

PostGIS 1.3.4 Manual

Abstract

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.

This is the manual for version 1.3.4

Table of Contents

1. Introduction

- 1.1. Credits
- 1.2. More Information

2. Installation

- 2.1. Requirements
- 2.2. PostGIS
 - 2.2.1. Creating PostGIS spatially-enabled databases from an in-built template
 - 2.2.2. Upgrading
 - 2.2.3. Common Problems
- 2.3. JDBC
- 2.4. Loader/Dumper

3. Frequently Asked Questions

4. Using PostGIS

- 4.1. GIS Objects
 - 4.1.1. OpenGIS WKB and WKT
 - 4.1.2. PostGIS EWKB, EWKT and Canonical Forms
 - 4.1.3. SQL-MM Part 3
- 4.2. Using OpenGIS Standards
 - 4.2.1. The SPATIAL_REF_SYS Table
 - 4.2.2. The GEOMETRY_COLUMNS Table
 - 4.2.3. Creating a Spatial Table
 - 4.2.4. Ensuring OpenGIS compliancy of geometries
- 4.3. Loading GIS Data
 - 4.3.1. Using SQL
 - 4.3.2. Using the Loader

4.4. Retrieving GIS Data

4.4.1. Using SQL

4.4.2. Using the Dumper

4.5. Building Indexes

4.5.1. GiST Indexes

4.5.2. Using Indexes

4.6. Complex Queries

4.6.1. Taking Advantage of Indexes

4.6.2. Examples of Spatial SQL

4.7. Using Mapserver

4.7.1. Basic Usage

4.7.2. Frequently Asked Questions

4.7.3. Advanced Usage

4.7.4. Examples

4.8. Java Clients (JDBC)

4.9. C Clients (libpq)

4.9.1. Text Cursors

4.9.2. Binary Cursors

5. Performance tips

5.1. Small tables of large geometries

5.1.1. Problem description

5.1.2. Workarounds

5.2. CLUSTERing on geometry indices

5.3. Avoiding dimension conversion

6. PostGIS Reference

6.1. OpenGIS Functions

6.1.1. Management Functions

6.1.2. Geometry Relationship Functions

6.1.3. Geometry Processing Functions

6.1.4. Geometry Accessors

6.1.5. Geometry Constructors

6.2. PostGIS Extensions

6.2.1. Management Functions

6.2.2. Operators

6.2.3. Measurement Functions

6.2.4. Geometry Outputs

6.2.5. Geometry Constructors

- 6.2.6. Geometry Editors
- 6.2.7. Linear Referencing
- 6.2.8. Misc
- 6.2.9. Long Transactions support

6.3. SQL-MM Functions

6.4. ArcSDE Functions

7. Reporting Problems

- 7.1. Reporting Software Bugs
- 7.2. Reporting Documentation Issues

A. Appendix

A.1. Release Notes

- A.1.1. Release 1.3.3
- A.1.2. Release 1.3.2
- A.1.3. Release 1.3.1
- A.1.4. Release 1.3.0
- A.1.5. Release 1.2.1
- A.1.6. Release 1.2.0
- A.1.7. Release 1.1.6
- A.1.8. Release 1.1.5
- A.1.9. Release 1.1.4
- A.1.10. Release 1.1.3
- A.1.11. Release 1.1.2
- A.1.12. Release 1.1.1
- A.1.13. Release 1.1.0
- A.1.14. Release 1.0.6
- A.1.15. Release 1.0.5
- A.1.16. Release 1.0.4
- A.1.17. Release 1.0.3
- A.1.18. Release 1.0.2
- A.1.19. Release 1.0.1
- A.1.20. Release 1.0.0
- A.1.21. Release 1.0.0RC6
- A.1.22. Release 1.0.0RC5
- A.1.23. Release 1.0.0RC4
- A.1.24. Release 1.0.0RC3
- A.1.25. Release 1.0.0RC2
- A.1.26. Release 1.0.0RC1

Chapter 1. Introduction

Table of Contents

- 1.1. Credits
- 1.2. More Information

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

Sandro Santilli <strk@refractions.net>

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

Mark Leslie <mleslie@refractions.net>

Ongoing maintenance and development of core functions.

Chris Hodgson <chodgson@refractions.net>

Maintains new functions and the 7.2 index bindings.

Paul Ramsey <pramsey@refractions.net>

Keeps track of the documentation and packaging.

Jeff Lounsbury <jeffloun@refractions.net>

Original development of the Shape file loader/dumper.

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.

Other contributors

In alphabetical order: Alex Bodnaru, Alex Mayrhofer, Bruce Rindahl, Bernhard Reiter, Bruno Wolff III, Carl Anderson, Charlie Savage, David Skea, David Techer, IIDA Tetsushi, Geographic Data BC, Gerald Fenoy, Gino Lucrezi, Klaus Foerster, Kris Jurka, Mark Cave-Ayland, Mark Sondheim, Markus Schaber, Michael Fuhr, Nikita Shulga, Norman Vine, Olivier Courtin, Ralph Mason, Steffen Macke.

Important Support Libraries

The [GEOS](#) geometry operations library, and the algorithmic work of Martin Davis <mbdavis@vividolutions.com> of Vivid Solutions in making it all work.

The [Proj4](#) cartographic projection library, and the work of Gerald Evenden and Frank Warmerdam in creating and maintaining it.

1.2. More Information

- The latest software, documentation and news items are available at the PostGIS web site, <http://postgis.refractory.net>.
- More information about the GEOS geometry operations library is available at <http://geos.refractory.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/~megeera/postgres/gist>.
- More information about Mapserver internet map server is available at <http://mapserver.gis.umn.edu>.
- The "Simple Features for Specification for SQL" is available at the OpenGIS Consortium web site: <http://www.opengis.org>.

Chapter 2. Installation

Table of Contents

2.1. Requirements

2.2. PostGIS

2.2.1. Creating PostGIS spatially-enabled databases from an in-built template

2.2.2. Upgrading

2.2.3. Common Problems

2.3. JDBC

2.4. Loader/Dumper

2.1. Requirements

PostGIS has the following requirements for building and usage:

- A complete installation of PostgreSQL (including server headers). PostgreSQL is available from <http://www.postgresql.org>. Version 7.2 or higher is required.
- 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 (`ST_Touches()`, `ST_Contains()`, `ST_Intersects()`) and operations (`ST_Buffer()`, `ST_Union()`, `ST_Difference()`) within PostGIS. GEOS is available for download from <http://geos.refractive.net>.

2.2. PostGIS

The PostGIS module is an extension to the PostgreSQL backend server. As such, PostGIS 1.3.4 *requires* full PostgreSQL server headers access in order to compile. The PostgreSQL source code is available at <http://www.postgresql.org>.

PostGIS 1.3.4 can be built against PostgreSQL versions 7.2.0 or higher. 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:

```
LDLFLAGS=-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.refractive.net/postgis-1.3.4.tar.gz>. Uncompress and untar the archive.

```
# gzip -d -c postgis-1.3.4.tar.gz | tar xvf -
```

3. Enter the `postgis-1.3.4` directory, and run:

```
# ./configure
```

- If you want support for coordinate reprojection, you must have the Proj4 library installed. If `./configure` didn't find it, try using `--with-proj=PATH` switch specify a specific Proj4 installation directory.
- If you want to use GEOS functionality, you must have the GEOS library installed. If `./configure` didn't find it, try using `--with-geos=PATH` to specify the full path to the `geos-config` program full path.

4. Run the compile and install commands.

```
# make # make install
```

All files are installed using information provided by `pg_config`

- Libraries are installed `[pkglibdir]/lib/contrib`.
- Important support files such as `lwpostgis.sql` are installed in `[prefix]/share/contrib`.
- Loader and dumper binaries are installed in `[bindir]/`.

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. Creating PostGIS spatially-enabled databases from an in-built template

Some packaged distributions of PostGIS (in particular the Win32 installers for PostGIS \geq 1.1.5) load the PostGIS functions into a template database called `template_postgis`. If the `template_postgis` database exists in your PostgreSQL installation then it is possible for users and/or applications to create spatially-enabled databases using a single command. Note

that in both cases, the database user must have been granted the privilege to create new databases.

From the shell:

```
# createdb -T template_postgis my_spatial_db
```

From SQL:

```
postgres=# CREATE DATABASE my_spatial_db TEMPLATE=template_postgis
```

2.2.2. Upgrading

Upgrading existing spatial databases can be tricky as it requires replacement or introduction of new PostGIS object definitions.

Unfortunately not all definitions can be easily replaced in a live database, so sometimes your best bet is a dump/reload process.

PostGIS provides a SOFT UPGRADE procedure for minor or bugfix releases, and an HARD UPGRADE procedure for major releases.

Before attempting to upgrade postgis, it is always worth to backup your data. If you use the `-Fc` flag to `pg_dump` you will always be able to restore the dump with an HARD UPGRADE.

2.2.2.1. Soft upgrade

Soft upgrade consists of sourcing the `lwpostgis_upgrade.sql` script in your spatial database:

```
$ psql -f lwpostgis_upgrade.sql -d your_spatial_database
```

If a soft upgrade is not possible the script will abort and you will be warned about HARD UPGRADE being required, so do not hesitate to try a soft upgrade first.

Note

If you can't find the `lwpostgis_upgrade.sql` file you are probably using a version prior to 1.1 and must generate that file by yourself. This is done with the following command:

```
$ utils/postgis_proc_upgrade.pl lwpostgis.sql > lwpostgis_upgrade.sql
```

2.2.2.2. Hard upgrade

By HARD UPGRADE we intend full dump/reload of postgis-enabled databases. You need an HARD UPGRADE when postgis objects' internal storage changes or when SOFT UPGRADE is not possible. The [Release Notes](#) appendix reports for each version whether you need a dump/reload (HARD UPGRADE) to upgrade.

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. `postgis_restore` accepts `createdb` parameters after the dump file name, and that can for instance be used if you are using a non-default character encoding for your database. Let's call it "newdb", with UNICOD as the character encoding:

```
$ sh utils/postgis_restore.pl lwpostgis.sql newdb olddb.dump -E=UNICODE > 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 < 8.0 to >= 8.0 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 spatial_ref_sys;
DROP
newdb=> \i spatial_ref_sys.sql
```

2.2.3. 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 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) distribution 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.config`. 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.config`.
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.config`

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-1.3.4/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. For more verbose documentation, see the online help, and the manual pages.

Chapter 3. Frequently Asked Questions

- 3.1. What kind of geometric objects can I store?
- 3.2. How do I insert a GIS object into the database?
- 3.3. How do I construct a spatial query?
- 3.4. How do I speed up spatial queries on large tables?
- 3.5. Why aren't PostgreSQL R-Tree indexes supported?
- 3.6. Why should I use the `AddGeometryColumn()` function and all the other OpenGIS stuff?
- 3.7. What is the best way to find all objects within a radius of another object?
- 3.8. How do I perform a coordinate reprojection as part of a query?

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(2 3,4 5,6 5,7 8)', -1)
);
```

For more information about other GIS objects, see the [object reference](#).

