pgSphere 1.1.5

pgSphere development team

pgSphere provides spherical data types, functions, and operators for PostgreSQL.

The project is hosted at *pgfoundry.org* (http://pgfoundry.org/projects/pgsphere/) and *https://github.com/akorotkov/pgsphere*

This document describes installation and usage of this module.

Table of Contents

1. What is pgSphere?

pgSphere is an extra module for PostgreSQL which adds spherical data types. It provides:

- input and output of data
- containing, overlapping, and other operators
- various input and converting functions and operators
- circumference and area of an object
- spherical transformation
- indexing of spherical data types
- several input and output formats

Hence, you can do a fast search and analysis for objects with spherical attributes as used in geographical, astronomical, or other applications using PostgreSQL. For instance, you can manage data of geographical objects around the world and astronomical data like star and other catalogs conveniently using an `SQL` interface.

The aim of pgSphere is to provide uniform access to spherical data. Because PostgreSQL itself supports a lot of software interfaces, you can now use the same database with different utilities and applications.

2. Installation

2.1. Download

pgSphere is not part of the PostgreSQL software. You can download it from the pgSphere homepage *https://github.com/akorotkov/pgsphere*

2.2. Installation

You will need PostgreSQL 9.1 or above. We assume that you have PostgreSQL already compiled and installed. Please note: Depending on your system configuration mostly you have to be logged in as the system superuser.

There are two ways to compile pgSphere. The first is to copy the sources into the contribution directory of PostgreSQL's source tree (`POSTGRESQL_SRC/src/contrib`). Then, change into `POSTGRESQL_SRC/src/contrib`. If the sources are not yet installed and the directory `pg_sphere` does not exist, take the gzipped pgSphere sources (e. g., `pg_sphere_xxx.tgz`) and run:

```
shell> tar -xzf path/to/pg_sphere_xxx.tgz
```

Now, change into the `pg_sphere` directory and run :

```
shell> make
```

and to install pgSphere :

```
shell> make install
```

The second way does not require the PostgreSQL sources but the configuration tool pg_config.
First unpack the pgSphere sources:

```
shell> tar -xzf path_to_pg_sphere_xxx.tgz
```

Now, change into the `pg_sphere` directory and run:

```
shell> make USE_PGXS=1 PG_CONFIG=/path/to/pg_config
```

To install pgSphere you have to run :

```
shell> make USE_PGXS=1 PG_CONFIG=/path/to/pg_config install
```

To check the installation change into the `pg_sphere` source directory again and run:

```
shell> make installcheck
```

The check status will be displayed. Please note, the check gives different results with different PostgreSQL-versions. Currently, the check should run without errors with PostgreSQL-version 8.4. Otherwise check the file `regression.diff`.

2.3. Creating a database with pgSphere

We assume you have already created a database datab, where datab is the name of any database. Presupposing the name of your PostgreSQL's superuser is *`postgres`*, type:

```
shell> psql -U postgres -c 'CREATE EXTENSION pg_sphere;' datab
```

Depending on your system, it may be necessary to give more **psql** options like port or host name. Please have a look at the PostgreSQL documentation for more details.

To get the version of installed pgSphere software, simply call:

```
pgsql> SELECT pg_sphere_version();
```

3. Data types

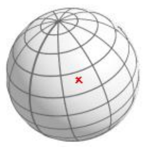
3.1. Overview

pgSphere provides spherical data types for storing with PostgreSQL. Furthermore, there is a data type to do transformations.

Table 1. Data types

SQL type name	spherical type
spoint	point (position)
strans	Euler transformation
scircle	circle
sline	line
sellipse	ellipse
spoly	polygon
spath	path
sbox	coordinate range

3.2. Point



A spherical point is an object without expanse but with a position. Use cases are:

- sites on earth
- star positions on the sky sphere
- spherical positions on planets

A spherical point (or position) is given by two values: longitude and latitude. Longitude is a floating point value between `0` and 2π. Latitude is a floating point value, too, but between $-\pi/2$ and $\pi/2$. It is possible to give a spherical position in degrees (`DEG`) or with a triple value of degrees, minutes and seconds (`DMS`). Degrees and minutes are integer values. The seconds are represented using a floating point value. A fourth method is specifying a longitude value as a triple value of hours, minutes and seconds (`HMS`). But, you can not use it with latitude values.

