30 / 686

**Chapter 4**

**使用 PostGIS: 数据管理和查询**

**4.1**

**GIS 对象**

由PostGIS的支持的GIS对象是OpenGIS Consortium (OGC)定义的GIS“简单特性”的超集。自0.9版本起， PostGIS支持

OGC标准中“SQL简单特性”所涉及的所有对象和特性。PostGIS扩展了这个标准以便支持3DZ，3DM和4D坐标

**4.1.1**

**OpenGIS的 WKB 和 WKT格式**

OpenGIS规范定义了两种表达空间对象的标准方法：Well-Known Text (WKT) 格式和Well-Known Binary (WKB)格式

这两种格式都包含了关于构成该对象所需要的对象的类型和坐标信息。WKT表示空间对象特征的样例如下

· POINT(0 0)

· LINESTRING(0 0,1 1,1 2)

· POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1, 2 1, 2 2, 1 2,1 1))

· MULTIPOINT((0 0),(1 2))

· MULTILINESTRING((0 0,1 1,1 2),(2 3,3 2,5 4))

· MULTIPOLYGON(((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)), ((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))

· GEOMETRYCOLLECTION(POINT(2 3),LINESTRING(2 3,3 4))

OpenGIS规范也需要空间对象的内部存储格式，包括空间参照系统即SRID等。SRID在创建用于插入数据库空间对象时的是必需的。

使用以下接口可以上述两种格式的对象输出：

bytea WKB = ST\_AsBinary(geometry);

text WKT = ST\_AsText(geometry);

geometry = ST\_GeomFromWKB(bytea WKB, SRID);

geometry = ST\_GeometryFromText(text WKT, SRID);

例如：下面是一个正确的插入语句，用了创建和插入OGC规范的空间对象

INSERT INTO geotable ( the\_geom, the\_name )

VALUES ( ST\_GeomFromText(‘POINT(-126.4 45.32)’, 312), ‘A Place’);

31 / 686

**4.1.2**

**PostGIS EWKB, EWKT 和 Canonical 形式**

OGC 规范只支持2D几何对象，并且在几何对象的输入和输出格式中也不带有SRID信息。PostGIS扩展了OGC规范（每一个有效的WKB/WKT对象都是一个有效的EWKB/EWKT对象），但未来可能会有变化，特别是OGC规范出台了与我们扩展格式相冲突的格式。这样你应该不依赖这样的特性

PostGIS EWKB/EWKT 格式添加了3dm,3dz,4d 坐标支持，并且带有SRID信息

EWKT表述的几何对象样例如下。带\*星号的是这个版本PostGIS新的对象支持

· POINT(0 0 0) -- XYZ

· SRID=32632;POINT(0 0) -- XY with SRID

· POINTM(0 0 0) -- XYM

· POINT(0 0 0 0) -- XYZM

· SRID=4326;MULTIPOINTM(0 0 0,1 2 1) -- XYM with SRID

· MULTILINESTRING((0 0 0,1 1 0,1 2 1),(2 3 1,3 2 1,5 4 1))

· POLYGON((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0))

· MULTIPOLYGON(((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0)),((-1 -1 0,-1 -2 0,-2 -2 0,-2 -1 0,-1 -1 0)))

· GEOMETRYCOLLECTIONM( POINTM(2 3 9), LINESTRINGM(2 3 4, 3 4 5) )

· MULTICURVE( (0 0, 5 5), CIRCULARSTRING(4 0, 4 4, 8 4) )

· POLYHEDRALSURFACE( ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)), ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0

0 0)), ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)), ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)), ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )

· TRIANGLE ((0 0, 0 9, 9 0, 0 0))

· TIN( ((0 0 0, 0 0 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 0 0 0)) )

如下接口使用时已经支持如下格式的输入与输出：

bytea EWKB = ST\_AsEWKB(geometry);

text EWKT = ST\_AsEWKT(geometry);

geometry = ST\_GeomFromEWKB(bytea EWKB);

geometry = ST\_GeomFromEWKT(text EWKT);