To view your GIS data in the table:

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

The return value should look something like this:

```
id | name      | geom
----+-----+-----
 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()`, `ST_Intersects()`, `ST_Contains()` and `ST_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
  _ST_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 `ST_DWithin(geometry, geometry, distance)` function is a handy way of performing an indexed distance search. It works by creating a search rectangle large enough to enclose the distance radius, then performing an exact distance search on the indexed subset of results.

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

```
SELECT * FROM geotable
WHERE ST_DWithin(geocolumn, 'POINT(1000 1000)', 100.0);
```

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 ST_Transform(the_geom,4269) FROM geotable;
```

Chapter 4. Using PostGIS

Table of Contents

4.1. GIS Objects

- 4.1.1. OpenGIS WKB and WKT
- 4.1.2. PostGIS EWKB, EWKT and Canonical Forms
- 4.1.3. SQL-MM Part 3

4.2. Using OpenGIS Standards

- 4.2.1. The `SPATIAL_REF_SYS` Table
- 4.2.2. The `GEOMETRY_COLUMNS` Table
- 4.2.3. Creating a Spatial Table
- 4.2.4. Ensuring OpenGIS compliancy of geometries

4.3. Loading GIS Data

- 4.3.1. Using SQL
- 4.3.2. Using the Loader

4.4. Retrieving GIS Data

- 4.4.1. Using SQL

4.4.2. Using the Dumper

4.5. Building Indexes

4.5.1. GiST Indexes

4.5.2. Using Indexes

4.6. Complex Queries

4.6.1. Taking Advantage of Indexes

4.6.2. Examples of Spatial SQL

4.7. Using Mapserver

4.7.1. Basic Usage

4.7.2. Frequently Asked Questions

4.7.3. Advanced Usage

4.7.4. Examples

4.8. Java Clients (JDBC)

4.9. C Clients (libpq)

4.9.1. Text Cursors

4.9.2. Binary Cursors

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 geotable ( 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 geotable ( 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
-----
0101000020040000000000000000000000000000000000000000000000000000
(1 row)
```

4.1.3. SQL-MM Part 3

The SQL Multimedia Applications Spatial specification extends the simple features for SQL spec by defining a number of circularly interpolated curves.

The SQL-MM definitions include 3dm, 3dz and 4d coordinates, but do not allow the embedding of SRID information.

The well-known text extensions are not yet fully supported. Examples of some simple curved geometries are shown below:

- CIRCULARSTRING(0 0, 1 1, 1 0)

- COMPOUNDCURVE(CIRCULARSTRING(0 0, 1 1, 1 0),(1 0, 0 1))
- CURVEPOLYGON(CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),(1 1, 3 3, 3 1, 1 1))
- MULTICURVE((0 0, 5 5),CIRCULARSTRING(4 0, 4 4, 8 4))
- MULTISURFACE(CURVEPOLYGON(CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),(1 1, 3 3, 3 1, 1 1)),((10 10, 14 12, 11 10, 10 10),(11 11, 11.5 11, 11 11.5, 11 11)))

Note

Currently, PostGIS cannot support the use of Compound Curves in a Curve Polygon.

Note

All floating point comparisons within the SQL-MM implementation are performed to a specified tolerance, currently 1E-8.

4.2. Using OpenGIS Standards

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

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

4.2.1. The SPATIAL_REF_SYS Table

The `SPATIAL_REF_SYS` table definition is as follows:

```
CREATE TABLE spatial_ref_sys (
  srid    INTEGER NOT NULL PRIMARY KEY,
  auth_name VARCHAR(256),
  auth_srid INTEGER,
  srtext  VARCHAR(2048),
  proj4text VARCHAR(2048)
)
```

The `SPATIAL_REF_SYS` columns are as follows:

SRID

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

AUTH_NAME

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

AUTH_SRID

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

SRTEXT

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

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

For a listing of EPSG projection codes and their corresponding WKT representations, see <http://www.opengis.org/techno/interop/EPG2WKT.TXT>. For a discussion of WKT in general, see the OpenGIS "Coordinate Transformation Services Implementation Specification" at <http://www.opengis.org/techno/specs.htm>. For information on the European Petroleum Survey Group (EPSG) and their database of spatial reference systems, see <http://epsg.org>.

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  VARRCHAR(256) NOT NULL,
  f_table_schema   VARCHAR(256) NOT NULL,
  f_table_name     VARCHAR(256) NOT NULL,
  f_geometry_column VARCHAR(256) NOT NULL,
  coord_dimension  INTEGER NOT NULL,
  srid             INTEGER NOT NULL,
  type            VARCHAR(30) NOT NULL
)
```

The columns are as follows:

F_TABLE_CATALOG, F_TABLE_SCHEMA, F_TABLE_NAME

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

F_GEOMETRY_COLUMN

The name of the geometry column in the feature table.

COORD_DIMENSION

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

SRID

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

TYPE

The type of the spatial object. To restrict the spatial column to a single type, use one of: `POINT`, `LINestring`, `POLYGON`, `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 INTEGER,  
  park_name VARCHAR,  
  park_date DATE,
```

```

park_type VARCHAR
);
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 INTEGER,
  road_name VARCHAR
);
SELECT AddGeometryColumn( 'roads', 'roads_geom', -1, 'GEOMETRY', 3 );

```

4.2.4. Ensuring OpenGIS compliancy of geometries

Most of the functions implemented by the GEOS library rely on the assumption that your geometries are valid as specified by the OpenGIS Simple Feature Specification. To check validity of geometries you can use the `IsValid()` function:

```

gisdb=# select isvalid('LINESTRING(0 0, 1 1)'),
        isvalid('LINESTRING(0 0,0 0)');

```

```

isvalid | isvalid
-----+-----
t       | f

```

By default, PostGIS does not apply this validity check on geometry input, because testing for validity needs lots of CPU time for complex geometries, especially polygons. If you do not trust your data sources, you can manually enforce such a check to your tables by adding a check constraint:

```

ALTER TABLE mytable
  ADD CONSTRAINT geometry_valid_check
  CHECK (isvalid(the_geom));

```

If you encounter any strange error messages such as "GEOS Intersection() threw an error!" or "JTS Intersection() threw an error!" when calling PostGIS functions with valid input geometries, you likely found an error in either PostGIS or one of the libraries it uses, and you should contact the PostGIS developers. The same is true if a PostGIS function returns an invalid geometry for valid input.

Note

Strictly compliant OGC geometries cannot have Z or M values. The `IsValid()` function won't consider higher dimensioned geometries invalid! Invocations of `AddGeometryColumn()` will add a constraint checking geometry

dimensions, so it is enough to specify 2 there.

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 (road_id, roads_geom, road_name)
  VALUES (1,GeomFromText('LINESTRING(191232 243118,191108 243242)',-1),'Jeff Rd');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (2,GeomFromText('LINESTRING(189141 244158,189265 244817)',-1),'Geordie Rd');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (3,GeomFromText('LINESTRING(192783 228138,192612 229814)',-1),'Paul St');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (4,GeomFromText('LINESTRING(189412 252431,189631 259122)',-1),'Graeme Ave');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (5,GeomFromText('LINESTRING(190131 224148,190871 228134)',-1),'Phil Tce');
INSERT INTO roads (road_id, roads_geom, road_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:

-d

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

-a

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

-c

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

-p

Only produces the table creation SQL code, without adding any actual data. This can be used if you need to completely separate the table creation and data loading steps.

-D

Use the PostgreSQL "dump" format for the output data. This can be combined with **-a**, **-c** and **-d**. It is much faster to load than the default "insert" SQL format. Use this for very large data sets.

-s <SRID>

Creates and populates the geometry tables with the specified SRID.

-k

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

-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.

-l

Create a GiST index on the geometry column.

-w

Output WKT format, for use with older (0.x) versions of PostGIS. Note that this will introduce coordinate drifts and will drop M values from shapefiles.

-W <encoding>

Specify encoding of the input data (dbf file). When used, all attributes of the dbf are converted from the specified encoding to UTF8. The resulting SQL output will contain a `SET CLIENT_ENCODING to UTF8` command, so that the backend will be able to reconvert from UTF8 to whatever encoding the database is configured to use internally.

Note that **-a**, **-c**, **-d** and **-p** are mutually exclusive.

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 road_id, AsText(road_geom) AS geom, road_name FROM roads;
```

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

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

&&

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

~=

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

=

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

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

```
SELECT road_id, road_name
FROM roads
WHERE roads_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 road_id, road_name
FROM roads
WHERE roads_geom && GeomFromText('POLYGON((...))',-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(roads_geom) AS geom
FROM roads
WHERE
roads_geom && SetSRID('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:

-f <filename>

Write the output to a particular filename.

-h <host>

The database host to connect to.

-p <port>

The port to connect to on the database host.

-P <password>

The password to use when connecting to the database.

-u <user>

The username to use when connecting to the database.

-g <geometry column>

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

-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.

-r

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

-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] );
```

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

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 advantage 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 ST_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, i.e. one `ST_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
ST_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.

Change in Behavior

As of PostGIS 1.3.0, most of the Geometry Relationship Functions, with the notable exceptions of `ST_Disjoint` and `ST_Relate`, include implicit bounding box overlap operators.

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)

- 4.6.2.1. What is the total length of all roads, expressed in kilometers?
- 4.6.2.2. How large is the city of Prince George, in hectares?
- 4.6.2.3. What is the largest municipality in the province, by area?
- 4.6.2.4. What is the length of roads fully contained within each municipality?
- 4.6.2.5. Create a new table with all the roads within the city of Prince George.
- 4.6.2.6. What is the length in kilometers of "Douglas St" in Victoria?
- 4.6.2.7. What is the largest municipality polygon that has a hole?

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

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

```
SELECT sum(ST_Length(the_geom))/1000 AS km_roads FROM bc_roads;
```

```
km_roads
-----
70842.1243039643
(1 row)
```

4.6.2.2. 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):

```
SELECT
  ST_Area(the_geom)/10000 AS hectares
FROM bc_municipality
WHERE name = 'PRINCE GEORGE';
```

```
hectares
-----
32657.9103824927
(1 row)
```

4.6.2.3. 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:

```
SELECT
  name,
  ST_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 then using the PostgreSQL "LIMIT" command we can easily pick off the largest value without using an aggregate function like max().

4.6.2.4. 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:

```
SELECT
  m.name,
  sum(ST_Length(r.the_geom))/1000 as roads_km
FROM
  bc_roads AS r,
  bc_municipality AS m
WHERE
  ST_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.

4.6.2.5. 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:

```
CREATE TABLE pg_roads as
SELECT
  ST_Intersection(r.the_geom, m.the_geom) AS intersection_geom,
  ST_Length(r.the_geom) AS rd_orig_length,
  r.*
FROM
  bc_roads AS r,
```



```
bc_municipality AS m
WHERE ST_Intersects(r.the_geom, m.the_geom)
AND m.name = 'PRINCE GEORGE';
```

4.6.2.6. What is the length in kilometers of "Douglas St" in Victoria?

```
SELECT
  sum(ST_Length(r.the_geom))/1000 AS kilometers
FROM
  bc_roads r,
  bc_municipality m
WHERE ST_Contains(m.the_geom, r.the_geom)
AND r.name = 'Douglas St'
AND m.name = 'VICTORIA';

kilometers
-----
4.89151904172838
(1 row)
```

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

```
SELECT gid, name, ST_Area(the_geom) AS area
FROM bc_municipality
WHERE ST_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:

CONNECTIONTYPE

For PostGIS layers, this is always "postgis".

CONNECTION

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

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

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

DATA

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

FILTER

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

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

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

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

4.7.2.1. 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.

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

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

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

4.7.2.1. 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)
```

4.7.2.2. 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"
```

4.7.2.3. 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. 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"
```

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.