Example 1. A position specified using longitude and latitude in radians

```
sql> SELECT spoint '(0.1,-0.2)';
```

Example 2. A position specified using longitude and latitude in degrees

```
sql> SELECT spoint '( 10.1d, -90d)';
```

Example 3. A position specified using longitude and latitude (DMS)

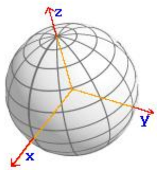
```
sql> SELECT spoint '( 10d 12m 11.3s, -13d 14m)';
```

Example 4. A position specified using longitude in `HMS`, and latitude in `RAD`

```
sql> SELECT spoint '( 23h 44m 10s, -1.4321 )';
```

As you can see you can combine the input format for longitude and latitude. The value pairs are always enclosed within braces. Spaces are optional.

3.3. Euler transformation



An Euler transformation is done with three counterclockwise object rotations around following the axes: `x`-axis, `y`-axis, or `z`-axis. Use cases are:

- spherical object transformations
- spherical coordinates transformations

The input syntax of an Euler transformation is:

`angle1, angle2, angle3 [, axes ]`

where *`axes`* is an optional 3 letter code with letters : `X`, `Y`, or `Z`. Default is `ZXZ`. *`angleN`* is any valid angle with the input format `RAD`, `DEG`, or `DMS`.

To do a transformation, you have to use a transformation operator (see Section 5.10).

Example 5. Create a transformation object

Create a transformation object to rotate a spherical object counterclockwise, first 20° around the `x`-axis, second -270° around the `z`-axis and last 70.5° around the `y`-axis.

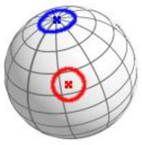
```
sql> SELECT strans '20d, -270d, 70.5d, XZY';
```

Example 6. Create a second transformation object

Create a transformation object to rotate a spherical object counterclockwise, first 2° 20’ around the `z`-axis, second 10° around the `x`-axis, and last 0° around the `z`-axis.

```
sql> SELECT strans '2d 20m, 10d, 0';
```

3.4. Circle



A spherical circle is an area around a point, where all points inside the circle have a distance less than or equal to the radius of the circle. Use cases are:

- sites on earth having a maximum distance from another site
- round cluster or nebula on sky sphere
- a position with an undirected position error

A circle is specified using a spherical point (spoint) and a radius :

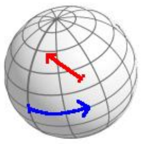
`< point , radius >`

Valid radius units are `RAD`, `DEG`, and `DMS`. The circle radius must be less than or equal to 90° and cannot be less than zero.

Example 7. A circle around the North Pole with a radius of 5°.

```
sql> SELECT scircle '< (0d, 90d), 5d >';
```

3.5. Line



A spherical line is part of a great circle (meridian) that has a beginning and an end and hence, a direction. Use cases are:

- direct connection of two points
- meteors on the sky sphere

To allow lines with a length larger than 180°, the input i syntax is a somewhat complex.

A general located line with a length *`length`* is defined as a line starting at position `(0d,0d)` and ending at position `(length,0d)` transformed with an Euler transformation *`euler`*. The input syntax is :

`( euler ), length`

Note:

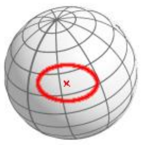
- For a simpler line input, use casting operators (Section 5.1) or constructor functions (Section 4).
- If the length is larger than 360°, the line length is truncated to 360°.
- The transformation *`euler`* will always be converted to an Euler transformation using axes `Z`, `X`, and `Z`.

Example 8. A line input

A line starting at position `(200d,+20d)` and ending at position `(200d,-10d)`.

```
sql> SELECT sline '( -90d, -20d, 200d, XYZ ), 30d ';
```

3.6. Ellipses



Within pgSphere, ellipses are defined as :

> *If the center of any spherical ellipse is the North Pole, the perpendicular projection into the `x`-`y`-plane gives an ellipse as in two-dimensional space.*

Use cases are:

- cluster or nebula on the sky sphere where the 'height' is lower than the 'width'
- to describe a position error

An ellipse always has:

- a major radius *`rad_1`*
- a minor radius *`rad_2`*
- a center *`center`*
- a position angle *`pos`* (inclination)

Hence, the input syntax is:

```
< { rad_1, rad_2 }, center, pos >
```

Note:

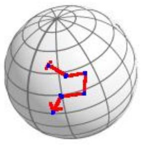
- The radii *`rad_1`* and *`rad_2`* have to be less than 90°.
- If *`rad_1`* is less than *`rad_2`* , the values will be swapped.
- The position angle *`pos`* is defined within pgSphere as a counterclockwise rotation around the ellipse center and is zero, if the ellipse is "parallel to the equator"

Example 9. Input of a spherical ellipse

An ellipse has a center at 20° longitude and 0° latitude. The minor radius is part of the equator. The major radius is 10°, the minor radius is 5°.

```
sql> SELECT sellipse '< { 10d, 5d } , ( 20d, 0d ), 90d >';
```

3.7. Path



A spherical path is a concatenation of spherical lines. Use cases are:

- rivers on earth
- trajectories of planets or comets on the sky plane

Paths within pgSphere are simplified lists of positions. The input syntax is :
`{pos1,pos2[,pos3[,pos4[,...]]]}`

Note:

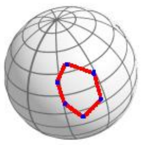
- The distance between 2 sequent positions has to be less than 180° and greater than zero.
- At least 2 positions are required.

Example 10. Path input example

A path going from `(10d,0d)` to `(80d,30d)` via `(45d,15d)`.

```
sql> SELECT spath '{ (10d,0d),(45d,15d),(80d,30d) } ';
```

3.8. Polygon



A spherical polygon is a closed spherical path where line segments cannot be crossed. One main use case are areas on the earth and sky sphere. Polygons within pgSphere have the same input syntax as paths:
`{pos1,pos2,pos3[,pos4[,... ]]}`

Note:

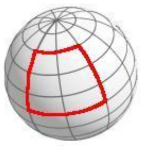
- A spherical polygon has the same restrictions as a spherical path (see Section 3.7). Except that a polygon needs at least 3 positions.
- The line segments can not be crossed.
- The maximum dimension of a polygon must be less than 180°.

Example 11. Input of polygon

A polygon going from `(270d,-10d)`. via `(270d,30d)` to `(290d,10d)` back to `(270d,-10d)`

```
sql> SELECT spoly '{ (270d,-10d), (270d,30d), (290d,10d) } ';
```

3.9. Coordinates range



A spherical box is a coordinates range. Hence, you can select objects within a longitude range and latitude range. The box is represented using two spherical points: the southwest (`pos_sw`) and the northeast (`pos_ne`) edge of the box. The input syntax is:

```
( pos_sw, pos_ne )
```

or

```
pos_sw, pos_ne
```

Note:

- If the latitude of the southwest edge is larger than the latitude of the northeast edge, pgSphere swaps the edges.
- If the longitude of the southwest edge is equal to the longitude of the northeast edge, pgSphere assumes a full latitude range, except that the latitudes are equal, too.

Example 12. Input of a full latitude range

A full latitude range between +20° and +23°.

```
sql> SELECT sbox '( (0d,20d), (0d,23d) )';
```

Example 13. A simple coordinates range

A coordinate range between -10° and +10° in latitude and 350° and 10° in longitude.

```
sql> SELECT sbox '( (350d,-10d), (10d,+10d) )';
```

4. Constructors

Constructors within pgSphere are functions needed to create spherical data types from other data types. For actual use, there isn't a difference to *usual* PostgreSQL functions. pgSphere constructor functions are named by returned data type. Constructor functions expecting one parameter only are implemented as casting operators, too. These are not mentioned here.

4.1. Point

There is only one constructor function for spherical points.

```
spoint(float8 lng, float8 lat);
```

where *lng* is the longitude of the spherical point in radians, *lng* is the latitude of the spherical point in radians.

Example 14. A spherical point from longitude and latitude

Get a spherical position with 270° of longitude and -30° of latitude.

```
sql> SELECT spoint ( 270.0*pi()/180.0,-30.0*pi()/180.0 ) AS spoint;
```

4.2. Euler transformation

There are two constructor functions for an Euler transformation:

```
strans(float8 phi, float8 theta, float8 psi);
strans(float8 phi, float8 theta, float8 psi, character axis);
```

where *phi*, *theta* and *psi* are the three angles of Euler transformation. The fourth parameter is the three letter code of Euler the transformation axis. If that parameter is omitted, pgSphere will assume `ZXZ`. For more information about that parameter, see Section 3.3.