例如一个有效的创建和出入PostGIS空间对象的SQL语句如下：

INSERT INTO geotable ( the\_geom, the\_name )

VALUES ( ST\_GeomFromEWKT(‘SRID=312;POINTM(-126.4 45.32 15)’), ‘A Place’ )

Canonical 形式表述PostgreSQL类型场景是；你做一个简单查询（没有函数调用），比如插入、更新或拷贝时候返回返回的几何类型表述。对于PostGIS 的“geometry“类型，Canonical表述方式是：

- Output

- binary: EWKB

ascii: HEXEWKB (EWKB in hex form)

- Input

- binary: EWKB

ascii: HEXEWKB|EWKT

流入该语句读取EWKT格式数据，然后处理canonical输入/输出，最后返回HEXEWKB 格式数据:

32 / 686

=# SELECT ‘SRID=4;POINT(0 0)’::geometry;

geometry

----------------------------------------------------

01010000200400000000000000000000000000000000000000

(1 row)

**4.1.3**

**SQL-MM 规范第三部分**

SQL Multimedia Applications Spatial 规范扩展了”Simple features for SQL”规范，包括定义了一些圆环插值曲线（circularly interpolated curves）。SQL-MM 定义了包括3dm, 3dz and 4d 坐标，但不允许这些格式的坐标携带SRID信息

目前该规范还不完全支持WKT的扩展格式。一些简单的曲线几何对象样例如下：

· CIRCULARSTRING(0 0, 1 1, 1 0)

CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0)

CIRCULARSTRING 是基本的曲线类型，在线性范畴内，和LINESTRING类似。一个简单段（可能是线段也可能是弧线段）包括3个点：起点、终点和这两点之间的其他点。例外是封闭的圆，起点和终点是相同的。在这种情况下第二个点必须是弧线的中心。为了把弧线连接起来，前一个弧线的终点必须是下一个弧线的起点，就像在LINESTRING中一样。这意味着一个有效的CIRCULARSTIRNG对象的点个数必须大于1，且为奇数个。

· COMPOUNDCURVE(CIRCULARSTRING(0 0, 1 1, 1 0),(1 0, 0 1))

一个复合曲线类型对象是一个简单的、连贯的曲线，它包含曲线段和直线段。这意味着该对象除了要有良好格式（比如WKT格式）的对象组成外，每个对象的终点（除了最后一个点）必须与下一个几何对象的起点重合。

· CURVEPOLYGON(CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),(1 1, 3 3, 3 1, 1 1))

在一个CURVE POLYGON对象中包含CURVE对象的例子： URVEPOLYGON(COMPOUNDCURVE(CIRCULARSTRING(0 0,2 0, 2 1,2 3, 4 3),(4 3, 4 5, 1 4, 0 0)), CIRCULARSTRING(1.7 1, 1.4 0.4, 1.6 0.4, 1.6 0.5, 1.7 1) )

CURVEPOLYGON与POLYGON对象很类似，有一个外环和0个或更多内环。所不同的是一个环的形式可以为

circular string, LINESTRING或复合的线（compound string）。

至于PostGIS 1.4版本， PostGIS 支持CURVE POLYGON对象中包含一个复合的曲线（compound curves）

· MULTICURVE((0 0, 5 5),CIRCULARSTRING(4 0, 4 4, 8 4))

MULTICURVE 是一个CURVE collection对象（集合对象），该对象可以包含：线性对象、圆形对象和复合线性对象

· MULTISURFACE(CURVEPOLYGON(CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),(1 1, 3 3, 3 1, 1 1)),((10 10, 14 12, 11 10,

10 10),(11 11, 11.5 11, 11 11.5, 11 11)))

这是一个SURFACE collection对象例子，可以是（线性的）polygon或曲线的polygon对象

**注意**

PostGIS 1.4版本之前不支持curve polygon对象中包含复合曲线类型，但PostGIS 1.4及更高版本支持curve polygon对象中包含复合曲线类型