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 achieve 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
ENDI
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 postgres-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

Table of Contents

5.1. Small tables of large geometries

5.1.1. Problem description

5.1.2. Workarounds

5.2. CLUSTERing on geometry indices

5.3. Avoiding dimension conversion

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 itself is small, but it uses lots of TOAST space. In our example case, the table itself 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 you 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 && ST_SetSRID('BOX3D(0 0,1 1)::box3d,4326);
```

Of course, 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.

5.2. CLUSTERing on geometry indices

For tables that are mostly read-only, and where a single index is used for the majority of queries, PostgreSQL offers the CLUSTER command. This command physically reorders all the data rows in the same order as the index criteria, yielding two performance advantages: First, for index range scans, the number of seeks on the data table is drastically reduced. Second, if your working set concentrates to some small intervals on the indices, you have a more efficient caching because the data rows are spread along fewer data pages. (Feel invited to read the CLUSTER command documentation from the PostgreSQL manual at this point.)

However, currently PostgreSQL does not allow clustering on PostGIS GIST indices because GIST indices simply ignores NULL values, you get an error message like:

```
lwgeom=# CLUSTER my_geom_index ON my_table;
ERROR: cannot cluster when index access method does not handle null values
HINT: You may be able to work around this by marking column "the_geom" NOT NULL.
```

As the HINT message tells you, one can work around this deficiency by adding a "not null" constraint to the table:

```
lwgeom=# ALTER TABLE my_table ALTER COLUMN the_geom SET not null;
ALTER TABLE
```

Of course, this will not work if you in fact need NULL values in your geometry column. Additionally, you must use the above method to add the constraint, using a CHECK constraint like "ALTER TABLE blubb ADD CHECK (geometry is not null);" will not work.

5.3. Avoiding dimension conversion

Sometimes, you happen to have 3D or 4D data in your table, but always access it using OpenGIS compliant asText() or asBinary() functions that only output 2D geometries. They do this by internally calling the force_2d() function, which introduces a significant overhead for large geometries. To avoid this overhead, it may be feasible to pre-drop those additional dimensions once and forever:

```
UPDATE mytable SET the_geom = force_2d(the_geom);
VACUUM FULL ANALYZE mytable;
```

Note that if you added your geometry column using AddGeometryColumn() there'll be a constraint on geometry dimension. To bypass it you will need to drop the constraint. Remember to update the entry in the geometry_columns table and recreate the constraint afterwards.

In case of large tables, it may be wise to divide this UPDATE into smaller portions by constraining the UPDATE to a part of the table via a WHERE clause and your primary key or another feasible criteria, and running a simple "VACUUM;" between your UPDATES. This drastically reduces the need for temporary disk space. Additionally, if you have mixed dimension geometries, restricting the UPDATE by "WHERE dimension(the_geom)>2" skips re-writing of geometries that already are in 2D.

Chapter 6. PostGIS Reference

Table of Contents

6.1. OpenGIS Functions

- 6.1.1. Management Functions
- 6.1.2. Geometry Relationship Functions
- 6.1.3. Geometry Processing Functions
- 6.1.4. Geometry Accessors
- 6.1.5. Geometry Constructors

6.2. PostGIS Extensions

- 6.2.1. Management Functions
- 6.2.2. Operators
- 6.2.3. Measurement Functions
- 6.2.4. Geometry Outputs
- 6.2.5. Geometry Constructors
- 6.2.6. Geometry Editors
- 6.2.7. Linear Referencing
- 6.2.8. Misc
- 6.2.9. Long Transactions support

6.3. SQL-MM Functions

6.4. ArcSDE Functions

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.

Note

PostGIS has begun a transition from the existing naming convention to an SQL-MM-centric convention. As a result, most of the functions that you know and love have been renamed using the standard spatial type (ST) prefix. Previous functions are still available, though are not listed in this document where updated functions are equivalent. These will be deprecated in a future release.

6.1. OpenGIS Functions

6.1.1. Management Functions

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

Syntax: AddGeometryColumn(<schema_name>, <table_name>, <column_name>, <sruid>, <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 `sruid` must be an integer value reference to an entry in the SPATIAL_REF_SYS table. The `type` must be an uppercase string corresponding to the geometry type, eg, 'POLYGON' or 'MULTILINESTRING'.

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

Syntax: DropGeometryColumn(<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.

ST_SetSRID(*geometry*, *integer*)

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

6.1.2. Geometry Relationship Functions

ST_Distance(*geometry*, *geometry*)

Return the cartesian distance between two geometries in projected units. Does not use indexes.

ST_DWithin(*geometry*, *geometry*, *float*)

Returns true if geometries are within the specified distance of one another. Uses indexes if available.

ST_Equals(*geometry*, *geometry*)

Returns 1 (TRUE) if the given Geometries are "spatially equal". 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

ST_Disjoint(*geometry*, *geometry*)

Returns 1 (TRUE) if the Geometries are "spatially disjoint".

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****')

ST_Intersects(geometry, geometry)

Returns 1 (TRUE) if the Geometries "spatially intersect".

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. To avoid index use, use the function `_ST_Intersects`.

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))

ST_Touches(geometry, geometry)

Returns 1 (TRUE) if the Geometries "spatially touch".

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. To avoid index use, use the function `_ST_Touches`.

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) -> ((a intersection b) = {empty set}) and (a intersection b) not empty

ST_Crosses(geometry, geometry)

Returns 1 (TRUE) if the Geometries "spatially cross".

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. To avoid index use, use the function `_ST_Crosses`.

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*****')

ST_Within(geometry A, geometry B)

Returns 1 (TRUE) if Geometry A is "spatially within" Geometry B. A has to be completely

inside B.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. To avoid index use, use the function `_ST_Within`.

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***')`

ST_Overlaps(geometry, geometry)

Returns 1 (TRUE) if the Geometries "spatially overlap".

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. To avoid index use, use the function `_ST_Overlaps`.

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

OGC SPEC s2.1.1.2 // s2.1.13.3

ST_Contains(geometry A, geometry B)

Returns 1 (TRUE) if Geometry A "spatially contains" Geometry B.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. To avoid index use, use the function `_ST_Contains`.

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 B, geometry A)`

ST_Covers(geometry A, geometry B)

Returns 1 (TRUE) if no point in Geometry B is outside Geometry A

Refer to <http://lin-ear-th-inking.blogspot.com/2007/06/subtleties-of-ogc-covers-spatial.html> for an explanation of the need of this function.

This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. To avoid index use, use the function `_ST_Covers`.

ST_CoveredBy(geometry A, geometry B)

Returns 1 (TRUE) if no point in Geometry A is outside Geometry B

Refer to <http://lin-ear-th-inking.blogspot.com/2007/06/subtleties-of-ogc-covers-spatial.html> for an explanation of the need of this function.

ST_Intersects(geometry, geometry)

Returns 1 (TRUE) if the Geometries "spatially intersect".

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)

ST_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

ST_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

ST_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).

ST_Area(geometry)

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

ST_Length(geometry)

The length of this Curve in its associated spatial reference.

synonym for length2d()

OGC SPEC 2.1.5.1

ST_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 –

ST_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

ST_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.

OGC SPEC s2.1.1.3

ST_ConvexHull(geometry)

The convex hull of a geometry represents the minimum closed geometry that encloses all geometries within the set.

It is usually used with MULTI and Geometry Collections. Although it is not an aggregate – you can use it in conjunction with ST_Collect to get the convex hull of a set of points. ST_ConvexHull(ST_Collect(somepointfield)). It is often used to determine an affected area based on a set of point observations.

```
SELECT d.disease_type, ST_ConvexHull(ST_Collect(d.the_geom)) As the_geom
FROM disease_obs As d
GROUP BY d.disease_type
```

Performed by the GEOS module

OGC SPEC s2.1.1.3

ST_Intersection(geometry, geometry)

Returns a geometry that represents the point set intersection of the Geometries.

In other words – that portion of geometry A and geometry B that is shared between the two geometries.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.3

ST_Shift_Longitude(geometry)

Reads every point/vertex in every component of every feature in a geometry, and if the longitude coordinate is <0, adds 360 to it. The result would be a 0–360 version of the data to be plotted in a 180 centric map

ST_SymDifference(geometry A, geometry B)

Returns a geometry that represents the portions of A and B that do not intersect. It is called a symmetric difference because $ST_SymDifference(A,B) = ST_SymDifference(B,A)$.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.3

ST_Difference(geometry A, geometry B)

Returns a geometry that represents that part of geometry A that does not intersect with geometry B.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.3

ST_Union(geometry, geometry)

Returns a geometry that represents the point set union of the Geometries.

Performed by the GEOS module.

Do not call with a GeometryCollection as an argument.

NOTE: this function was formerly called `GeomUnion()`, which was renamed from "Union" because UNION is an SQL reserved word.

OGC SPEC s2.1.1.3

ST_Union(geometry set)

Returns a geometry that represents the point set union of 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.

ST_MemUnion(geometry set)

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

6.1.4. Geometry Accessors

ST_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

ST_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')

ST_SRID(geometry)

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

OGC SPEC s2.1.1.1

ST_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
```

ST_Envelope(geometry)

Returns a valid geometry (POINT, LINESTRING or POLYGON) representing the bounding box of the geometry. Degenerate cases (vertical lines, point) will return a geometry of lower dimension than POLYGON.

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.

ST_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

ST_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

ST_IsClosed(geometry)

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

ST_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

ST_NumGeometries(geometry)

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

ST_GeometryN(geometry,int)

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

Note

Index is 1-based as for OGC specs since version 0.8.0. Previous versions implemented this as 0-based instead.

ST_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.

ST_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.

Note

Index is 1-based as for OGC specs since version 0.8.0. Previous versions implemented this as 0-based instead.

ST_ExteriorRing(geometry)

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

ST_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.

ST_NumInteriorRing(geometry)

Synonym to NumInteriorRings(geometry). The OpenGIS specs are ambiguous about the exact function naming, so we provide both spellings.

ST_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.

Note

Index is 1-based as for OGC specs since version 0.8.0. Previous versions implemented this as 0-based instead.

ST_EndPoint(geometry)

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

ST_StartPoint(geometry)

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

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.

Note

This function also indicates if the geometry is measured, by returning a string of the form 'POINTM'.

ST_GeometryType(geometry)

Returns the type of the geometry as a string. EG: 'Linestring', 'Polygon', etc. This function differs from GeometryType(geometry) in the case of the string that is returned, as well as the fact that it will not indicate whether the geometry is measured.

ST_X(geometry)

Return the X coordinate of the point. Input must be a point.

ST_Y(geometry)

Return the Y coordinate of the point. Input must be a point.