Example 15. Create an Euler transformation object

Create a transformation object to rotate a spherical object counterclockwise, first 20° around the `x`-axis, second -270° around the `z`-axis, and last 70.5° around the `y`-axis.

```
sql> SELECT strans ( 20.0*pi()/180.0, -270.0*pi()/180.0, 70.5*pi()/180.0, 'XZY');
```

4.3. Circle

The function

```
scircle(spoint center, float8 radius);
```

returns a spherical circle with center at *center* and a radius *radius* in radians. The circle radius has to be larger than or equal to zero but less or equal to 90°. Otherwise, this function returns an error.

Example 16. A circle around the north pole

Get a spherical circle around the North Pole with a radius of 30°.

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)
```

```
sql> SELECT scircle ( spoint '(0d,90d)', 30.0*pi()/180.0 );
      scircle
--------------------
 <(0d , 90d) , 30d>
(1 row)
```

4.4. Line

The input of spherical lines using Euler transformation and length is quite circumstantial (see Section 3.5). For short lines it is easier to input a line specifying the beginning and the end of the line.

`sline``(spoint` *`begin`*`, spoint` *`end`*`);`

If the distance between *`begin`* and *`end`* is 180° (π), this function returns an error because the location of the line is undefined. However, if longitudes of *`begin`* and *`end`* are equal, pgSphere assumes a meridian and returns the corresponding spherical line.

Example 17. A line created using begin and end of line

A line starting at `spoint '(270d,10d)'` and ending at `spoint '(270d,30d)'`:

```
sql> SELECT set_sphere_output('DEG')
 set_sphere_output
-------------------
  SET DEG
 (1 row)

sql> SELECT sline( spoint '(270d,10d)', spoint '(270d,30d)');
          sline
 ----------------------------
  ( 10d, 90d, 270d, ZXZ ), 20d
 (1 row)
```

Furthermore, there is a function for inputing a line using Euler transformation *`trans`* and line length *`length`*

`sline``(strans` *`trans`*`, float8` *`length`*`);`

where the line length *`length`* must be given in radians.

Example 18. A line created with its transformation and length

The same line as in Example 17, but using transformation and line length.

```
sql> SELECT sline ( strans '10d, 90d, 270d, ZXZ', 20.0*pi()/180.0 );
          sline
------------------------------
 ( 10d, 90d, 270d, ZXZ ), 20d
(1 row)
```

4.5. Ellipse

You can use the function

`sellipse``(spoint` *`center`*`, float8` *`major_rad`*`, float8` *`minor_rad`*`, float8` *`incl`*`);`

to create a spherical ellipse. The first parameter *`center`* is the center of ellipse. The parameter *`major_rad`* and *`minor_rad`* are the major and the minor radii of the ellipse in radians. If the major radius is smaller than minor radius, pgSphere swaps the values automatically. The last parameter *`incl`* is the inclination angle in radians. For more informations about ellipses, see Section 3.6.

Example 19. Create an ellipse

An ellipse with a center at 20° of longitude and 0° of latitude. The minor radius is part of the equator. The major radius has a size of 10°. The minor radius has 5°.

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)
sql> SELECT sellipse ( spoint '( 20d, 0d )', 10.0*pi()/180.0, 5.0*pi()/180.0,
       pi()/2.0 );
             sellipse
-----------------------------------
 <{ 10d , 5d }, (20d , -0d) , 90d>
(1 row)
```

4.6. Polygon

The aggregate function

  `spoly``(spoint` *`edge`*`);`

can be used to create a polygon from a set of spherical points. There are the same restrictions as for using the input function of spherical polygon (see Section 3.8). The function returns `NULL`, if the polygon couldn't be created.

Example 20. Create a spherical polygon using a set of spherical points

Create a table and put in some spherical points with a unique ID. Then, create two polygons with different edge sequences.

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)

sql> CREATE TABLE points ( i int PRIMARY KEY, p spoint );
sql> INSERT INTO points VALUES (1, '( 0d, 0d)');
sql> INSERT INTO points VALUES (2, '(10d, 0d)');
sql> INSERT INTO points VALUES (3, '( 0d,10d)');
sql> SELECT spoly(data.p) FROM ( SELECT p FROM points ORDER BY i ASC ) AS data ;
               spoly
-----------------------------------
 {(0d , 0d),(10d , 0d),(0d , 10d)}
(1 row)

sql> SELECT spoly(data.p) FROM ( SELECT p FROM points ORDER BY i DESC ) AS data ;
               spoly
-----------------------------------
 {(0d , 10d),(10d , 0d),(0d , 0d)}
(1 row)
```

4.7. Path

Similar to spherical polygons, you can use the aggregate function