**注意**

SQL-MM规范里面的浮点型精度的点之间比较的误差当前是1E-8.（译者注：这种场景通常是点之间的比较，比如是否是同一个点，距离计算等等）。





33 / 686

**4.2**

**PostGIS 地理数据类型**

地理数据类型提供了对地理坐标（有时候也称为大地坐标，或者经度/维度，或者维度/经度）上的空间特征的原生支持。地理坐标是一种用弧度单位表示的球面坐标。

PostGIS几何数据类型的基本类型是平面类型。平面上两点最短路径是一条直线。这意味着几何数据类型的计算（面积，距离，长度，交集等等）可以使用笛卡尔数学和矢量方式来计算。

PostGIS 地理类型的基础是一个球体。球面上两点的最短距离是一个大圆弧。这意味着地理计算（areas, distances, lengths, intersections等等）必须在球面上使用更多复杂函数计算。要想得到更精确的计算值，计算时候必须要把地球是一个椭球因素考虑进去，但计算实际上也变得非常复杂了。

因为基本的数学计算（地理数据类型对象之间的关系及运算等等）要复杂得多，所以支持地理类型的函数要比支持几何数据类型的函数要少得多。将来的话，随着新算法的增加，PostGIS对地理类型数据的支持能力会随之扩展。

该类型的一个限制是当前只支持WGS 84 参考系(SRID:4326)，数据格式是经纬度。该参考系使用的是新数据类型地理数据类型。GEOS的函数都不支持这种新类型。作为一种解决方案，用户可以在几何数据类型和地理数据类型之间进行来回转换。（译者注：以便充分利用几何数据类型的函数）。

新的geography地理数据类型使用PostgreSQL 8.3+ 类型修饰符（typmod）的方式来定义字段，因此创建一个地理数据类型字段是一个简单的步骤。除了curve类型，所有的OGC标准地理数据类型都支持。

**4.2.1**

**地理数据类型基础**

地理数据类型目前只支持最简单简单特征对象。如果SRID值为4326，那么标准几何类型数据可以自动转换为geography类型。你也可以使用EWKT和EWKB规范描述的数据格式来插入数据

· POINT: 创建一个带有2D 点几何数据类型的表：

CREATE TABLE testgeog(gid serial PRIMARY KEY, the\_geog geography(POINT,4326) );

创建一个带有Z坐标的Point类型列

CREATE TABLE testgeog(gid serial PRIMARY KEY, the\_geog geography(POINTZ,4326) );

· LINESTRING

· POLYGON

· MULTIPOINT

· MULTILINESTRING

· MULTIPOLYGON

· GEOMETRYCOLLECTION

新的geography类型字段不会注册在geometry\_columns表中（就是在该表中能查到，实际上这并非一个表，而是一个视图，老的PostGIS中是一个表，在新版本中是一个视图），geography类型字段注册在一个新的视图中geography\_columns ，该视图依赖于更底层的系统表，因此没有必要使用AddGeom...之类的函数来手动注册在该视图中

现在：查看视图"geography\_columns" 确认你的表是否在该视图里面

你可以使用CREATE TABLE语法创建一个带有GEOGRAPHY 类型的列，没有必要使用AddGeometryColumns() 函数来创建列以便在元数据视图geography\_columns中进行表注册

CREATE TABLE global\_points (

id SERIAL PRIMARY KEY,

name VARCHAR(64),

location GEOGRAPHY(POINT,4326)

);

34 / 686

注意列location 是一个GEOGRAPHY 类型，该类型字段有两种可选的修饰符（译者注：所谓修饰符相当于Java里面变量里面的一些关键字比如public static CARD\_TYPE，那么这里的public、static就是一种修饰符）：第一种是类型修改器（typmod），它能限制列的形状和维度。还有一个SRID修饰符，是一个整型数字，用于限制字段类型的坐标系。

类型修饰符可能的值有：POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON。该类型修饰符也支持通过后缀坐标：Z,M和ZM来限制维度。因此，一个修饰符‘LINESTRINGM’表示一个3维的LINESTRING对象，它把第三个维度当做一个测量维度。类似地，‘POINTZM’ 表示该字段需要一个四维数据。