ST_Z(geometry)

Return the Z coordinate of the point, or NULL if not available. Input must be a point.

ST_M(geometry)

Return the M coordinate of the point, or NULL if not available. Input must be a point.

Note

This is not (yet) part of the OGC spec, but is listed here to complete the point coordinate extractor function list.

6.1.5. Geometry Constructors

ST_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

ST_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

ST_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

ST_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

ST_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

ST_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

ST_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

ST_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

ST_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

ST_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

ST_GeomFromWKB(bytea,[<sruid>])

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

ST_GeometryFromWKB(bytea,[<sruid>])

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

ST_PointFromWKB(bytea,[<sruid>])

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

ST_LineFromWKB(bytea,[<sruid>])

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

ST_LinestringFromWKB(bytea,[<sruid>])

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

ST_PolyFromWKB(bytea,[<sruid>])

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

ST_PolygonFromWKB(bytea,[<sruid>])

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

ST_MPointFromWKB(bytea,[<sruid>])

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

ST_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

ST_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

ST_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

ST_BdPolyFromText(text WKT, integer SRID)

Construct a Polygon given an arbitrary collection of closed linestrings as a MultiLineString text representation.

Throws an error if WKT is not a MULTILINESTRING. Throws an error if output is a MULTIPOLYGON; use [BdMPolyFromText](#) in that case, or see [BuildArea\(\)](#) for a postgis-specific approach.

OGC SFSQL 1.1 – 3.2.6.2

Availability: 1.1.0 – requires GEOS >= 2.1.0.

ST_BdMPolyFromText(text WKT, integer SRID)

Construct a MultiPolygon given an arbitrary collection of closed linestrings as a MultiLineString text representation.

Throws an error if WKT is not a MULTILINESTRING. Forces MULTIPOLYGON output even when result is really only composed by a single POLYGON; use [BdPolyFromText](#) if you're sure a single POLYGON will result from operation, or see [BuildArea\(\)](#) for a postgis-specific approach.

OGC SFSQL 1.1 – 3.2.6.2

Availability: 1.1.0 – requires GEOS >= 2.1.0.

6.2. PostGIS Extensions

6.2.1. Management Functions

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

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

UpdateGeometrySRID([<schema_name>], <table_name>, <column_name>, <sruid>)

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

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".

postgis_version()

Returns PostGIS version number and compile-time options

Note

Prior to version 1.1.0 this was a procedural function, thus possibly returning inaccurate information (in case of incomplete database upgrades).

postgis_lib_version()

Returns the version number of the PostGIS library.

Availability: 0.9.0

postgis_lib_build_date()

Returns build date of the PostGIS library.

Availability: 1.0.0RC1

postgis_script_build_date()

Returns build date of the PostGIS scripts.

Availability: 1.0.0RC1

postgis_scripts_installed()

Returns version of the postgis scripts installed in this database.

Note

If the output of this function doesn't match the output of `postgis_scripts_released()` you probably missed to properly upgrade an existing database. See the [Upgrading](#) section for more info.

Availability: 0.9.0

`postgis_scripts_released()`

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

Note

Starting with version 1.1.0 this function returns the same value of `postgis_lib_version()`. Kept for backward compatibility.

Availability: 0.9.0

`postgis_geos_version()`

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

Availability: 0.9.0

`postgis_jts_version()`

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

Availability: 1.1.0

`postgis_proj_version()`

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

Availability: 0.9.0

`postgis_uses_stats()`

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

Availability: 0.9.0

`postgis_full_version()`

Reports full postgis version and build configuration infos.

Availability: 0.9.0

6.2.2. Operators

A &< B

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

A &> B

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

A << B

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

A >> B

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

A &<| B

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

A |&> B

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

A <<| B

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

A |>> B

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

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.

A @ B

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

A ~ B

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

A && B

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

6.2.3. Measurement Functions

ST_area2d(geometry)

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

ST_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.

ST_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.

ST_length2d(geometry)

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

ST_length3d(geometry)

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

ST_length_spheroid(geometry,spheroid)

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

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

Eg:

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

An example calculation might look like this:

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

ST_length3d_spheroid(geometry,spheroid)

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

ST_distance(geometry, geometry)

Returns the smaller distance between two geometries.

ST_max_distance(linestring,linestring)

Returns the largest distance between two line strings.

ST_perimeter(geometry)

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

ST_perimeter2d(geometry)

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

ST_perimeter3d(geometry)

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

ST_azimuth(geometry, geometry)

Returns the azimuth of the segment defined by the given Point geometries, or NULL if the two points are coincident. Return value is in radians.

Availability: 1.1.0

6.2.4. Geometry Outputs

ST_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.

ST_AsEWKT(geometry)

Returns a Geometry in EWKT format (as text).

ST_AsEWKB(geometry, { 'NDR'|'XDR' })

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

ST_AsHEXEWKB(geometry, {'NDR'|'XDR'})

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

ST_AsSVG(geometry, [rel], [precision])

Return the geometry as 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. Multipoint geometries are delimited by commas (","), GeometryCollection geometries are delimited by semicolons (";").

ST_AsGML([version], geometry, [precision])

Return the geometry as a GML element. The version parameter, if specified, may be either 2 or 3. If no version parameter is specified then the default is assumed to be 2. The third argument may be used to reduce the maximum number of significant digits used in output (defaults to 15).

ST_AsKML(geometry, [precision])

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

ST_AsGeoJson([version], geometry, [precision], [options])

Return the geometry as a GeoJson element. (Cf [GeoJson specifications 1.0](#)). 2D and 3D Geometries are both supported. GeoJson only support SFS 1.1 geometry type (no curve support for example).

The version parameter, if specified, must be 1.

The third argument may be used to reduce the maximum number of significant digits used in output (defaults to 15).

The last 'options' argument could be used to add Bbox or Crs in GeoJSON output:

- 0: means no option (default value)
- 1: Geojson CRS
- 2: Geojson Bbox
- 3: Both Geojson Bbox and CRS

Geojson CRS pattern generated is: `auth_name:auth_srid` from `spatial_ref_sys` table (EPSG:4326 for instance).

6.2.5. Geometry Constructors

ST_GeomFromEWKT(text)

Makes a Geometry from EWKT.

ST_GeomFromEWKB(bytea)

Makes a Geometry from EWKB.

ST_MakePoint(<x>, <y>, [<z>], [<m>])

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

ST_MakePointM(<x>, <y>, <m>)

Creates a 3dm point geometry.

ST_MakeBox2D(<LL>, <UR>)

Creates a BOX2D defined by the given point geometries.

ST_MakeBox3D(<LLB>, <URT>)

Creates a BOX3D defined by the given point geometries.

ST_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.

ST_MakeLine(geometry, geometry)

Creates a Linestring from the two given point geometries.

ST_LineFromMultiPoint(multipoint)

Creates a LineString from a MultiPoint geometry.

ST_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](#)).

ST_BuildArea(geometry)

Creates an areal geometry formed by the constituent linework of given geometry. The return type can be a Polygon or MultiPolygon, depending on input. If the input lineworks do not form polygons NULL is returned.

See also [BdPolyFromText](#) and [BdMPolyFromText](#) – wrappers to this function with standard OGC interface.

Availability: 1.1.0 – requires GEOS >= 2.1.0.

ST_Polygonize(geometry set)

Aggregate. Creates a GeometryCollection containing possible polygons formed from the

constituent linework of a set of geometries.

Availability: 1.0.0RC1 – requires GEOS >= 2.1.0.

ST_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 operates on rows 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.

ST_Collect and ST_Union are often interchangeable. ST_Collect is in general orders of magnitude faster than ST_Union because it does not try to dissolve boundaries. It merely rolls up single geometries into MULTI and MULTI or mixed geometry types into Geometry Collections. Unfortunately geometry collections are not well-supported by GIS tools. To prevent ST_Collect from returning a Geometry Collection when collecting MULTI geometries, one can use the below trick that utilizes ST_Dump to expand the MULTIs out to singles and then regroup them.

```
Thread ref: http://postgis.refrations.net/pipermail/postgis-users/2008-June/020331.html
SELECT stusps,
       ST_Multi(ST_Collect(f.the_geom)) as singlegeom
FROM (SELECT stusps, (ST_Dump(the_geom)).geom As the_geom
      FROM
      somestatetable ) As f
GROUP BY stusps
```

ST_Collect(geometry, geometry)

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

ST_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.

ST_Dump is useful for expanding geometries. It is the reverse of a GROUP BY in that it creates new rows. For example it can be use to expand MULTIPOLYGONS into POLYGONS.

```
SELECT sometable.*, (ST_Dump(the_geom)).geom As the_geom
FROM somestatetable
```

Availability: PostGIS 1.0.0RC1. Requires PostgreSQL 7.3 or higher.

ST_DumpRings(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). The 'path' field holds the polygon ring index, contains a single element: 0 for the shell, hole number for holes. The 'geom' field contains the corresponding ring as a polygon.

Availability: PostGIS 1.1.3. Requires PostgreSQL 7.3 or higher.

6.2.6. Geometry Editors

ST_AddBBOX(geometry)

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

ST_DropBBOX(geometry)

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

ST_AddPoint(linestring, point, [<position>])

Adds a point to a LineString before point <pos> (0-based index). Third parameter can be omitted or set to -1 for appending.

ST_RemovePoint(linestring, offset)

Removes point from a linestring. Offset is 0-based.

Availability: 1.1.0

ST_SetPoint(linestring, N, point)

Replace point N of linestring with given point. Index is 0-based.

Availability: 1.1.0

ST_Force_collection(geometry)

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

ST_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).

ST_Force_3dz(geometry), ST_Force_3d(geometry)

Forces the geometries into XYZ mode.

ST_Force_3dm(geometry)

Forces the geometries into XYM mode.

ST_Force_4d(geometry)

Forces the geometries into XYZM mode.

ST_Multi(geometry)

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

ST_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.

ST_Affine(geometry, float8, float8, float8, float8, float8, float8, float8, float8, float8, float8, float8)

Applies an 3d affine transformation to the geometry. The call

```
Affine(geom, a, b, c, d, e, f, g, h, i, xoff, yoff, zoff)
```

represents the transformation matrix

```
/ a b c xoff \  
| d e f yoff |  
| g h i zoff |  
\ 0 0 0 1 /
```

and the vertices are transformed as follows:

```
x' = a*x + b*y + c*z + xoff  
y' = d*x + e*y + f*z + yoff  
z' = g*x + h*y + i*z + zoff
```

All of the translate / scale functions below are expressed via such an affine transformation.

Availability: 1.1.2.

ST_Affine(geometry, float8, float8, float8, float8, float8)

Applies an 2d affine transformation to the geometry. The call

```
Affine(geom, a, b, d, e, xoff, yoff)
```

represents the transformation matrix

```
/ a b 0 xoff \  
/ a b xoff \  
| d e yoff |  
\ 0 0 1 /
```

```
| d e 0 yoff | rsp. | d e yoff |
| 0 0 1 0 | \ 0 0 1 /
\ 0 0 0 1 /
```

and the vertices are transformed as follows:

```
x' = a*x + b*y + xoff
y' = d*x + e*y + yoff
z' = z
```

This method is a subcase of the 3D method above.

Availability: 1.1.2.