```
spath(spoint edge);
```

to create a spherical path using a set of spherical points. There are the same restrictions as with the input function of spherical path (see Section 3.7). The function returns `NULL` if the path couldn't be created.

Example 21. Create a spherical path using a set of spherical points

Create a table and put in some spherical points with a unique ID. Then, create a spherical path from it.

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)

sql> CREATE TABLE points ( i int PRIMARY KEY, p spoint );
sql> INSERT INTO points VALUES (1, '( 0d, 10d)');
sql> INSERT INTO points VALUES (2, '( 0d,  0d)');
sql> INSERT INTO points VALUES (3, '( 0d,-10d)');
sql> SELECT spath(data.p) FROM ( SELECT p FROM points ORDER BY i ASC ) AS data ;
                       spath
---------------------------------------------------
 {(0d , 10d),(0d , 0d),(0d , -10d)}
(1 row)
sql> SELECT spath(data.p) FROM ( SELECT p FROM points ORDER BY i DESC ) AS data ;
                       spath
---------------------------------------------------
 {(0d , -10d),(0d , 0d),(0d , 10d)}
(1 row)
```

4.8. Coordinates range

The function

```
sbox(spoint south_west, spoint north_east);
```

creates an sbox object with its first parameter *`south_west`* as the southwest edge and its second parameter *`northeast`* as the north-east edge of the coordinates range.

Example 22. Create a spherical box using edges

A coordinate range between 0° and +10° in latitude and longitude.

```
sql> SELECT sbox ( spoint '(0d,0d),(10d,10d)' );
```

5. Operators

5.1. Casting

pgSphere provides some casting operators. So, you can transform an object to another data type. A cast is done using a `CAST(x AS typename)`, `x::typename` or `typename(x)` construct.

Table 2. Castings

casting argument	type target	returns
spoint	scircle	circle with center position spoint and radius 0.0
spoint	sellipse	an ellipse at position spoint and radius 0.0
spoint	sline	a line with length 0.0 at position spoint
scircle	sellipse	the scircle as sellipse
sline	strans	the Euler transformation of sline
sellipse	scircle	the bounding circle of sellipse
sellipse	strans	the Euler transformation of sellipse

Example 23. Cast a spherical point as a circle

```
sql> SELECT CAST ( spoint '(10d,20d)' AS scircle );
      scircle
--------------------
 <(10d , 20d) , 0d>
(1 row)
```

5.2. Equality

All data types of pgSphere have equality operators. The equality operator is as in `SQL` =. Furthermore, there are two valid negators to indicate that two objects are not equal: `!=` and `<>`.

Example 24. Equality of two spherical points

```
sql> SELECT spoint '(10d,20d)' = spoint '(370d,20d)' ;
 test
------
 t
(1 row)
```

5.3. Contain and overlap

On the sphere, an equality relationship is rarely used. There are frequently questions like *Is object `a` contained by object `b`?* or *Does object `a` overlap object `b`?* pgSphere supports such queries using binary operators returning *true* or *false*:

Table 3. Contain and overlap operators

operator	operator returns true, if
@ or <@	the left object is contained by the right object
~ or @>	the left object contains the right object
!@ or !<@	the left object is not contained by the right object
!~ or !@>	the left object does not contain the right object
&&	the objects overlap each other
!&&	the objects do not overlap each other

An overlap or contain operator does not exist for all combinations of data types. For instance, scircle @ spoint is useless because a spherical point can never contain a spherical circle.

Example 25. Is the left circle contained by the right circle?

```
sql> SELECT scircle '<(0d,20d),2d>' @ scircle '<(355d,20d),10d>' AS test ;
test
------
 t
(1 row)
```

Example 26. Are the circles overlapping?

```
sql> SELECT scircle '<(0d,20d),2d>' && scircle '<(199d,-10d),10d>' AS test ;
 test
------
 f
(1 row)
```

5.4. Crossing of lines

Another binary relationship is *crossing*. pgSphere supports only crossing of lines. The correlative operator is named `#`.

Example 27. Are the lines crossed?

```
sql> SELECT sline '(0d,0d,0d),10d' # sline '(90d,5d,5d,XYZ),10d' AS test ;
 test
------
 t
(1 row)
```

5.5. Distance

The binary distance operator <-> is a non-boolean operator returning the distance between two objects in radians. Currently, pgSphere supports only distances between points, circles, and between point and circle. If the objects are overlapping, the distance operator returns zero (0.0).

