MANUEL DE L'UTILISATEUR DE POSTGIS

Publié par

Paul Ramsey (Traduction Jean David TECHER)

MANUEL DE L'UTILISATEUR DE POSTGIS

Publié par Paul Ramsey (Traduction Jean David TECHER)

PostGIS est un module d'extension pour PostgreSQL. PostgreSQL est un SGBDR (Système de Gestion de Base de Données Relationnelles). PostGIS garantit l'adéquation des SIG (Système d'Information Géographique) par rapport à la base de données PotgreSQL, concernant notamment leur stockage, l'accès aux fonctions spatiales et l'analyse spatiale et l'analyse des SIG.

Table des matières

Chapitre 1. Introduction

PostGIS est développé et maintenue par l'équipe de Refractions Research. Situé à Victoria (Colombie britannique/Canada), Refractions Research est prestataire de service en géomatique, spécialisée dans l'intégration des données et le développement des projets libres orientés SIG/Web. Notre principal objectif est d'assurer non seulement le développement de PostGIS (notamment un support plus important pour les fonctions spatiales et l'analyse topologique définies par l'OpenGIs Consortium) mais aussi de confirmer son positionement dans les divers problématiques liés aux SIG: outils de développement pour l'utilisateur, , édition/visualisation de données SIG, outils d'applications Web.

1.1. L'Equipe de développement

Dave Blasby <dblasby@refractions.net>

Le développeur principal de PostGIS. Le support d'index spatial, la connexion avec MapServer, l'analyse coté serveur des fonctions et leurs implémentations objets sont ses principales problématiques.

Chris Hodgson <chodgson@refractions.net>

s'occupe des nouvelles fonctions et le support des index depuis la version 7.2

Paul Ramsey <pramsey@refractions.net>

le spécialiste Java. Il s'occupe des objets JBDC (Accès à PostGIS en Java), de la documentation, et du paquetage.

Jeff Lounsbury <jeffloun@refractions.net>

le spécialiste du couple Shapefiles/PostGIS. Il s'occupe de l'importation et de l'exportation de données Shapefiles/PostGIS

1.2. Où se procurer des informations?

- Les tous derniers développements, la documentation, les nouvelles rubriques sont accessibles sur le site Web de PosTGIS, http://postgis.refractions.net.
- pour la base de données PostgreSQL http://www.postgresql.org.
- pour le développement/support des index spatiaux (GiST) http://www.sai.msu.su/~megera/postgres/gist.
- pour MapServer http://mapserver.gis.umn.edu (http://mapserver.gis.umn.edu/).
- Les "spécifications en SQL (notamment schémas, entités,...) (http://www.opengis.org/techno/specs/99-049.pdf)" sont consultables sur le site de l'OpenGIS Consortium : http://www.opengis.org.

Chapitre 2. Installation

2.1. Recommandations et pré-requis

Les pré-requis suivants sont nécessaires pour un bon usage et l'installation de PostGIS:

- l'ensemble complet des sources de PostgreSQL à partir de laquelle vous constituerez votre propre distribution binaire de PostgreSQL. L'installation de PostGIS nécessite la configuration que vous aurez choisie pour installer votre distribution de PostgreSQL, de manière à rester aussi conforme à votre plateforme. On peut se procurer PostgreSQL à l'adresse suivante http://www.postgresql.org.
- un compilateur GNU C (`gcc`). D'autres compilateurs type ANSI C peuvent être utiliser pour compiler PostGIS. Cependant, des dificultés sont survenues concernant le compilateur `gcc`.
- GNU Make (`gmake` ou `make`). Sur la plupart des systèmes, GNU `make` est la version par défaut de `make`. Vérifiez votre version de `make` par la commande `make -v`. D'autres versions de `make` ne pourront pas accéder et permettre la compilation de PostGIS. Cette note concerne surtout le fichier `Makefile` de PostGIS.
- (OPTIONNEL) Proj4: la librairie de reprojection spatiale . La librairie Proj4 est utile pour permettre de reprojeter des données spatiales issues de divers systèmes de projections (Lambert II Etendu, Lambert 93,....) au sein de PostGIS - fonction filename>transform() the PostGIS - . Proj4 est accessible depuis http://www.remotesensing.org/proj.
- (OPTIONNEL) GEOS: la librairie d'accès aux fonctionnalités spatiales supplémentaires. PostGIS est par défaut livré avec les fonctions spatiales les plus répandues. Pour accéder à de plus amples fonctionnalités, il est intéressant d'installer Geos: fonctions IsValid(), Intersect(),.... Geos est accessible depuis http://geos.refractions.net.