ST_Translate(geometry, float8, float8, float8)

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

ST_Scale(geometry, float8, float8, float8)

scales the geometry to a new size by multiplying the ordinates with the parameters. I.e: `scale(geom, Xfactor, Yfactor, Zfactor)`.

Availability: 1.1.0

ST_RotateZ(geometry, float8), ST_RotateX(geometry, float8), ST_RotateY(geometry, float8)

Rotate the geometry around the Z, X or Y axis by the given angle given in radians. Follows the right-hand rule.

Availability: 1.1.2.

ST_TransScale(geometry, float8, float8, float8, float8)

First, translates the geometry using the first two floats, then scales it using the second two floats, working in 2D only. Using `transscale(geom, X, Y, XFactor, YFactor)` internally calls `affine(geom, XFactor, 0, 0, 0, YFactor, 0, 0, 0, 1, X*XFactor, Y*YFactor, 0)`.

Availability: 1.1.0.

ST_Reverse(geometry)

Returns the geometry with vertex order reversed.

ST_ForceRHR(geometry)

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

ST_Simplify(geometry, tolerance)

Returns a "simplified" version of the given geometry using the Douglas-Peucker 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 lose its simplicity (see [IsSimple](#))

ST_SimplifyPreserveTopology(geometry, tolerance)

Returns a "simplified" version of the given geometry using the Douglas–Peucker algorithm. Will avoid creating derived geometries (polygons in particular) that are invalid.

ST_SnapToGrid(geometry, originX, originY, sizeX, sizeY), ST_SnapToGrid(geometry, sizeX, sizeY), ST_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

The returned geometry might lose its simplicity (see [IsSimple](#)).

Note

Before release 1.1.0 this function always returned a 2d geometry. Starting at 1.1.0 the returned geometry will have same dimensionality as the input one with higher dimension values untouched. Use the version taking a second geometry argument to define all grid dimensions.

Availability: 1.0.0RC1

ST_SnapToGrid(geometry, geometry, sizeX, sizeY, sizeZ, sizeM)

Snap all points of the input geometry to the grid defined by its origin (the second argument, must be a point) and cell sizes. Specify 0 as size for any dimension you don't want to snap to a grid.

Availability: 1.1.0

ST_Segmentize(geometry, maxlength)

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

ST_LineMerge(geometry)

Returns a (set of) `LineString`(s) formed by sewing together constituent linework of input.

Availability: 1.1.0 – requires GEOS \geq 2.1.0

6.2.7. Linear Referencing

ST_line_interpolate_point(linestring, location)

Returns a point interpolated along a line. First argument must be a `LINestring`. Second argument is a float8 between 0 and 1 representing fraction of total `2d length` the point has to be located.

See `line_locate_point()` for computing the line location nearest to a Point.

Note

Since release 1.1.1 this function also interpolates M and Z values (when present), while prior releases set them to 0.0.

Availability: 0.8.2

ST_line_substring(linestring, start, end)

Return a linestring being a substring of the input one starting and ending at the given fractions of total `2d length`. Second and third arguments are float8 values between 0 and 1.

If 'start' and 'end' have the same value this is equivalent to `line_interpolate_point()`.

See `line_locate_point()` for computing the line location nearest to a Point.

Note

Since release 1.1.1 this function also interpolates M and Z values (when present), while prior releases set them to unspecified values.

Availability: 1.1.0

ST_line_locate_point(LineString, Point)

Returns a float between 0 and 1 representing the location of the closest point on `LineString` to the given `Point`, as a fraction of total `2d line length`.

You can use the returned location to extract a `Point` (`line_interpolate_point`) or a substring (`line_substring`).

Availability: 1.1.0

ST_locate_along_measure(geometry, float8)

Return a derived geometry collection value with elements that match the specified

measure. Polygonal elements are not supported.

Semantic is specified by: ISO/IEC CD 13249-3:200x(E) – Text for Continuation CD Editing Meeting

Availability: 1.1.0

ST_locate_between_measures(geometry, float8, float8)

Return a derived geometry collection value with elements that match the specified range of measures inclusively. Polygonal elements are not supported.

Semantic is specified by: ISO/IEC CD 13249-3:200x(E) – Text for Continuation CD Editing Meeting

Availability: 1.1.0

6.2.8. Misc

ST_Summary(geometry)

Returns a text summary of the contents of the geometry.

ST_box2d(geometry)

Returns a BOX2D representing the maximum extents of the geometry.

ST_box3d(geometry)

Returns a BOX3D representing the maximum extents of the geometry.

ST_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.

ST_zmflag(geometry)

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

ST_HasBBOX(geometry)

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

ST_ndims(geometry)

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

ST_nrings(geometry)

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

ST_npoints(geometry)

Returns the number of points in the geometry.

ST_isvalid(geometry)

returns true if this geometry is valid.

ST_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.

ST_estimated_extent([schema], table, geocolumn)

Return the 'estimated' extent of the given spatial table. The estimated is taken from the geometry column's statistics. The current schema will be used if not specified.

For PostgreSQL >= 8.0.0 statistics are gathered by VACUUM ANALYZE and resulting extent will be about 95% of the real one.

For PostgreSQL < 8.0.0 statistics are gathered by update_geometry_stats() and resulting extent will be exact.

ST_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.

ST_mem_size(geometry)

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

ST_point_inside_circle(geometry, float, float, float)

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

ST_XMin(box3d) ST_YMin(box3d) ST_ZMin(box3d)

Returns the requested minima of a bounding box.

ST_XMax(box3d) ST_YMax(box3d) ST_ZMax(box3d)

Returns the requested maxima of a bounding box.

ST_Accum(geometry set)

Aggregate. Constructs an array of geometries.

6.2.9. Long Transactions support

This module and associated pl/pgsql functions have been implemented to provide long locking support required by [Web Feature Service](#) specification.

Note

Users must use `serializable transaction level` otherwise locking mechanism would break.

EnableLongTransactions()

Enable long transaction support. This function creates the required metadata tables, needs to be called once before using the other functions in this section. Calling it twice is harmless.

Availability: 1.1.3

DisableLongTransactions()

Disable long transaction support. This function removes the long transaction support metadata tables, and drops all triggers attached to lock-checked tables.

Availability: 1.1.3

CheckAuth([<schema>], <table>, <rowid_col>)

Check updates and deletes of rows in given table for being authorized. Identify rows using <rowid_col> column.

Availability: 1.1.3

LockRow([<schema>], <table>, <rowid>, <authid>, [<expires>])

Set lock/authorization for specific row in table <authid> is a text value, <expires> is a timestamp defaulting to now()+1hour. Returns 1 if lock has been assigned, 0 otherwise (already locked by other auth)

Availability: 1.1.3

UnlockRows(<authid>)

Remove all locks held by specified authorization id. Returns the number of locks released.

Availability: 1.1.3

AddAuth(<authid>)

Add an authorization token to be used in current transaction.

Availability: 1.1.3

6.3. SQL-MM Functions

This is a listing of the SQL-MM defined functions that PostGIS currently supports. The implementations of these functions follow the ArcSDE implementation, and thus deviate somewhat from the spec. These deviations will be noted.

As of version 1.2.0, these functions have been implemented by wrapping existing PostGIS functions. As a result, full support for curved geometries may not be in place for many functions.

Note

SQL-MM defines the default SRID of all geometry constructors as 0. PostGIS uses a default SRID of -1.

ST_Area

Return the area measurement of an ST_Surface or ST_MultiSurface value.

SQL-MM 3: 8.1.2, 9.5.3

ST_AsBinary

Return the well-known binary representation of an ST_Geometry value.

SQL-MM 3: 5.1.37

ST_AsText

Return the well-known text representation of an ST_Geometry value.

SQL-MM 3: 5.1.25

ST_Boundary

Return the boundary of the ST_Geometry value.

SQL-MM 3: 5.1.14

ST_Buffer

Return a buffer around the ST_Geometry value.

SQL-MM 3: 5.1.17

ST_Centroid

Return mathematical centroid of the ST_Surface or ST_MultiSurface value.

SQL-MM 3: 8.1.4, 9.5.5

ST_Contains

Test if an ST_Geometry value spatially contains another ST_Geometry value.

SQL-MM 3: 5.1.31

ST_ConvexHull

The convex hull of a geometry represents the minimum geometry that encloses all geometries within the set.

It is usually used with MULTI and Geometry Collections. Although it is not an aggregate – you can use it in conjunction with ST_Collect to get the convex hull of a set of points. ST_ConvexHull(ST_Collect(somepointfield)). It is often used to determine an affected area based on a set of point observations.

SQL-MM 3: 5.1.16

ST_CoordDim

Return the coordinate dimension of the ST_Geometry value.

SQL-MM 3: 5.1.3

ST_Crosses

Test if an ST_Geometry value spatially crosses another ST_Geometry value.

SQL-MM 3: 5.1.29

ST_Difference

Return an ST_Geometry value that represents the point set difference of two ST_Geometry values.

SQL-MM 3: 5.1.20

ST_Dimension

Return the dimension of the ST_Geometry value.

SQL-MM 3: 5.1.2

ST_Disjoint

Test if an ST_Geometry value is spatially disjoint from another ST_Geometry value.

SQL-MM 3: 5.1.26

ST_Distance

Return the distance between two geometries.

SQL-MM 3: 5.1.23

ST_EndPoint

Return an ST_Point value that is the end point of an ST_Curve value.

SQL-MM 3: 7.1.4

ST_Envelope

Return the bounding rectangle for the ST_Geometry value.

SQL-MM 3: 5.1.15

ST_Equals

Test if an ST_Geometry value is spatially equal to another ST_Geometry value.

SQL-MM 3: 5.1.24

ST_ExteriorRing

Return the exterior ring of an ST_Surface

SQL-MM 3: 8.2.3, 8.3.3

ST_GeometryN

Return the indicated ST_Geometry value from an ST_GeomCollection.

SQL-MM 3: 9.1.5

ST_GeometryType

Return the geometry type of the ST_Geometry value.

SQL-MM 3: 5.1.4

ST_GeomFromText

Return a specified ST_Geometry value.

SQL-MM 3: 5.1.40

ST_GeomFromWKB

Return a specified ST_Geometry value.

SQL-MM 3: 5.1.41

ST_InteriorRingN

Return the specified interior ring of an ST_Surface value.

SQL-MM 3: 8.2.6, 8.3.5

ST_Intersection

Return an ST_Geometry value that represents the point set intersection of two ST_Geometry values.

In other words – that portion of geometry A and geometry B that is shared between the two geometries.

SQL-MM 3: 5.1.18

ST_Intersects

Test if an ST_Geometry value spatially intersects another ST_Geometry value.

SQL-MM 3: 5.1.27

ST_IsClosed

Test if an ST_Curve or ST_MultiCurve value is closed.

Note

SQL-MM defines the result of ST_IsClosed(NULL) to be 0, while PostGIS returns NULL.

SQL-MM 3: 7.1.5, 9.3.3

ST_IsEmpty

Test if an ST_Geometry value corresponds to the empty set.

Note

SQL-MM defines the result of ST_IsEmpty(NULL) to be 0, while PostGIS returns NULL.