Example 28. Distance between two circles

```
sql> SELECT 180 * ( scircle '<(0d,20d),2d>' <-> scircle '<(0d,40d),2d>' )
        / pi() AS dist ;
 dist
------
 16
(1 row)
```

5.6. Length and circumference

The length/circumference operator `@-@` is a non-boolean unary operator returning the circumference or length of an object. In the current implementation, pgSphere supports only circumferences of circles, polygons, and boxes. It supports lengths of lines and paths too. Instead of using the operator, you can use the functions `circum(object)` or `length(object)`.

Example 29. Circumference of a circle

```
sql> SELECT 180 * ( @-@ scircle '<(0d,20d),30d>' )/ pi() AS circ ;
 circ
------
 180
(1 row)
```

Example 30. Length of a line

```
sql> SELECT 180 * ( @-@ sline '(0d,0d,0d),30d' )/ pi() AS length ;
 length
--------
 30
(1 row)
```

5.7. Center

The center operator `@@` is a non-boolean unary operator returning the center of an object. In the current implementation of pgSphere, only centers of circles and ellipses are supported. Instead of using the operator, you can use the function `center(object)`.

Example 31. Center of a circle

```
sql> SELECT @@ scircle '<(0d,20d),30d>';
```

5.8. Change the direction

The unary operator `-` changes the direction of sline or spath objects. You can use it with an Euler transformation object in the figurative sense, too (Section 5.10).

Example 32. Swap begin and end of a sline

```
sql> SELECT - sline (spoint '(0d,0d)', spoint '(10d,0d)');
```

5.9. Turn the path of a line

The unary operator `!` turns the path of sline objects, but preserves begin and end of the spherical line. The length of returned line will be 360° minus the line length of operator's argument.

The operator `!` returns `NULL`, if the length of sline argument is 0, because the path of returned sline is undefined.

Example 33. Return length and check if north pole on slines

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)

sql> SELECT length ( sline ( spoint '(0d,0d)', spoint '(0d,10d)' ) ) *
        180.0 / pi() AS length;
 length
--------
     10
(1 row)

sql> SELECT spoint '(0d,90d)' @
        sline ( spoint '(0d,0d)', spoint '(0d,10d)' ) AS test;
 test
------
 f
(1 row)

sql> SELECT length ( ! sline ( spoint '(0d,0d)', spoint '(0d,10d)' ) ) *
        180.0 / pi() AS length;
 length
--------
    350
(1 row)

sql> SELECT spoint '(0d,90d)' @
        ! sline ( spoint '(0d,0d)', spoint '(0d,10d)' ) AS test;
 test
------
 t
(1 row)
```

5.10. Transformation

As in a plane, translations and rotations are needed to do object or coordinate transformations. With pgSphere, it is done using Euler transformations (strans). On a sphere, there aren't real translations. All movements on a sphere are rotations around axes.

The general syntax for a transformation is always:

`object operator euler`

where operators are `+` for a usual transformation, `-` for an inverse transformation. You can transform any object having a pgSphere data type, except the data type sbox.

Example 34. Transformation of a point

Rotate a spherical point counterclockwise, first 90° around the `x`-axis, second 90° around the `z`-axis, and last 40.5° around the `x`-axis.

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)

sql> SELECT spoint '(30d,0d)' + strans '90d, 90d, 40.5d, XZX AS spoint';
   spoint
---------------
 (90d , 70.5d)
(1 row)
```

You can use the `+` and `-` operator as unary operators for transformations, too. `+strans` just returns the transformation itself, `-strans` returns the inverse transformation.

Example 35. An inverse transformation

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)

sql> SELECT - strans '20d, 50d, 80d, XYZ' AS inverted;
      inverted
-----------------------
 280d, 310d, 340d, ZYX
(1 row)
```

6. Functions

The functions described below are implemented without having an operator. If you are missing some functions, see Section 5 and use the operators.

6.1. Area function

The `area` function returns the area of a spherical object in square radians. Supported data types are: scircle, spolygon (if the polygon is convex), and sbox.

Example 36. Area of a spherical circle as a multiple of π

```
sql> SELECT area( scircle '<(0d,90d),60d>' ) / pi() AS area;
 area
------
 1
(1 row)
```

6.2. spoint functions

6.2.1. Longitude and latitude

The functions