SRID修饰符目前只限于：4326（WGS84）这个值。如果你没有指定SRID值，那么默认使用值是0（未定义的球面），所有的计算基于WGS84坐标系来计算。

未来可选的SRID值将会允许在除了WGS84参考系外的坐标系上计算。

一旦你创建了表，你可以在GEOGRAPHY\_COLUMNS 视图中看到该表：

-- See the contents of the metadata view

SELECT \* FROM geography\_columns;

如果你使用的是GEOMETRY类型列，你可以插入该类型数据到表中，像正常其他类型数据一样的方式：

-- Add some data into the test table

INSERT INTO global\_points (name, location) VALUES (‘Town’, ST\_GeographyFromText(‘SRID=4326; ←

POINT(-110 30)’) );

INSERT INTO global\_points (name, location) VALUES (‘Forest’, ST\_GeographyFromText(‘SRID ←

=4326;POINT(-109 29)’) );

INSERT INTO global\_points (name, location) VALUES (‘London’, ST\_GeographyFromText(‘SRID ←

=4326;POINT(0 49)’) );

在GEOMETRY类型字段上面创建索引也一样。PostGIS 会意识到该列类型是一个GEOGRAPHY 类型，并创建一个合适的基于球面的索引，而不是一个使用GEOMETRY类型的常规平面索引

-- Index the test table with a spherical index

CREATE INDEX global\_points\_gix ON global\_points USING GIST ( location );

对GEOGRAPHY类型的查询和测量函数使用的单位都是米，因此距离参数应该用米表示，返回值也应该是米（对于面积来说是平方米）

-- Show a distance query and 注意, London is outside the 1000km tolerance

SELECT name FROM global\_points WHERE ST\_DWithin(location, ST\_GeographyFromText(‘SRID ←

=4326;POINT(-110 29)’), 1000000);

你可以使用GEOGRAPHY类型来计算一架飞机从西雅图飞到伦敦这条航线(LINESTRING(-

122.33 47.606, 0.0 51.5))到雷克雅末克（冰岛首都）(POINT(-21.96 64.15))的距离，这是非常强大的功能：

-- Distance calculation using GEOGRAPHY (122.2km)

SELECT ST\_Distance(‘LINESTRING(-122.33 47.606, 0.0 51.5)’::geography, ‘POINT(-21.96

←

64.15)’:: geography);

-- Distance calculation using GEOMETRY (13.3 "degrees")

SELECT ST\_Distance(‘LINESTRING(-122.33 47.606, 0.0 51.5)’::geometry, ‘POINT(-21.96 64.15) ←

‘:: geometry);

GEOGRAPHY 类型用来计算雷克雅默克到西雅图与伦敦的航线之间的最短球面距离。

[Great Circle mapper](http://gc.kls2.com/cgi-bin/gc?PATH=SEA-LHR)GEOMETRY 类型计算的是雷克雅默克到西雅图与伦敦的航线之间笛卡尔距离，这种计算方式是无意义的，因为它计算的是两者之间的平面距离。实际上这两者之间的距离的单位也许叫做“度“，但这个结果并不对应两个点之间的真实角度差，所以称为”度”甚至是不准确的。

35 / 686

**4.2.2**

**什么时候使用geography类型而不是geometry数据类型？**

新的GEOGRAPHY类型允许你以经度/维度坐标的方式存储数据，但这是有代价的：支持 GEOGRAPHY类型的函数要比支持GEOMETRY类型的函数要少一些，而前者会多花一些CPU时间来执行。

你选择的数据类型应该支持你构建的应用。你的数据跨全球了还是一个大的区域，或者是当地的一个州，县还是直辖市？

如果你的数据只是包含了一个小范围区域，考虑到性能和功能的可用性，你可能发现选择一个合适的投影系，并使用GEOMETRY类型是最佳选择。

如果你的数据包含的是全球性数据或者包含了一个州的数据，你可能发现GEOGRAPHY类型能够让你构建一个不需要考虑投影细节的应用系统。你需要把你的数据以经度/维度格式存储，使用支持GEOGRAPHY类型的函数。