2.2. PostGIS

Le module PostGIS est utilisé avec PostgreSQL. PostGIS 0.8.x *(ici x désigne un numéro de version)* a besoin des sources de PostgreSQL pour être compiler et installer. Les codes sources de PostgreSQL sont accessibles à http://www.postgresql.org.

PostGIS 0.8.x peut être compilé pour les versions de PostgreSQL allant de la 7.1.0 à 7.4.x. Les versions antérieures de PostgreSQL ne sont pas supportées pour la compilation de PostGIS..

1. Avant d'installer PostGIS, vous devez compiler et installer PostgreSQL.

   NOTE: Si vous souhaitez compiler PostGIS avec GEOS, il vous faudra lier PostgreSQL avec la librairie standard C++ en indiquant dans la ligne de configuration de PostgreSQL:

   ```
   LDFLAGS=-lstdc++ ./configure VOS_OPTIONS_DE_CONFIGURATION_DE_POSTGRESQL
   ```

   C'est dirons-nous un moyen de vous mettre en garde et à l'abricontre des bugs liés aux évènement d'exception en C++. Par exemple, un calcul d'intersection sur des couches spatiales non valides peuvent se traduirent parfois par un arrêt brutal du serveur PostgreSQL. Optez donc pour cette option de configuration.

2. Téléchargez l'archive de PostGIS à http://postgis.refractions.net/postgis-0.8.x.tar.gz. Décompressez l'archive dans le répertoire "contrib" des sources de PostgreSQL.

   ```
   # cd [repertoire des sources de postgresql]/contrib
   # gzip -d -c postgis-0.8.x.tar.gz | tar xvf -
   ```

3. Une fois que votre distribution de PostgreSQL est mise à jour, entrez dans le répertoire de "postgis" et éditez le fichier `Makefile`.

   - Si vous compilez PostGIS 0.7.2 pour les versions de PostgreSQL 7.2.x, vous devez mettre la variable `USE_PG72` à *1*. Ceci est automatiquement pris en compte pour les toutes dernières version de PostGIS.
   - Si vous voulez disposez des fonctionnalités de reprojection spatiale, assurez-vous d'avoir installé la librairie Proj4. Mettez à jour la variable `USE_PROJ` en la mettant à *1*, et éventuellement faites pointer la variable `PROJ_DIR` par rapport à votre répertoire d'installation de Proj4 (`/usr/local` par défaut).
   - Si vous voulez bénéficier des fonctionnalités spatiales supplémentaires qu'offre GEOS, assurez-vous d'avoir installer Geos. Mettez à jour la `USE_GEOS` à *1*, et éventuellement faites pointer la variable `GEOS_DIR` par rapport à votre répertoire d'installation de Geos (`/usr/local` par défaut).

4. Lancez la compilation et l'installation de PostGIS en faisant.

   ```
   # make
   # make install
   ```

   Tous les fichiers obtenues à la suite de l'installation sont installés de manière relative (chemins d'accès relatifs) par rapport au répertoire d'installation de PostgreSQL: `[prefix]`.

   - Les librairies sont installées dans `[prefix]/lib/contrib`.
   - Les fichiers SQL propres à postgis (`postgis.sql`,`spatial_ref_sys.sql`,...) sont localisés dans `[prefix]/share/contrib`.
   - les executables d'importation et d'exportation Shapefiles/PostGIS sont dans `[prefix]/bin`.