SQL-MM 3: 5.1.7

ST_IsRing

Test if an ST_Curve value is a ring.

Note

SQL-MM defines the result of ST_IsRing(NULL) to be 0, while PostGIS returns NULL.

SQL-MM 3: 7.1.6

ST_IsSimple

Test if an ST_Geometry value has no anomalous geometric points, such as self intersection or self tangency.

Note

SQL-MM defines the result of `ST_IsSimple(NULL)` to be 0, while PostGIS returns NULL.

SQL-MM 3: 5.1.8

ST_IsValid

Test if an `ST_Geometry` value is well formed.

Note

SQL-MM defines the result of `ST_IsValid(NULL)` to be 0, while PostGIS returns NULL.

SQL-MM defines the result of `ST_IsValid(NULL)` to be 1

SQL-MM 3: 5.1.9

ST_Length

Return the length measurement of an `ST_Curve` or `ST_MultiCurve` value.

SQL-MM 3: 7.1.2, 9.3.4

ST_LineFromText

Return a specified `ST_LineString` value.

SQL-MM 3: 7.2.8

ST_LineFromWKB

Return a specified `ST_LineString` value.

SQL-MM 3: 7.2.9

ST_MLineFromText

Return a specified `ST_MultiLineString` value.

SQL-MM 3: 9.4.4

ST_MLineFromWKB

Return a specified `ST_MultiLineString` value.

SQL-MM 3: 9.4.5

ST_MPointFromText

Return a specified ST_MultiPoint value.

SQL-MM 3: 9.2.4

ST_MPointFromWKB

Return a specified ST_MultiPoint value.

SQL-MM 3: 9.2.5

ST_MPolyFromText

Return a specified ST_MultiPolygon value.

SQL-MM 3: 9.6.4

ST_MPolyFromWKB

Return a specified ST_MultiPolygon value.

SQL-MM 3: 9.6.5

ST_NumGeometries

Return the number of geometries in an ST_GeomCollection.

SQL-MM 3: 9.1.4

ST_NumInteriorRing

Return the number of interior rings in an ST_Surface.

SQL-MM 3: 8.2.5

ST_NumPoints

Return the number of points in an ST_LineString or ST_CircularString value.

SQL-MM 3: 7.2.4

ST_OrderingEquals

ST_OrderingEquals compares two geometries and t (TRUE) if the geometries are equal and the coordinates are in the same order; otherwise it returns f (FALSE).

Note

This function is implemented as per the ArcSDE SQL specification rather than SQL-MM.
http://edndoc.esri.com/arcscde/9.1/sql_api/sqlapi3.htm#ST_OrderingEqu

SQL-MM 3: 5.1.43

ST_Overlaps

Test if an ST_Geometry value spatially overlaps another ST_Geometry value.

SQL-MM 3: 5.1.32

ST_Perimeter

Return the length measurement of the boundary of an ST_Surface or ST_MultiSurface value.

SQL-MM 3: 8.1.3, 9.5.4

ST_Point

Returns an ST_Point with the given coordinate values.

SQL-MM 3: 6.1.2

ST_PointFromText

Return a specified ST_Point value.

SQL-MM 3: 6.1.8

ST_PointFromWKB

Return a specified ST_Point value.

SQL-MM 3: 6.1.9

ST_PointN

Return the specified ST_Point value in an ST_LineString or ST_CircularString

SQL-MM 3: 7.2.5, 7.3.5

ST_PointOnSurface

Return an ST_Point value guaranteed to spatially intersect the ST_Surface or ST_MultiSurface value.

SQL-MM 3: 8.1.5, 9.5.6

ST_PolyFromText

Return a specified ST_Polygon value.

SQL-MM 3: 8.3.6

ST_PolyFromWKB

Return a specified ST_Polygon value.

SQL-MM 3: 8.3.7

ST_Polygon

Return a polygon build from the specified linestring and SRID.

SQL-MM 3: 8.3.2

ST_Relate

Test if an ST_Geometry value is spatially related to another ST_Geometry value.

SQL-MM 3: 5.1.25

ST_SRID

Return the spatial reference system identifier of the ST_Geometry value.

SQL-MM 3: 5.1.5

ST_StartPoint

Return an ST_Point value that is the start point of an ST_Curve value.

SQL-MM 3: 7.1.3

ST_SymDifference

Return an ST_Geometry value that represents the point set symmetric difference of two ST_Geometry values.

SQL-MM 3: 5.1.21

ST_Touches

Test if an ST_Geometry value spatially touches another ST_Geometry value.

SQL-MM 3: 5.1.28

ST_Transform

Return an ST_Geometry value transformed to the specified spatial reference system.

SQL-MM 3: 5.1.6

ST_Union

Return an ST_Geometry value that represents the point set union of two ST_Geometry values.

SQL-MM 3: 5.1.19

ST_Within

Test if an ST_Geometry value is spatially within another ST_Geometry value.

SQL-MM 3: 5.1.30

ST_WKBToSQL

Return an ST_Geometry value for a given well-known binary representation.

SQL-MM 3: 5.1.36

ST_WKTTToSQL

Return an ST_Geometry value for a given well-known text representation.

SQL-MM 3: 5.1.34

ST_X

Returns the x coordinate value of an ST_Point value.

SQL-MM 3: 6.1.3

ST_Y

Returns the y coordinate value of an ST_Point value.

SQL-MM 3: 6.1.4

6.4. ArcSDE Functions

Additional functions have been added to improve support for an ArcSDE style interface.

SE_EnvelopesIntersect

Returns t (TRUE) if the envelopes of two geometries intersect; otherwise, it returns f (FALSE).

SE_Is3d

Test if a geometry value has z coordinate values.

SE_IsMeasured

Test if a geometry value has m coordinate values.

SE_LocateAlong

Return a derived geometry collection value with elements that match the specified measur.

SE_LocateBetween

Return a derived geometry collection value with elements that match the specified range of measures inclusively.

SE_M

Returns the m coordinate value of an ST_Point value.

SE_Z

Returns the z coordinate value of an ST_Point value

Chapter 7. Reporting Problems

Table of Contents

- 7.1. Reporting Software Bugs
- 7.2. Reporting Documentation Issues

7.1. Reporting Software Bugs

Reporting bugs effectively is a fundamental way to help PostGIS development. The most effective bug report is that enabling PostGIS developers to reproduce it, so it would ideally contain a script triggering it and every information regarding the environment in which it was detected. Good enough info can be extracted running `SELECT postgis_full_version()` [for postgis] and `SELECT version()` [for postgresql].

If you aren't using the latest release, it's worth taking a look at its [release changelog](#) first, to find out if your bug has already been fixed.

Using the [PostGIS bug tracker](#) will ensure your reports are not discarded, and will keep you informed on its handling process. Before reporting a new bug please query the database to see if it is a known one, and if it is please add any new information you have about it.

You might want to read Simon Tatham's paper about [How to Report Bugs Effectively](#) before filing a new report.

7.2. Reporting Documentation Issues

The documentation should accurately reflect the features and behavior of the software. If it doesn't, it could be because of a software bug or because the documentation is in error or deficient.

Documentation issues can also be reported to the [PostGIS bug tracker](#).

If your revision is trivial, just describe it in a new bug tracker issue, being specific about its location in the documentation.

If your changes are more extensive, a Subversion patch is definitely preferred. This is a four step process on Unix (assuming you already have [Subversion](#) installed):

1. Check out a copy of PostGIS' Subversion trunk. On Unix, type:

```
svn checkout http://svn.refractorion.net/postgis/trunk/
```

This will be stored in the directory `./trunk`

2. Make your changes to the documentation with your favorite text editor. On Unix, type

(for example):

```
vi trunk/doc/postgis.xml
```

Note that the documentation is written in SGML rather than HTML, so if you are not familiar with it please follow the example of the rest of the documentation.

3. Make a patch file containing the differences from the master copy of the documentation. On Unix, type:

```
svn diff trunk/doc/postgis.xml > doc.patch
```

4. Attach the patch to a new issue in bug tracker.

Appendix A. Appendix

Table of Contents

A.1. Release Notes

- A.1.1. Release 1.3.3
- A.1.2. Release 1.3.2
- A.1.3. Release 1.3.1
- A.1.4. Release 1.3.0
- A.1.5. Release 1.2.1
- A.1.6. Release 1.2.0
- A.1.7. Release 1.1.6
- A.1.8. Release 1.1.5
- A.1.9. Release 1.1.4
- A.1.10. Release 1.1.3
- A.1.11. Release 1.1.2
- A.1.12. Release 1.1.1
- A.1.13. Release 1.1.0
- A.1.14. Release 1.0.6
- A.1.15. Release 1.0.5
- A.1.16. Release 1.0.4
- A.1.17. Release 1.0.3
- A.1.18. Release 1.0.2
- A.1.19. Release 1.0.1
- A.1.20. Release 1.0.0
- A.1.21. Release 1.0.0RC6
- A.1.22. Release 1.0.0RC5
- A.1.23. Release 1.0.0RC4
- A.1.24. Release 1.0.0RC3
- A.1.25. Release 1.0.0RC2
- A.1.26. Release 1.0.0RC1

A.1. Release Notes

A.1.1. Release 1.3.3

Release date: 2008/04/12

This release fixes bugs shp2pgsql, adds enhancements to SVG and KML support, adds a ST_SimplifyPreserveTopology function, makes the build more sensitive to GEOS versions, and fixes a handful of severe but rare failure cases.

A.1.2. Release 1.3.2

Release date: 2007/12/01

This release fixes bugs in ST_EndPoint() and ST_Envelope, improves support for JDBC building and OS/X, and adds better support for GML output with ST_AsGML(), including GML3 output.

A.1.3. Release 1.3.1

Release date: 2007/08/13

This release fixes some oversights in the previous release around version numbering, documentation, and tagging.

A.1.4. Release 1.3.0

Release date: 2007/08/09

This release provides performance enhancements to the relational functions, adds new relational functions and begins the migration of our function names to the SQL-MM convention, using the spatial type (SP) prefix.

A.1.4.1. Added Functionality

JDBC: Added Hibernate Dialect (thanks to Norman Barker)

Added ST_Covers and ST_CoveredBy relational functions. Description and justification of these functions can be found at <http://lin-ear-th-inking.blogspot.com/2007/06/subtleties-of-ogc-covers-spatial.html>

Added ST_DWithin relational function.

A.1.4.2. Performance Enhancements

Added cached and indexed point-in-polygon short-circuits for the functions ST_Contains, ST_Intersects, ST_Within and ST_Disjoint

Added inline index support for relational functions (except ST_Disjoint)

A.1.4.3. Other Changes

Extended curved geometry support into the geometry accessor and some processing functions

Began migration of functions to the SQL-MM naming convention; using a spatial type (ST) prefix.

Added initial support for PostgreSQL 8.3

A.1.5. Release 1.2.1

Release date: 2007/01/11

This release provides bug fixes in PostgreSQL 8.2 support and some small performance enhancements.

A.1.5.1. Changes

Fixed point-in-polygon shortcut bug in Within().

Fixed PostgreSQL 8.2 NULL handling for indexes.

Updated RPM spec files.

Added short-circuit for Transform() in no-op case.