如果你不理解投影系，你也不想学习这些知识，你也接受支持GEOGRAPHY类型函数的局限性，那么对于你来说，使用GEOGRAPHY类型要比使用GEOMETRY类型要更容易一些。你只需要把经纬度数据加载到表中，然后就可以使用了

参考13.10 节Geography 和. Geometry分别有哪些函数支持。参考13.3 节获取支持geography类型数据的函数列表和对应的描述。

**4.2.3**

**关于Geography 深度问题**

1. 你的计算是在球面上还是椭球体上？

默认情况下所有的距离和面积计算都是在椭球体上。你应该会发现椭球体上的计算结果会和当地比较好的平面参考系的计算结果相差不大。对于大型的面积计算，椭球上的计算要比任意的平面投影系上计算都要精确。所有的地理类型函数都有使用球面计算的参数，只需要把函数的逻辑参数设置为FALSE。这会某种程度上加速计算，特别是对于那些几何类型对象很简单的情况。

2. 关于国际日期变更线和北极线是怎样的？

所有的计算都和国际日期变更线或者北极线无关，坐标都是椭球体坐标（经度/维度），因此如果有一个几何对象穿越了国际日期变更线，从计算角度来说和其他任意形状都没有关系。

3. 你能处理的最长弧线是哪条？

我们把两个点之间的大圆弧当做这两个点的“穿插线或者说连线”。这意味着任意两点有两种方式连接，这由你在大圆上往哪个方向走。（译者注：在大圆上设置两个点，这两个点分隔的圆弧除了在穿过直径的那两个点外，其他任意两个点划分的弧度都是不同的）。我们的代码都假定这两个点是以较短的那条路径连接的。因此，哪些包含超过180度弧的图形都不符合PostGIS格式。

4. 为何计算Europe / Russia 的面积很慢？我这里插入表中的几何区域是比较大的

因为多边形是非常非常巨大！在GIS中，区域太大不好原因有两点：他们的边界是巨大的，无论你运行的是什么样的查询，查询都会导致全表扫描；第二，他们的顶点数目是巨大的，测试（距离，几何对象之间的包含关系）必须遍历他们的顶点列表至少有一次，有时是N次（N是其他候选特征的顶点数目）。对于GEOMETRY类型，我们建议，当你有非常大的多边形对象时候，但在小范围内做查询，你应该把你的几何数据分成更小的部分，从而使索引可以有效地查询部分的对象，这样查询也不必每次扫描全部对象。你能把整个欧洲地理特征都存储在一个polygon几何列中，但并不意味着你应该（可以）这么干！

**4.3**

**使用 OpenGIS 规范**

OpenGIS 的"Simple Features Speciﬁcation for SQL"规范定义了标准的GIS对象类型，以及处理这些类型对象的函数以及一些元数据表。为了保证元数据的一致性，OpenGIS规范定义并实现了一些具体的程序来实现一些操作比如创建和删除空间列。

有两个OpenGIS元数据表： SPATIAL\_REF\_SYS 和 GEOMETRY\_COLUMNS（译者注：这实际上是一个视图，早期版本的PostGIS，这是一个表）。SPATIAL\_REF\_SYS 表包含了空间数据库中使用的数值ID和参考系的文本描述。

36 / 686

**4.3.1**

**SPATIAL\_REF\_SYS 表和空间参考系**

spatial\_ref\_sys 是一个兼容OGC规范的PostGIS的系统表，该表包含了3000度个已知的空间参考系 [（spatial reference](http://www.sharpgis.net/post/2007/05/Spatial-references2c-coordinate-systems2c-projections2c-datums2c-ellipsoids-e28093-confusing.aspx) [systems](http://www.sharpgis.net/post/2007/05/Spatial-references2c-coordinate-systems2c-projections2c-datums2c-ellipsoids-e28093-confusing.aspx))，以及参考系之间转换/重投影时候的细节。

