 0.60 Simplify(geometry, tolerance)

Returns a "simplified" version of the given geometry using the Douglas-Peuker algorithm. Will actually do something only with (multi)lines and (multi)polygons but you can safely call it with any kind of geometry. Since simplification occurs on a object-by-object basis you can also feed a GeometryCollection to this function. Note that returned geometry might loose its simplicity (see IsSimple)

* 0.60 SnapToGrid(geometry, originX, originY, sizeX, sizeY), SnapToGrid(geometry, sizeX, sizeY), SnapToGrid(geometry, size)

Snap all points of the input geometry to the grid defined by its origin and cell size. Remove consecutive points falling on the same cell, eventually returning NULL if output points are not enough to define a geometry of the given type. Collapsed geometries in a collection are stripped from it. Note that returned geometry might loose its simplicity (see IsSimple).

* 0.60 Segmentize(geometry, maxlength)

Return a modified [multi]polygon having no ring segment longer then the given distance. Interpolated points will have Z and M values (if needed) set to 0. Distance computation is performed in 2d only.

6.2.7. Misc

* 0.60

* 0.60 Summary(geometry)

Returns a text summary of the contents of the geometry.

* 0.60 box2d(geometry)

  Returns a BOX2D representing the maximum extents of the geometry.

* 0.60 box3d(geometry)

  Returns a BOX3D representing the maximum extents of the geometry.

* 0.60 extent(geometry set)

  The extent() function is an "aggregate" function in the terminology of PostgreSQL. That means that it operators 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 extend of all features in the table. Similarly, "SELECT EXTENT(GEOM) FROM GEOMTABLE GROUP BY CATEGORY" will return one extent result for each category.

* 0.60 zmflag(geometry)

  Returns ZM (dimension semantic) flag of the geometries as a small int. Values are: 0=2d, 1=3dm, 2=3dz, 3=4d.

* 0.60 HasBBOX(geometry)

  Returns TRUE if the bbox of this geometry is cached, FALSE otherwise. Use addBBOX() and dropBBOX() to control caching.

* 0.60 ndims(geometry)

  Returns number of dimensions of the geometry as a small int. Values are: 2,3 or 4.

* 0.60 nrings(geometry)

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

* 0.60 npoints(geometry)

  Returns the number of points in the geometry.

* 0.60 isvalid(geometry)

  returns true if this geometry is valid.

* 0.60 expand(geometry, float)

  This function returns a bounding box expanded in all directions from the bounding box of the input geometry, by an amount specified in the second argument. Very useful for distance() queries, to add an index filter to the query.

* 0.60 estimated_extent([schema], table, geocolumn)

Return the 'estimated' extent of the given spatial table. The estimated is taken from the geometry column's statistics gathered by running VACUUM ANALYZE, and is statistically about 95% of 'real' extent. The current schema will be used if not specified.

NOTE: This is only available with PostgreSQL >= 8.0.0

* 0.60 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.

* 0.60 mem_size(geometry)

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

* 0.60 numb_sub_objects(geometry)

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

* 0.60 point_inside_circle(geometry,float,float,float)

The syntax for this functions is point_inside_circle(<geometry>,<circle_cen
Returns the true if the geometry is a point and is inside the circle. Returns false otherwise.

* 0.60 xmin(box3d) ymin(box3d) zmin(box3d)

Returns the requested minima of a bounding box.

* 0.60 xmax(box3d) ymax(box3d) zmax(box3d)

Returns the requested maxima of a bounding box.

* 0.60 line_interpolate_point(geometry, proportion)

Interpolates a point along a line. First argument must be a LINESTRING. Second argument is a float between 0 and 1. Returns a point.

* 0.60 Accum(geometry set)

Aggregate. Constructs an array of geometries.

Chapter 7. Release Notes

7.1. Release 1.0.0

This is a major postgis release, with internal storage of postgis types redesigned to be smaller and faster on indexed queries.

7.1.1. Upgrading

You need a dump/reload to upgrade from precedent releases. See the upgrading chapter for more informations.

7.1.2. Changes

Faster canonical input parsing.

Lossless canonical output.

EWKB Canonical binary IO with PG>73.

Support for up to 4d coordinates, providing lossless shapefile->postgis->shapefile conversion.

New function: UpdateGeometrySRID(), AsGML(), SnapToGrid(), ForceRHR(), estimated_extent(), accum().

Vertical positioning indexed operators.

JOIN selectivity function.

More geometry constructors / editors.

Postgis extension API.

UTF8 support in loader.

7.1.3. Credits

This release includes contributions from (in alphabetical order): Carl Anderson, David Blasby, IIDA Tetsushi, Kris Jurka, Mark Cave-Ayland, Markus Schaber, Ralph Mason.