```
  long(spoint p);
  lat(spoint p);
```

returns the longitude or latitude value of a spherical position *p* in radians.

Example 37. Get the longitude and latitude of a spherical point in degrees

```
sql> SELECT long ( spoint '(10d,20d)' ) * 180.0 / pi() AS longitude;
 longitude
------------
 10
(1 row)

sql> SELECT lat ( spoint '(10d,20d)' ) * 180.0 / pi() AS latitude;
 latitude
----------
 20
(1 row)
```

6.2.2. Cartesian coordinates

The functions

```
  x(spoint p);
  y(spoint p);
  z(spoint p);
```

return the Cartesian `x`, `y` or `z` value of a spherical position *p*. The returned values are always between `-1.0` and `+1.0`.

Example 38. Get the Cartesian z-value of a spherical point

```
sql> SELECT z ( spoint '(10d,-90d)' ) AS z;
 z
----
 -1
(1 row)
```

You can get a float8 array of Cartesian values using the function

```
xyz(spoint p);
```

Example 39. Get the Cartesian values of a spherical point

```
sql> SELECT xyz ( spoint '(0d,0d)' ) AS cart;
  cart
---------
 {1,0,0}
(1 row)
```

6.3. strans functions

6.3.1. Converting to ZXZ

Using the function `strans_zxz(strans)`, you can convert an Euler transformation to `ZXZ` axes transformation.

Example 40. Change the transformation axes to ZXZ

Convert the transformation `strans '20d, -270d, 70.5d, XZY'` to a `ZXZ` transformation.

```
sql> SELECT strans_zxz ( strans '20d, -270d, 70.5d, XZY' );
```

6.3.2. Angles and axes

It is possible to get the components of an Euler transformation.

Table 4. Getting Euler transformation attributes

function	description
`phi`	first angle of a transformation
`theta`	second angle of a transformation
`psi`	third angle of a transformation
`axes`	transformation axes as a three letter code

The angles will always returned as a float8 value in radians. The axes are returned as a three letter code.

Example 41. Get the second axis and its rotation angle

```
sql> SELECT theta( strans '20d,30d,40d,XZY' ) * 180 / pi() AS theta;
 theta
-------
 30
(1 row)
```

```
sql> SELECT substring ( axes ( strans '20d,30d,40d,XZY' ) from 2 for 1 ) AS axis;
 axis
------
  Z
(1 row)
```

6.4. scircle functions

You can get the radius of a spherical circle in radians using the `radius` function. The center of the circle is available with the operator `@@` (Section 5.7).

Example 42. Radius of a spherical circle in degrees

```
sql> SELECT 180.0 * radius( scircle '<(0d,90d),60d>' ) / pi() AS radius;
 radius
--------
     60
(1 row)
```

6.5. sellipse functions

pgSphere provides 4 functions to get the parameters of a spherical ellipse:

Table 5. Getting spherical ellipse attributes

function	description
`lrad`	the major radius of the ellipse
`srad`	the minor radius of the ellipse
`center`	the center of the ellipse
`inc`	the inclination of the ellipse

To get the ellipse center, you can use the operator `@@` (Section 5.7) instead of using the function `center(sellipse)`.

Example 43. Get the minor radius of an ellipse

```
sql> SELECT srad ( sellipse '< { 10d, 5d }, ( 20d, 0d ), 90d >' )
        * 180.0/ pi() AS srad ;
 srad
------
  5
(1 row)
```

6.6. sline functions

6.6.1. Begin and end

To get the beginning and the end of a line, pgSphere provides two functions:

sl_beg(sline *line*);

```
sl_end(sline line);
```

Example 44. Get the beginning of a line

```
sql> SELECT sl_beg( sline '(10d, 90d, 270d, ZXZ ), 20d';
```

6.6.2. Create a meridian

You can create a meridian as a line using the function

```
meridian(float8 lng);
```

The function returns a line starting at a latitude of -90° and ending at a latitude of 90°. The line goes along the given longitude `lng` in radians.

Example 45. A meridian for longitude 20°

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)

sql> SELECT meridian (20.0 *pi() / 180.0 );
             sline
-------------------------------
 ( 270d, 90d, 20d, ZXZ ), 180d
(1 row)
```

6.7. spath functions

6.7.1. Count of points

You can get the count of points of a spherical path using the function:

```
npoints(spath path);
```

Example 46. Count of spath's points

```
sql> SELECT npoints ( spath '{(0,0),(1,0)}' );
 npoints
---------
       2
 (1 row)
```

6.7.2. Positions at a path

pgSphere provides two functions to get points at a path.

```
spoint(spath path, int4 i);
spoint(spath  path, float8 f);
```

The first function returns the `i`-th point of a path. If `i` is less than 1 or larger than the count of spath points, the function returns `NULL`. The second function does nearly the same, but does linear interpolation between edge positions.