虽然PostGIS的表spatial\_ref\_sys table中包含了超过3000种较为常用的空间参考系定义（这些参考系是由Proj4库提供处理支持的），但这个表并不能涵盖全部已知的参考系，如果你对Proj4库熟悉的话，你甚至可以自定义自己的投影系统。记住大多数空间参考系是区域性的，超出他们的使用区域范围是无意义的。

一个很棒的找到没有在这里定义的空间参考系的优质资源：<http://spatialreference.org/>

一些较为常用的空间参考系是： 4326 - WGS 84 Long Lat, 4269 - NAD 83 Long Lat, [3395 -WGS 84 World Mercator,](http://spatialreference.org/ref/epsg/3395/) 2163 - US National Atlas Equal Area, 以及NAD 83的各个投影系, WGS 84 UTM参考系（Universal Transverse Mercator Projection--通用横轴墨卡托投影）—— UTM参考系是最理想的测量参考系之一，但是只覆盖了6个弧度区域。

各种US州平面空间参考系（单位为米或英尺）——通常每个州有1到2个参考系。大多数参考系都是以米为单位的，但也有不少以英尺为单位或者是ESRI创建的参考系，你可以在这里找到：[spatialreference.org.](http://spatialreference.org)

更多关于你该使用哪种UTM 参考系来处理你的感兴趣区域数据，请参考： [utmzone PostGIS plpgsql helper](http://trac.osgeo.org/postgis/wiki/UsersWikiplpgsqlfunctionsDistance) [function.](http://trac.osgeo.org/postgis/wiki/UsersWikiplpgsqlfunctionsDistance)

SPATIAL\_REF\_SYS 表定义如下：

CREATE TABLE

srid

auth\_name

auth\_srid

srtext

proj4text

spatial\_ref\_sys (

INTEGER NOT NULL PRIMARY KEY,

VARCHAR(256),

INTEGER,

VARCHAR(2048),

VARCHAR(2048)

)

表 SPATIAL\_REF\_SYS 包含的列如下：

[SRID](http://en.wikipedia.org/wiki/SRID) ：一个能标识数据库里面空间参考系（SRS）唯一性的整型值。

AUTH\_NAME：本参考系统所引用的标准或标准体的名称。例如，“EPSG”将是一个有效的AUTH\_NAME。

AUTH\_SRID：一个由AUTH\_NAME引用的由Authority定义的空间参考系的ID值。在EPSG中，这由EPSG 投影代码（projection code）来决定。

SRTEXT ：空间参考系的WKT描述。一个WKT描述的SRS样例如下：

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]

]

EPSG投影代码列表及其与WKT描述的对应关系，请前往这里查找： [http://www.opengeospatial.org/.](http://www.opengeospatial.org/)

关于WKT的一般讨论，请参考OpenGIS的 "Coordinate Transformation Services Implementation Speciﬁcation" http://www.opengeospatial.org/standards. 关于ESPG机构（European Petroleum Survey Group）和他们的空间参考系的数据集，请查看这里： [http://www.epsg.org.](http://www.epsg.org/)

37 / 686

PROJ4TEXT ：PostGIS 使用Proj4库提供坐标转换功能。PROJ4TEXT 列包含一个具体SRID的Proj4坐标定义的字符串形式。例如：

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

更多关于此的信息，请参考Proj4 项目地址： http://trac.osgeo.org/proj/.脚本文件 spatial\_ref\_sys.sql 包含了所有EPSG投影系的SRTEXT 和 PROJ4TEXT 格式定义

**4.3.2**

**关于几何类型列的视图**

在PostGIS 2.0.0版本之前， geometry\_columns 是一个可以直接编辑的白哦，但有时候和真实的定义的几何数据列不同步。在

PostGIS 2.0.0中, GEOMETRY\_COLUMNS 变成了一个视图，和之前版本的字段结构是一样的，但从数据库系统目录表中读取数据：

\d geometry\_columns

View "public.geometry\_columns"