5. Pour pouvoir accéder aux fonctionnalités de PostGIS, toute base PostgreSQL doit être munie du langage procédural PL/pgSQL. Avant de charger le fichier `postgis.sql`, vous devez donc munir toute base de ce langage. Celà se fait en ayant recours à la commande `createlang`. Le guide de l'utilisateur PostgreSQL 7.1 contient tous les renseignements nécessaires pour un usage plus avancé de cette commande.

   ```
   # createlang plpgsql [votrebasededonnees]
   ```

6. Chargez maintenant les définitions d'objets géométriques , des fonctions spatiales dans votre base de données en chargeant le fichier de défintion `postgis.sql`.

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

   Les fonctionalités de PostGIS ont maintenant été chargées dans la base et prêtes à l'emploi.

7. Pour avoir aussi accès aux divers définitions de systèmes de référencement spatiaux, vous pouvez aussi charger le fichier de définition `spatial_ref_sys.sql`. Vous aurez ainsi rempli la table `SPATIAL_REF_SYS`.

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

2.2.1. Mise à jour

Les mises à jour de PostGIS d'une version à une autre ne sont pas sans risque. En effet, d'une version à une autre,la définition et l'implémentation des objets peuvent changer. Pour éviter tout problème conflictuel quand vous mettez à jour vos versions de PostGIS, le mieux à faire est de sauvegarder temporairement les tables contenues dans votre base (en SQL par exemple), on appelle cà un dump. Puis effacez votre ancienne base, crée une nouvelle, chargez le tout dernier fichier `postgis.sql` et réimporter votre sauvegarde:

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

> **Note :** En passant de la version 0.5 à 0.6+, toutes les géométries ont leur SRID à -1. Pour créer des géométries répondant aux spécifications de l'OpenGIS, vous devez créer un SRID valid dans la SPATIAL_REF_SYS, et mettre à jour vos géométries avec le SRID de référence en utilisant la requête SQL suivante (avec les substitutions appropriées):
>
> ```
> UPDATE TABLE <table> SET <geocolumn> = SetSRID(<geocolumn>,<SRID>);
> ```

2.2.2. Problèmes connus

Il y a plusieurs points à vérifier lorsque votre installation ou votre mise à jour ne se déroule pas comme voulu

1. Il est recommandé de décompresser l'archive de PostGIS dans le répertoire "contrib" des codes sources de PostgreSQL. En dépit de cette recommandation, si vous ne pouvez le faire vous pouvez spécifier la variable d'environnement `PGSQL_SRC`environment désignant votre répertoire des sources de PostgreSQL. Vous pourrez certes compiler PostGIS, mais il se peut que la commande `make install` ne fonctionne pas. En définitive, vous serez amené à copier manuellement les fichiers, les librairies et les executables de PostGIS dans les répertoires appropriés.
2. Vérifiez bien que le version de PostgreSQL que vous utilisez et qui par défaut est déjà peut-être installée sur votre système est la même que celle que vous avez compilé pour installer PostGIS. Vous serez peut-être amené à désinstallerla version déjà installée en cas de non compatibilité des versions de PostgreSQL. Par exemple sous Red Hat 9, PostgreSQL est installé par défaut pour 7.3 par RPM. Pour connaître la version de PostgreSQL que vous avez, connectez-vous à votre base de données en lançant `psql` et exécutez la requête suivante:

   ```
   SELECT version();
   ```

ou bien exemple sous Red Hat 9 par RPM pour tester si postgresql est déjà installé `rpm -qa postgresql`. Pour la désintaller, il suffira de faire `rpm -e -nodeps postgresql`.