Example 47. Get the "center" of a one segment spath

```
sql> SELECT spoint ( spath '{(0d,0d),(30d,0d)}', 1.5 );
  spoint
------------
 (15d , 0d)
(1 row)
```

6.8. spoly functions

6.8.1. Count of edges

Similar to an spath (Section 6.7.1), you can get the count of edges of a spherical polygon using the function:

`npoints``(spoly` *`polygon`*`);`

Example 48. Count of edges of a spherical polygon

```
sql> SELECT npoints ( spoly '{(0,0),(1,0),(1,1)}' );
 npoints
---------
       3
 (1 row)
```

6.9. sbox functions

The functions

`sw``(sbox` *`box`*`);`
`ne``(sbox` *`box`*`);`
`se``(sbox` *`box`*`);`
`nw``(sbox` *`box`*`);`

return the corresponding southwest, northeast, southeast, or northwest edge. The returned value will be a spherical point.

Example 49. The southwest edge of a box

```
sql> SELECT sw ( sbox '( (0d,0d), (90d,0d) )' ) ;
```

7. Create an index

7.1. Spherical index

pgSphere uses `GiST` to create spherical indices. An index speeds up the execution time of operators `<@`, `@`, `&&`, `#`, `=`, and `!=`. You can create an index with the following spherical data types:

- point (spoint)
- circle (scircle)
- line (sline)
- ellipse (sellipse)
- polygon (spoly)
- path (spath)
- coordinates range (sbox)

Example 50. Simple index of spherical points

```
CREATE TABLE test (
  pos spoint NOT NULL
);
-- Put in data now
CREATE INDEX test_pos_idx USING GIST ON test (pos);
VACUUM ANALYZE test;
```

8. Usage examples

8.1. General

tbw

8.2. Geographical

tbw

8.3. Astronomical

8.3.1. Coordinates transformation

A commonly used task is a coordinate transformation. With the parameters of a new coordinate system (plane) relative to an old one,

Ω	longitude of the ascending node	angle between line of nodes and the zero point of longitude in the old plane.
ω	argument of pericenter	the angle from the ascending node to the position in the new plane.
i	inclination	angle between the new plane and the old plane.

you can do a transformation of an object *object* from an old into a new coordinate system using:

```
object - strans 'ω, i, Ω'
```

or

```
object - strans (ω, i, Ω)
```

Otherwise, for a transformation of an object *object* from the new into the old coordinate system, use the operator +:

```
object + strans 'ω, i, Ω'
```

or

```
object + strans (ω, i, Ω)
```

Example 51. perihelion and aphelion coordinates of a comet's orbit

We are assuming the orbital elements of a comet are Ω=30°, i=60° and ω=90°. We get the spherical position of perihelion and aphelion with:

```
sql> SELECT set_sphere_output('DEG');
 set_sphere_output
-------------------
 SET DEG
(1 row)

sql> SELECT spoint '(0,0)' + strans '90d,60d,30d' AS perihelion;
   perihelion
--------------
 (120d , 60d)
(1 row)

sql> SELECT spoint '(180d,0)' + strans '90d,60d,30d' AS aphelion;
```

```
       aphelion
---------------
 (300d , -60d)
(1 row)
```

9. FAQ

tbw

10. Appendix

10.1. Changes from version 1.0 to 1.1

- Speed up index query time. The index scan is much faster than in 1.0.
- Remove B-tree index. It was used to cluster the index.
- This version is compatible to PostgreSQL 8.4

10.2. Changes from version 1.1 to 1.1.5

- Modify behavior of function `set_sphere_output_precision`.
- Compatibility with PostgreSQL 9.2 and later.
- Creating spath and spolygon objects "as aggregate" from tables should now work.
- Improved accuracy of spoint (great circle) distance calculation.
- Add new contains operators consistent with the geometry operators of the plain used in PostgreSQL 8.2 and later: <@ and @>, the old operators @ and ~ still work.
- Improved correctness of spatial indexing.