Column

|

Type

| Modifiers

-------------------+------------------------+-----------

f\_table\_catalog

f\_table\_schema

f\_table\_name

| character varying(256) |

| character varying(256) |

| character varying(256) |

f\_geometry\_column | character varying(256) |

coord\_dimension

srid

type

| integer

| integer

| character varying(30)

|

|

|

列的定义和之前的版本相比没有变化：

F\_TABLE\_CATALOG,F\_TABLE\_SCHEMA,F\_TABLE\_NAME ：包含几何数据列的表全名。注意：术语“catalog”和“schema”都是Oracle语法的。PostgreSQL里面没有和"catalog"对应的对象，因此这一列是空的——至于"schema"概念，PostgreSQL里面就是schema 名称（默认是public）

F\_GEOMETRY\_COLUMN ：含有空间特征的表里面的几何数据类型列。

COORD\_DIMENSION ：几何数据列的空间维度（2,3或4维）

SRID：含有空间特征的表里面的几何列的空间参考系ID值，SPATIAL\_REF\_SYS表的外键。

TYPE：空间对象的类型。为了把空间列限制为单个类型，使用下面其中之一： POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION 或者对应的XYM坐标版本POINTM, LINESTRINGM, POLYGONM, MULTIPOINTM, MULTILINESTRINGM, MULTIPOLYGONM,GEOMETRYCOLLECTIONM. 对于异构类型（混合类型），你可以使用"GEOMETRY" 作为定义该字段时候的类型

**注意**

注意：TYPE这个参数（可能）不是OpenGIS的规范，但是是保证类型一致性的必要条件

**4.3.3**

**创建一个带有空间数据的表**

创建一个含有空间数据的表是很简单的，一步即可。正如下面创建一个带有2D LINESTRING 几何类型列的道路表一样，该字段列用WGS 84经纬度格式表示：

CREATE TABLE ROADS ( ID int4

, ROAD\_NAME varchar(25), geom geometry(LINESTRING,4326) );



38 / 686

我们可以使用标准的ALTER TABLE 语法来添加额外的列，和接下来我们添加一个3D LINESTRING类型列一样的方法：

ALTER TABLE roads ADD COLUMN geom2 geometry(LINESTRINGZ,4326);

为了后向兼容，你依然可以用管理函数两步创建空间表：

· 创建一个正常的不含空间数据的表：

例如: CREATE TABLE ROADS ( ID int4, ROAD\_NAME varchar(25) )

· 使用OpenGIS "AddGeometryColumn"函数来添加一个空间类型列到表中，参考函数AddGeometryColumn获取更多信息。

该函数用法是：

AddGeometryColumn(

<schema\_name>,

<table\_name>,

<column\_name>,

<srid>,

<type>,

<dimension>

)

或者使用当前连接下的schema值，即current\_schema()函数返回的值：

AddGeometryColumn(

<table\_name>,

<column\_name>,

<srid>,

<type>,

<dimension>

)

样例1: SELECT AddGeometryColumn(‘public’, ‘roads’, ‘geom’, 423, ‘LINESTRING’, 2)

样例2: SELECT AddGeometryColumn( ‘roads’, ‘geom’, 423, ‘LINESTRING’, 2)

下面的例子是用SQL语句来创建表，添加一个空间类型列（假定存在SRID值为128的空间参考系）：

CREATE TABLE parks (

park\_id

park\_name

park\_date

park\_type

INTEGER,

VARCHAR,

DATE,

VARCHAR

);

SELECT AddGeometryColumn(‘parks’, ‘park\_geom’, 128, ‘MULTIPOLYGON’, 2 );

这是另一个例子，使用几何数据类型，SRID值没有定义，为0：

CREATE TABLE roads (

road\_id INTEGER,

road\_name VARCHAR

);

SELECT AddGeometryColumn( ‘roads’, ‘roads\_geom’, 0, ‘GEOMETRY’, 3 );