JDBC: Fixed JTS handling for multi-dimensional geometries (thanks to Thomas Marti for hint and partial patch). Additionally, now JavaDoc is compiled and packaged. Fixed classpath problems with GCJ. Fixed pgjdbc 8.2 compatibility, losing support for jdk 1.3 and older.

A.1.6. Release 1.2.0

Release date: 2006/12/08

This release provides type definitions along with serialization/deserialization capabilities for SQL-MM defined curved geometries, as well as performance enhancements.

A.1.6.1. Changes

Added curved geometry type support for serialization/deserialization

Added point-in-polygon shortcircuit to the Contains and Within functions to improve performance for these cases.

A.1.7. Release 1.1.6

Release date: 2006/11/02

This is a bugfix release, in particular fixing a critical error with GEOS interface in 64bit

systems. Includes an updated of the SRS parameters and an improvement in reprojections (take Z in consideration). Upgrade is *encouraged*.

A.1.7.1. Upgrading

If you are upgrading from release 1.0.3 or later follow the [soft upgrade](#) procedure.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.7.2. Bug fixes

fixed CAPI change that broke 64-bit platforms

loader/dumper: fixed regression tests and usage output

Fixed setSRID() bug in JDBC, thanks to Thomas Marti

A.1.7.3. Other changes

use Z ordinate in reprojections

spatial_ref_sys.sql updated to EPSG 6.11.1

Simplified Version.config infrastructure to use a single pack of version variables for everything.

Include the Version.config in loader/dumper USAGE messages

Replace hand-made, fragile JDBC version parser with Properties

A.1.8. Release 1.1.5

Release date: 2006/10/13

This is an bugfix release, including a critical segfault on win32. Upgrade is *encouraged*.

A.1.8.1. Upgrading

If you are upgrading from release 1.0.3 or later follow the [soft upgrade](#) procedure.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.8.2. Bug fixes

Fixed MingW link error that was causing pgsqll2shp to segfault on Win32 when compiled for

PostgreSQL 8.2

fixed NullPointerException in Geometry.equals() method in Java

Added EJB3Spatial.odt to fulfill the GPL requirement of distributing the "preferred form of modification"

Removed obsolete synchronization from JDBC Jts code.

Updated heavily outdated README files for shp2pgsql/pgsql2shp by merging them with the manpages.

Fixed version tag in jdbc code that still said "1.1.3" in the "1.1.4" release.

A.1.8.3. New Features

Added -S option for non-multi geometries to shp2pgsql

A.1.9. Release 1.1.4

Release date: 2006/09/27

This is a bugfix release including some improvements in the Java interface. Upgrade is *encouraged*.

A.1.9.1. Upgrading

If you are upgrading from release 1.0.3 or later follow the [soft upgrade](#) procedure.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.9.2. Bug fixes

Fixed support for PostgreSQL 8.2

Fixed bug in collect() function discarding SRID of input

Added SRID match check in MakeBox2d and MakeBox3d

Fixed regress tests to pass with GEOS-3.0.0

Improved pgsql2shp run concurrency.

A.1.9.3. Java changes

reworked JTS support to reflect new upstream JTS developers' attitude to SRID handling. Simplifies code and drops build depend on GNU trove.

Added EJB2 support generously donated by the "Geodetix s.r.l. Company"
<http://www.geodetix.it/>

Added EJB3 tutorial / examples donated by Norman Barker <nbarker@ittvis.com>

Reorganized java directory layout a little.

A.1.10. Release 1.1.3

Release date: 2006/06/30

This is an bugfix release including also some new functionalities (most notably long transaction support) and portability enhancements. Upgrade is *encouraged*.

A.1.10.1. Upgrading

If you are upgrading from release 1.0.3 or later follow the [soft upgrade](#) procedure.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.10.2. Bug fixes / correctness

BUGFIX in distance(poly,poly) giving wrong results.

BUGFIX in pgsq2shp successful return code.

BUGFIX in shp2pgsql handling of MultiLine WKT.

BUGFIX in affine() failing to update bounding box.

WKT parser: forbidden construction of multigeometries with EMPTY elements (still supported for GEOMETRYCOLLECTION).

A.1.10.3. New functionalities

NEW Long Transactions support.

NEW DumpRings() function.

NEW AsHEXEWKB(geom, XDR|NDR) function.

A.1.10.4. JDBC changes

Improved regression tests: MultiPoint and scientific ordinates

Fixed some minor bugs in jdbc code

Added proper accessor functions for all fields in preparation of making those fields private

later

A.1.10.5. Other changes

NEW regress test support for loader/dumper.

Added `--with-proj-libdir` and `--with-geos-libdir` configure switches.

Support for build Tru64 build.

Use Jade for generating documentation.

Don't link `pgsql2shp` to more libs than required.

Initial support for PostgreSQL 8.2.

A.1.11. Release 1.1.2

Release date: 2006/03/30

This is a bugfix release including some new functions and portability enhancements. Upgrade is *encouraged*.

A.1.11.1. Upgrading

If you are upgrading from release 1.0.3 or later follow the [soft upgrade](#) procedure.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.11.2. Bug fixes

BUGFIX in `SnapToGrid()` computation of output bounding box

BUGFIX in `EnforceRHR()`

`jdbc2` SRID handling fixes in JTS code

Fixed support for 64bit archs

A.1.11.3. New functionalities

Regress tests can now be run **before** postgis installation

New `affine()` matrix transformation functions

New `rotate{X,Y,Z}()` function

Old translating and scaling functions now use `affine()` internally

Embedded access control in `estimated_extent()` for builds against `pgsql >= 8.0.0`

A.1.11.4. Other changes

More portable `./configure` script

Changed `./run_test` script to have more sane default behaviour

A.1.12. Release 1.1.1

Release date: 2006/01/23

This is an important Bugfix release, upgrade is *highly recommended*. Previous version contained a bug in `postgis_restore.pl` preventing [hard upgrade](#) procedure to complete and a bug in GEOS-2.2+ connector preventing GeometryCollection objects to be used in topological operations.

A.1.12.1. Upgrading

If you are upgrading from release 1.0.3 or later follow the [soft upgrade](#) procedure.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.12.2. Bug fixes

Fixed a premature exit in `postgis_restore.pl`

BUGFIX in geometrycollection handling of GEOS-CAPI connector

Solaris 2.7 and MingW support improvements

BUGFIX in `line_locate_point()`

Fixed handling of postgresql paths

BUGFIX in `line_substring()`

Added support for localized cluster in regress tester

A.1.12.3. New functionalities

New Z and M interpolation in `line_substring()`

New Z and M interpolation in `line_interpolate_point()`

added `NumInteriorRing()` alias due to OpenGIS ambiguity

A.1.13. Release 1.1.0

Release date: 2005/12/21

This is a Minor release, containing many improvements and new things. Most notably: build procedure greatly simplified; transform() performance drastically improved; more stable GEOS connectivity (CAPI support); lots of new functions; draft topology support.

It is *highly recommended* that you upgrade to GEOS-2.2.x before installing PostGIS, this will ensure future GEOS upgrades won't require a rebuild of the PostGIS library.

A.1.13.1. Credits

This release includes code from Mark Cave Ayland for caching of proj4 objects. Markus Schaber added many improvements in his JDBC2 code. Alex Bodnaru helped with PostgreSQL source dependency relief and provided Debian specfiles. Michael Fuhr tested new things on Solaris arch. David Techer and Gerald Fenoy helped testing GEOS C-API connector. Hartmut Tschauer provided code for the azimuth() function. Devrim GUNDUZ provided RPM specfiles. Carl Anderson helped with the new area building functions. See the [credits](#) section for more names.

A.1.13.2. Upgrading

If you are upgrading from release 1.0.3 or later you *DO NOT* need a dump/reload. Simply sourcing the new lwpostgis_upgrade.sql script in all your existing databases will work. See the [soft upgrade](#) chapter for more information.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.13.3. New functions

scale() and transscale() companion methods to translate()

line_substring()

line_locate_point()

M(point)

LineMerge(geometry)

shift_longitude(geometry)

azimuth(geometry)

locate_along_measure(geometry, float8)

locate_between_measures(geometry, float8, float8)

SnapToGrid by point offset (up to 4d support)

BuildArea(any_geometry)

OGC BdPolyFromText(linestring_wkt, srid)

OGC BdMPolyFromText(linestring_wkt, srid)

RemovePoint(linestring, offset)

ReplacePoint(linestring, offset, point)

A.1.13.4. Bug fixes

Fixed memory leak in polygonize()

Fixed bug in lwgeom_as_anytype cast functions

Fixed USE_GEOS, USE_PROJ and USE_STATS elements of postgis_version() output to always reflect library state.

A.1.13.5. Function semantic changes

SnapToGrid doesn't discard higher dimensions

Changed Z() function to return NULL if requested dimension is not available

A.1.13.6. Performance improvements

Much faster transform() function, caching proj4 objects

Removed automatic call to fix_geometry_columns() in AddGeometryColumns() and update_geometry_stats()

A.1.13.7. JDBC2 works

Makefile improvements

JTS support improvements

Improved regression test system

Basic consistency check method for geometry collections

Support for (Hex)(E)wkb

Autoprobing DriverWrapper for HexWKB / EWKT switching

fix compile problems in ValueSetter for ancient jdk releases.

fix EWKT constructors to accept SRID=4711; representation

added preliminary read-only support for java2d geometries

A.1.13.8. Other new things

Full autoconf-based configuration, with PostgreSQL source dependency relief

GEOS C-API support (2.2.0 and higher)

Initial support for topology modelling

Debian and RPM specfiles

New lwpostgis_upgrade.sql script

A.1.13.9. Other changes

JTS support improvements

Stricter mapping between DBF and SQL integer and string attributes

Wider and cleaner regression test suite

old jdbc code removed from release

obsoleted direct use of postgis_proc_upgrade.pl

scripts version unified with release version

A.1.14. Release 1.0.6

Release date: 2005/12/06

Contains a few bug fixes and improvements.

A.1.14.1. Upgrading

If you are upgrading from release 1.0.3 or later you *DO NOT* need a dump/reload.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.14.2. Bug fixes

Fixed palloc(0) call in collection deserializer (only gives problem with --enable-cassert)

Fixed bbox cache handling bugs

Fixed geom_accum(NULL, NULL) segfault

Fixed segfault in addPoint()

Fixed short-allocation in lwcollection_clone()

Fixed bug in segmentize()

Fixed bbox computation of SnapToGrid output

A.1.14.3. Improvements

Initial support for postgresql 8.2

Added missing SRID mismatch checks in GEOS ops

A.1.15. Release 1.0.5

Release date: 2005/11/25

Contains memory–alignment fixes in the library, a segfault fix in loader's handling of UTF8 attributes and a few improvements and cleanups.

Note

Return code of shp2pgsql changed from previous releases to conform to unix standards (return 0 on success).