Vérifier aussi que vous avez fait les changements nécessaisres au débutr du fichier `Makefile` de PostGIS. Celà suppose

1. Changer la variable `USE_PG72=0` à `USE_PG72=1` si vous utilisez une version de PostgreSQL supérieure ou égale à 7.2. Si cette variable est incorrecte, celà ce traduira par une redondance d'erreures lors de la compilation de PostGIS ou lors du chargement du fichier postgis.sql.
2. Si vous souhaitez bénéficier des fonctionnalités de reprojections spatiales, vous devez installer Proj4 sur votre système, Mettez la vairable `USE_PROJ` à 1 et la variable `PROJ_DIR` selon le répertoire d'installation de Proj4.
3. Si vous voulez bénéficier des fonctionnalités spatiales supplémentaires autrement dit installer Geos, mettez la variable `USE_GEOS` à 1 et la variable `GEOS_DIR` selon votre répertoire d'installation de Geos.

2.3. Exemple concret pour Red Hat 9

Cet exemple ici est fournit sans avoir recours aux RPM disponibles sur le site de PostGIS. Ici toutes les sources sont téléchargés vers le répertoire /usr/src/messources. Pour la version 0.8.2 de PostGIS, le `Makefile` est déjà configuré pour accepter Proj4 et Geos par défaut. Ici c'est PostgreSQL 7.4.2 qui est sera installé. Tout est installé avec la session de root de red Hat. L'initialisation de PostgreSQL a lieu pour l'utilisateur utilisaateurgis. En effet, pour éviter des problèmes de failles de sécurité, root ne peut lancer PostgreSQL.

```
mkdir /usr/src/messources

cd /usr/src/messources

wget ftp://ftp.remotesensing.org/pub/proj/proj-4.4.7.tar.gz

wget http://geos.refractions.net/geos-1.0.0.tar.bz2

wget ftp://ftp.fr.postgresql.org/source/v7.4.2/postgresql-7.4.2.tar.gz

wget http://postgis.refractions.net/postgis-0.8.1.tar.gz

tar xvzf proj-4.4.7.tar.gz

cd proj-4.4.7

./configure; make; make install

cd ..

tar xvjf geos-1.0.0.tar.bz2

cd geos-1.0

./configure; make; make install

cd ..

tar xvzf postgresql-7.4.2.tar.gz

cp postgis-0.8.2.tar.gz postgresql-7.4.2/contrib
```

```
cd postgresql-7.4.2/contrib

tar xvzf postgis-0.8.2.tar.gz

cd ..

rpm -e -nodeps postgresql

LDFLAGS=-lstdc++ ./configure -with-libs=/usr/local/lib -with-includes=/usr/local/include \

-enable-multibyte -with-CXX -prefix=/usr -sysconfdir=/etc -docdir=/usr/doc/postgresql-$version

make

cd contrib/postgis-0.8.2

make ; make install

perl -i.bak -pe 's/\$libdir/\/usr\/lib\/postgresql/g' postgis.sql

cd  ../..; make install

cd contrib/postgis-0.8.2

adduser utilisateurgis

su utilisateurgis

initdb -D /usr/src/data

pg_ctl  -D /usr/src/data start

createdb testgis

createlang plpgsql testgis

psql -d testgis -f postgis.sql

psql -d testgis -f spatial_ref_sys
```

2.4. JDBC

L'extension JDBC java de la distribution permet d'incorporer les fonctions internes ainsi que les objets de l'implémentation de PostGIS. Celà s'avère utile pour créer des interface-client en java dan sle but de faire des requêtes spatiales, interroger la base de données sous PostGIS...

1. Entrez dans le sous-répertoire `jdbc` de la distribution de PostGIS
2. Editez le `Makefile` précisez les chemins d'accès exactes pour le compilateur (`JAVAC`) and l'interpréteur (`JAVA`).
3. Lancez `make`. Copiez le fichier `postgis.jar` à l'endroit où vous stockez habituellement vos librairies javas.

2.5. Importateur/Exportateur de Shapefiles

Ces exécutables sont obtenus automatiquement lors de l'installation de PostGIS. Pour les installer manuellement:

```
# cd postgis-0.8.x/loader
# make
# make install
```

L'importateur s'apelle `shp2pgsql` and convertit les ESRI Shape files en fichier SQL qui peuvent ensuite être chargés dans PostGIS/PostgreSQL. L'exportateur - `pgsql2shp` - convertit les tables PostGIS en ESRI shape files.