A.1.15.1. Upgrading

If you are upgrading from release 1.0.3 or later you *DO NOT* need a dump/reload.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.15.2. Library changes

Fixed memory alignment problems

Fixed computation of null values fraction in analyzer

Fixed a small bug in the getPoint4d_p() low–level function

Speedup of serializer functions

Fixed a bug in force_3dm(), force_3dz() and force_4d()

A.1.15.3. Loader changes

Fixed return code of shp2pgsql

Fixed back-compatibility issue in loader (load of null shapefiles)

Fixed handling of trailing dots in dbf numerical attributes

Segfault fix in shp2pgsql (utf8 encoding)

A.1.15.4. Other changes

Schema aware postgis_proc_upgrade.pl, support for postgres 7.2+

New "Reporting Bugs" chapter in manual

A.1.16. Release 1.0.4

Release date: 2005/09/09

Contains important bug fixes and a few improvements. In particular, it fixes a memory leak preventing successful build of GiST indexes for large spatial tables.

A.1.16.1. Upgrading

If you are upgrading from release 1.0.3 you *DO NOT* need a dump/reload.

If you are upgrading from a release *between 1.0.0RC6 and 1.0.2* (inclusive) and really want a live upgrade read the [upgrade section](#) of the 1.0.3 release notes chapter.

Upgrade from any release prior to 1.0.0RC6 requires an [hard upgrade](#).

A.1.16.2. Bug fixes

Memory leak plugged in GiST indexing

Segfault fix in transform() handling of proj4 errors

Fixed some proj4 texts in spatial_ref_sys (missing +proj)

Loader: fixed string functions usage, reworked NULL objects check, fixed segfault on MULTILINESTRING input.

Fixed bug in MakeLine dimension handling

Fixed bug in translate() corrupting output bounding box

A.1.16.3. Improvements

Documentation improvements

More robust selectivity estimator

Minor speedup in distance()

Minor cleanups

GiST indexing cleanup

Looser syntax acceptance in box3d parser

A.1.17. Release 1.0.3

Release date: 2005/08/08

Contains some bug fixes – *including a severe one affecting correctness of stored geometries* – and a few improvements.

A.1.17.1. Upgrading

Due to a bug in a bounding box computation routine, the upgrade procedure requires special attention, as bounding boxes cached in the database could be incorrect.

An **hard upgrade** procedure (dump/reload) will force recomputation of all bounding boxes (not included in dumps). This is *required* if upgrading from releases prior to 1.0.0RC6.

If you are upgrading from versions 1.0.0RC6 or up, this release includes a perl script (utils/rebuild_bbox_caches.pl) to force recomputation of geometries' bounding boxes and invoke all operations required to propagate eventual changes in them (geometry statistics update, reindexing). Invoke the script after a make install (run with no args for syntax help). Optionally run utils/postgis_proc_upgrade.pl to refresh postgis procedures and functions signatures (see [Soft upgrade](#)).

A.1.17.2. Bug fixes

Severe bugfix in lwgeom's 2d bounding box computation

Bugfix in WKT (-w) POINT handling in loader

Bugfix in dumper on 64bit machines

Bugfix in dumper handling of user-defined queries

Bugfix in create_undef.pl script

A.1.17.3. Improvements

Small performance improvement in canonical input function

Minor cleanups in loader

Support for multibyte field names in loader

Improvement in the postgis_restore.pl script

New rebuild_bbox_caches.pl util script

A.1.18. Release 1.0.2

Release date: 2005/07/04

Contains a few bug fixes and improvements.

A.1.18.1. Upgrading

If you are upgrading from release 1.0.0RC6 or up you *DO NOT* need a dump/reload.

Upgrading from older releases requires a dump/reload. See the [upgrading](#) chapter for more informations.

A.1.18.2. Bug fixes

Fault tolerant btree ops

Memory leak plugged in pg_error

Rtree index fix

Cleaner build scripts (avoided mix of CFLAGS and CXXFLAGS)

A.1.18.3. Improvements

New index creation capabilities in loader (-I switch)

Initial support for postgresql 8.1dev

A.1.19. Release 1.0.1

Release date: 2005/05/24

Contains a few bug fixes and some improvements.

A.1.19.1. Upgrading

If you are upgrading from release 1.0.0RC6 or up you *DO NOT* need a dump/reload.

Upgrading from older releases requires a dump/reload. See the [upgrading](#) chapter for more informations.

A.1.19.2. Library changes

BUGFIX in 3d computation of length_spheroid()

BUGFIX in join selectivity estimator

A.1.19.3. Other changes/additions

BUGFIX in shp2pgsql escape functions

better support for concurrent postgis in multiple schemas

documentation fixes

jdbc2: compile with "--target 1.2 --source 1.2" by default

NEW -k switch for pgsq2shp

NEW support for custom createdb options in postgis_restore.pl

BUGFIX in pgsq2shp attribute names unicity enforcement

BUGFIX in Paris projections definitions

postgis_restore.pl cleanups

A.1.20. Release 1.0.0

Release date: 2005/04/19

Final 1.0.0 release. Contains a few bug fixes, some improvements in the loader (most notably support for older postgis versions), and more docs.

A.1.20.1. Upgrading

If you are upgrading from release 1.0.0RC6 you *DO NOT* need a dump/reload.

Upgrading from any other precedent release requires a dump/reload. See the [upgrading](#) chapter for more informations.

A.1.20.2. Library changes

BUGFIX in transform() releasing random memory address

BUGFIX in force_3dm() allocating less memory then required

BUGFIX in join selectivity estimator (defaults, leaks, tuplecount, sd)

A.1.20.3. Other changes/additions

BUGFIX in shp2pgsql escape of values starting with tab or single-quote

NEW manual pages for loader/dumper

NEW shp2pgsql support for old (HWGEOM) postgis versions

NEW -p (prepare) flag for shp2pgsql

NEW manual chapter about OGC compliancy enforcement

NEW autoconf support for JTS lib

BUGFIX in estimator testers (support for LWGEOM and schema parsing)

A.1.21. Release 1.0.0RC6

Release date: 2005/03/30

Sixth release candidate for 1.0.0. Contains a few bug fixes and cleanups.

A.1.21.1. Upgrading

You need a dump/reload to upgrade from precedent releases. See the [upgrading](#) chapter for more informations.

A.1.21.2. Library changes

BUGFIX in multi()

early return [when noop] from multi()

A.1.21.3. Scripts changes

dropped {x,y}{min,max}(box2d) functions

A.1.21.4. Other changes

BUGFIX in postgis_restore.pl scrip

BUGFIX in dumper's 64bit support

A.1.22. Release 1.0.0RC5

Release date: 2005/03/25

Fifth release candidate for 1.0.0. Contains a few bug fixes and a improvements.

A.1.22.1. Upgrading

If you are upgrading from release 1.0.0RC4 you *DO NOT* need a dump/reload.

Upgrading from any other precedent release requires a dump/reload. See the [upgrading](#) chapter for more informations.

A.1.22.2. Library changes

BUGFIX (segfaulting) in box3d computation (yes, another!).

BUGFIX (segfaulting) in estimated_extent().

A.1.22.3. Other changes

Small build scripts and utilities refinements.

Additional performance tips documented.

A.1.23. Release 1.0.0RC4

Release date: 2005/03/18

Fourth release candidate for 1.0.0. Contains bug fixes and a few improvements.

A.1.23.1. Upgrading

You need a dump/reload to upgrade from precedent releases. See the [upgrading](#) chapter for more informations.

A.1.23.2. Library changes

BUGFIX (segfaulting) in geom_accum().

BUGFIX in 64bit architectures support.

BUGFIX in box3d computation function with collections.

NEW subselects support in selectivity estimator.

Early return from force_collection.

Consistency check fix in SnapToGrid().

Box2d output changed back to 15 significant digits.

A.1.23.3. Scripts changes

NEW distance_sphere() function.

Changed get_proj4_from_srid implementation to use PL/PGSQL instead of SQL.

A.1.23.4. Other changes

BUGFIX in loader and dumper handling of MultiLine shapes

BUGFIX in loader, skipping all but first hole of polygons.

jdbc2: code cleanups, Makefile improvements

FLEX and YACC variables set `*after*` `pgsql` `Makefile.global` is included and only if the `pgsql` `*stripped*` version evaluates to the empty string

Added already generated parser in release

Build scripts refinements

improved version handling, central `Version.config`

improvements in `postgis_restore.pl`

A.1.24. Release 1.0.0RC3

Release date: 2005/02/24

Third release candidate for 1.0.0. Contains many bug fixes and improvements.

A.1.24.1. Upgrading

You need a `dump/reload` to upgrade from precedent releases. See the [upgrading](#) chapter for more informations.

A.1.24.2. Library changes

BUGFIX in `transform()`: missing SRID, better error handling.

BUGFIX in memory alignment handling

BUGFIX in `force_collection()` causing `mapserver` connector failures on simple (single) geometry types.

BUGFIX in `GeometryFromText()` missing to add a `bbox` cache.

reduced precision of `box2d` output.

prefixed `DEBUG` macros with `PGIS_` to avoid clash with `pgsql` one

plugged a leak in `GEOS2POSTGIS` converter

Reduced memory usage by early releasing `query-context` `palloced` one.

A.1.24.3. Scripts changes

BUGFIX in 72 index bindings.

BUGFIX in `probe_geometry_columns()` to work with `PG72` and support multiple geometry columns in a single table

NEW `bool::text` cast

Some functions made IMMUTABLE from STABLE, for performance improvement.

A.1.24.4. JDBC changes

jdbc2: small patches, box2d/3d tests, revised docs and license.

jdbc2: bug fix and testcase in for pgjdbc 8.0 type autoregistration

jdbc2: Removed use of jdk1.4 only features to enable build with older jdk releases.

jdbc2: Added support for building against pg72jdbc2.jar

jdbc2: updated and cleaned makefile

jdbc2: added BETA support for jts geometry classes

jdbc2: Skip known-to-fail tests against older PostGIS servers.

jdbc2: Fixed handling of measured geometries in EWKT.

A.1.24.5. Other changes

new performance tips chapter in manual

documentation updates: pgsq172 requirement, lwpostgis.sql

few changes in autoconf

BUILDDATE extraction made more portable

fixed spatial_ref_sys.sql to avoid vacuuming the whole database.

spatial_ref_sys: changed Paris entries to match the ones distributed with 0.x.

A.1.25. Release 1.0.0RC2

Release date: 2005/01/26

Second release candidate for 1.0.0 containing bug fixes and a few improvements.

A.1.25.1. Upgrading

You need a dump/reload to upgrade from precedent releases. See the [upgrading](#) chapter for more informations.

A.1.25.2. Library changes

BUGFIX in pointarray box3d computation

BUGFIX in distance_spheroid definition

BUGFIX in transform() missing to update bbox cache

NEW jdbc driver (jdbc2)

GEOMETRYCOLLECTION(EMPTY) syntax support for backward compatibility

Faster binary outputs

Stricter OGC WKB/WKT constructors

A.1.25.3. Scripts changes

More correct STABLE, IMMUTABLE, STRICT uses in lwpostgis.sql

stricter OGC WKB/WKT constructors

A.1.25.4. Other changes

Faster and more robust loader (both i18n and not)

Initial autoconf script

A.1.26. Release 1.0.0RC1

Release date: 2005/01/13

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

A.1.26.1. Upgrading

You need a dump/reload to upgrade from precedent releases. See the [upgrading](#) chapter for more informations.

A.1.26.2. 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.