Chapitre 3. Frequently Asked Questions

1. What kind of geometric objects can I store?

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

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

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

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

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

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

```
INSERT INTO gtest (ID, NAME, GEOM) VALUES (1, 'First Geometry', GeometryFromText('LINESTRING(2 3,4 5
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. How do I construct a spatial query?

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

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

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

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

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

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

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

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

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

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

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

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

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.

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.

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.

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

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

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

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

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

Chapitre 4. Using PostGIS

4.1. GIS Objects

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

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

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

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

4.1.1. Standard versus Canonical Forms

The OpenGIS specification defines two standard ways of expressing spatial objects: the Well-Known Text (WKT) form (shown in the previous section) and the Well-Known Binary (WKB) form. Both WKT and WKB include information about the type of the object and the coordinates which form the object.

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

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

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

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

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

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

4.2. Using OpenGIS Standards

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

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

4.2.1. The SPATIAL_REF_SYS Table

The SPATIAL_REF_SYS table definition is as follows:

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

The SPATIAL_REF_SYS columns are as follows:

SRID

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

AUTH_NAME

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

AUTH_SRID

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

SRTEXT

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

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

For a listing of EPSG projection codes and their corresponding WKT representations, see http://www.opengis.org/techno/interop/EPSG2WKT.TXT. For a discussion of WKT in general, see the OpenGIS "Coordinate Transformation Services Implementation Specification" at http://www.opengis.org/techno/specs.htm.

PROJ4TEXT

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

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

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

4.2.2. The GEOMETRY_COLUMNS Table

The `GEOMETRY_COLUMNS` table definition is as follows:

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

The columns are as follows:

F_TABLE_CATALOG, F_TABLE_SCHEMA, F_TABLE_NAME

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

F_GEOMETRY_COLUMN

The name of the geometry column in the feature table.

COORD_DIMENSION

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

SRID

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

TYPE

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

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

4.2.3. Creating a Spatial Table

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

- Create a normal non-spatial table.

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

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

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

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

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

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

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

4.3. Loading GIS Data

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

4.3.1. Using SQL

If you can convert your data to a text representation, then using formatted SQL might be the easiest way to get your data into PostGIS. As with Oracle and other SQL databases, data can be bulk loaded by piping a large text file full of SQL "INSERT" statements into the SQL terminal monitor.

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

```
BEGIN;
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (1,GeometryFromText('LINESTRING(191232 243118,191108 2
1),'Jeff Rd');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (2,GeometryFromText('LINESTRING(189141 244158,189265 2
1),'Geordie Rd');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (3,GeometryFromText('LINESTRING(192783 228138,192612 2
1),'Paul St');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (4,GeometryFromText('LINESTRING(189412 252431,189631 2
1),'Graeme Ave');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (5,GeometryFromText('LINESTRING(190131 224148,190871 2
1),'Phil Tce');
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (6,GeometryFromText('LINESTRING(198231 263418,198213 2
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.*

-D

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

-s <SRID>

Creates and populates the geometry tables with the specified SRID.

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

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

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

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

4.4. Retrieving GIS Data

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

4.4.1. Using SQL

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

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

```
(6 rows)
```

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

&&

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

~=

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

=

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

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

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

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

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

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

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

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

```
SELECT
  AsText(GEOM) AS GEOM
FROM ROADS_GEOM
WHERE
  GEOM && GeometryFromText('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 into a shape file. The basic syntax is:

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

The commandline options are:

-d

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

-f <filename>

Write the output to a particular filename.

-h <host>

The database host to connect to.

-p <port>

The port to connect to on the database host.

-P <password>

The password to use when connecting to the database.

-u <user>

The username to use when connecting to the database.

-g <geometry column>

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

4.5. Building Indexes

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

- B-Trees are used for data which can be sorted along one axis; for example, numbers, letters, dates. GIS data cannot be rationally sorted along one axis (which is greater, (0,0) or (0,1) or (1,0)?) so B-Tree indexing is of no use for us.
- R-Trees break up data into rectangles, and sub-rectangles, and sub-sub rectangles, etc. R-Trees are used by some spatial databases to index GIS data, but the PostgreSQL R-Tree implementation is not as robust as the GiST implementation.

- GiST (Generalized Search Trees) indexes break up data into "things to one side", "things which overlap", "things which are inside" and can be used on a wide range of data-types, including GIS data. PostGIS uses an R-Tree index implemented on top of GiST to index GIS data.

4.5.1. GiST Indexes

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

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

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

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

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

```
VACUUM ANALYZE;
```

GiST indexes have two advantages over R-Tree indexes in PostgreSQL. Firstly, GiST indexes are "null safe", meaning they can index columns which include null values. Secondly, GiST indexes support the concept of "lossiness" which is important when dealing with GIS objects larger than the PostgreSQL 8K page size. Lossiness allows PostgreSQL to store only the "important" part of an object in an index – in the case of GIS objects, just the bounding box. GIS objects larger than 8K will cause R-Tree indexes to fail in the 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 you run the "VACUUM ANALYZE [tablename]" command on the tables you are having problems with. "VACUUM ANALYZE" gathers statistics about the number and distributions of values in a table, to provide the query planner with better information to make decisions around index usage. You should regularly vacuum your databases anyways – many PostgreSQL DBAs have "VACUUM" run as an off-peak cron job on a regular basis.
- If vacuuming does not work, you can force the planner to use the index information by using the "SET 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".

4.6. Complex Queries

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

4.6.1. Taking Advantage of Indexes

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

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

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

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

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

4.7. 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] GIST_GEOMETRY_OPS );
```

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

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

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

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

4.7.2. 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 US-
ING 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.

4.7.3. Examples

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

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

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

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

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

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

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

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

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

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

```
FONT arial
END
END
END
```

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

4.8. Java Clients (JDBC)

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

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

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

      /*
      * Add the geometry types to the connection. Note that you
      * must cast the connection to the pgsql-specific connection * implementation before call-
ing 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

...

Chapitre 5. PostGIS Reference

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

5.1. OpenGIS Functions

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

Syntax: AddGeometryColumn(<schema_name>, <table_name>, <column_name>, <srid>, <type>, <dimension>). Adds a geometry column to an existing table of attributes. The `schema_name` is the name of the table schema (unused for pre-schema PostgreSQL installations). The `srid` must be an integer value reference to an entry in the SPATIAL_REF_SYS table. The `type` must be an uppercase string corresponding to the geometry type, eg, 'POLYGON' or 'MULTILINESTRING'.

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.

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')

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
```

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

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

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

equals(geometry)

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

Performed by the GEOS module

OGC SPEC s2.1.1.1

disjoint(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 //s2.1.13.3 - a.Relate(b, 'FF*FF****')

intersects(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 //s2.1.13.3 - Intersects(g1, g2) –> Not (Disjoint(g1, g2))

touches(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 // s2.1.13.3- a.Touches(b) -> (I(a) intersection I(b) = {empty set}) and (a intersection b) not empty

crosses(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 // s2.1.13.3 - a.Relate(b, 'T*T******')

within(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 // s2.1.13.3 - a.Relate(b, 'T*F**F***')

overlaps(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 // s2.1.13.3

contains(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 // s2.1.13.3 - same as within(geometry,geometry)

intersects(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1 // s2.1.13.3 - NOT disjoint(geometry,geometry)

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 s2.1.1.1 // s2.1.13.3

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

buffer(geometry,double)

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.

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.1

convexhull(geometry)

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

Performed by the GEOS module

OGC SPEC s2.1.1.1

intersection(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.1

GeomUnion(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

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

OGC SPEC s2.1.1.1

GeomUnion(geometry set)

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

Performed by the GEOS module

Do not call with a GeometryCollection in the argument set

Not explicitly defined in OGC SPEC

memGeomUnion(geometry set)

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

difference(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.1

difference(geometry,geometry)

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

Performed by the GEOS module

Do not call with a GeometryCollection as an argument

OGC SPEC s2.1.1.1

Envelope(geometry)

Returns a POLYGON representing the bounding box of the geometry.

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

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

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.

X(geometry)

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

Y(geometry)

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

Z(geometry)

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

NumPoints(geometry)

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

PointN(geometry,integer)

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

ExteriorRing(geometry)

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

NumInteriorRings(geometry)

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

InteriorRingN(geometry,integer)

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

IsClosed(geometry)

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

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

NumGeometries(geometry)

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

GeometryN(geometry,int)

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

1 is 1st geometry

Distance(geometry,geometry)

Return the cartesian distance between two geometries in projected units.

AsText(geometry)

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

OGC SPEC s2.1.1.1

SRID(geometry)

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

OGC SPEC s2.1.1.1

GeometryFromText(varchar, integer)

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

GeomFromText(varchar, integer)

As above. A synonym for GeometryFromText.

SetSRID(geometry)

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

EndPoint(geometry)

Returns the last point of the geometry as a point.

StartPoint(geometry)

Returns the first point of the geometry as a point.

Centroid(geometry)

Returns the centroid of the geometry as a point.

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

5.2. Other Functions

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 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 boux overlaps B’s bounding box the operator returns true.

area2d(geometry)

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

area(geometry)

Returns the area of the geometry if it is a polygon or multi-polygon. (same as area2(<polygon|multipolygon>)

asbinary(geometry,'NDR')

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

isvalid(geometry)

returns true if this geometry is valid.

asbinary(geometry,'XDR')

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

GeomFromText(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

GeometryFromText(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

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

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

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

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

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

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

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

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

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

GeomFromWKB(text,[<srid>])

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

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

GeomFromWKB(text,[<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

PointFromWKB(text,[<srid>])

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

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a POINT

LineFromWKB(text,[<srid>])

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

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a LINESTRING

LinestringFromWKB(text,[<srid>])

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

from the conformance suite

throws an error if WKB is not a LINESTRING

PolyFromWKB(text,[<srid>])

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

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a POLYGON

PolygonFromWKB(text,[<srid>])

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

from the conformance suite

throws an error if WKB is not a POLYGON

MPointFromWKB(text,[<srid>])

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

OGC SPEC 3.2.7.2 - option SRID is from the conformance suite

throws an error if WKB is not a MULTIPOINT

MLineFromWKB(text,[<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

MPolyFromWKB(text,[<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

GeomCollFromWKB(text,[<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

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 -

box3d(geometry)

Returns a BOX3D representing the maximum extents of the geometry.

collect(geometry set)

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

mem_collect(geometry set)

This does the the same of collect(geometry), only more memory-friendly (uses less memory and more processor time).

distance_spheroid(point, point, spheroid)

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

extent(geometry 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.

find_srid(varchar,varchar,varchar)

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

force_collection(geometry)

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

force_2d(geometry)

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

force_3d(geometry)

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

length2d(geometry)

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

length(geometry)

The length of this Curve in its associated spatial reference.

synonym for length2d()

OGC SPEC 2.1.5.1

length3d(geometry)

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

length_spheroid(geometry,spheroid)

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

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

Eg:

SPHEROID["GRS_1980",6378137,298.257222101]

An example calculation might look like this:

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

length3d_spheroid(geometry,spheroid)

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

max_distance(linestring,linestring)

Returns the largest distance between two line strings.

mem_size(geometry)

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

npoints(geometry)

Returns the number of points in the geometry.

nrings(geometry)

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

numb_sub_objects(geometry)

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

perimeter2d(geometry)

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

perimeter3d(geometry)

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

point_inside_circle(geometry,float,float,float)

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

postgis_version()

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

summary(geometry)

Returns a text summary of the contents of the geometry.

transform(geometry,integer)

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

translate(geometry,float8,float8,float8)

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

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

Returns the requested minima of a bounding box.

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

Returns the requested maxima of a bounding box.

simplify(geometry, tolerance)

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

line_interpolate_point(geometry, distance)

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