**4.3.4**

**手动在geometry\_columns表中注册几何数据类型列**

函数AddGeometryColumn()可以创建一个几何列，并把这个新列注册在视图geometry\_columns中。如果你的软件需要使用视图geometry\_columns，那么任何几何类型的列都必须在这个视图中注册。从PostGIS 2.0开始, geometry\_columns不再是一个可编辑的表，而是一个视图，所有的几何类型列都是自动注册的（译者注：如果使用约束创建几何列，那么当创建一个依赖于该列的视图时候，该列不会自动注册到geometry\_columns中，原文说都自动注册，是有问题的）

39 / 686

然而如果在创建列的过程中没有定义成一个具体的几何类型，那么该列可能被注册为一个一般类型的geometry类型列。下面这些样例中有两个可能会出现这种状况，但是你不能在SQL视图或者批量插入中使用AddGeometryColumn方法来创建几何列（译者注：原因就是上面说的那样）。对于这些情况，你可以通过再添加列约束的方式来把列正确地注册到geometry\_columns 视图中。注意在 PostGIS 2.0+版本中，如果你的几何列是通过类型修饰符方式创建的，那么列会自动进行注册，不需要做任何事情。（译者注：最好使用2.0以后版本啊，省去了各种蛋疼的问题！）

--Lets say you have a view created like this

CREATE VIEW

public.vwmytablemercator AS

SELECT gid, ST\_Transform(geom,3395) As geom, f\_name

FROM public.mytable;

-- For it to register correctly in PostGIS 2.0+

-- You need to cast the geometry

--

DROP VIEW public.vwmytablemercator;

CREATE VIEW

public.vwmytablemercator AS

SELECT gid, ST\_Transform(geom,3395)::geometry(Geometry, 3395) As geom, f\_name

FROM public.mytable;

-- If you know the geometry type for sure is a 2D POLYGON then you could do

DROP VIEW public.vwmytablemercator;

CREATE VIEW

public.vwmytablemercator AS

SELECT gid, ST\_Transform(geom,3395)::geometry(Polygon, 3395) As geom, f\_name

FROM public.mytable;

--Lets say you created a derivative table by doing a bulk insert

SELECT poi.gid, poi.geom, citybounds.city\_name

INTO myschema.my\_special\_pois

FROM poi INNER JOIN citybounds ON ST\_Intersects(citybounds.geom, poi.geom);

--Create 2d index on new table

CREATE INDEX idx\_myschema\_myspecialpois\_geom\_gist

ON myschema.my\_special\_pois USING gist(geom);

-- If your points are 3D points or 3M points,

-- then you might want to create an nd index instead of a 2d index

-- like so

CREATE INDEX my\_special\_pois\_geom\_gist\_nd

ON my\_special\_pois USING gist(geom gist\_geometry\_ops\_nd);

--To manually register this new table’s geometry column in geometry\_columns

-- Note that this approach will work for both PostGIS 2.0+ and PostGIS 1.4+

-- For PostGIS 2.0 it will also change the underlying structure of the table to

-- to make the column typmod based.

-- For PostGIS prior to 2.0, this technique can also be used to register views

SELECT populate\_geometry\_columns(‘myschema.my\_special\_pois’::regclass);

--If you are using PostGIS 2.0 and for whatever reason, you

-- you need the old constraint based definition behavior

-- (such as case of inherited tables where all children do not have the same type and srid)

-- set new optional

use\_typmod argument to false

SELECT populate\_geometry\_columns(‘myschema.my\_special\_pois’::regclass, false);

虽然旧的基于约束的办法依然支持，在视图中一个基于约束的几何类型列不会自动注册在geometry\_columns视图中，而类型修饰符的方式可以自动注册，不管是在表中还是在视图中。在这个样例中 我们定义了两个几何列，一个是使用类型修饰符的方式，另一个是使用约束的方式。

CREATE TABLE pois\_ny(gid SERIAL PRIMARY KEY

, poi\_name text, cat varchar(20)

, geom geometry(POINT,4326) );

SELECT AddGeometryColumn(‘pois\_ny’, ‘geom\_2160’, 2160, ‘POINT’, 2, false);

40 / 686

如果我们在psql下执行

\d pois\_ny;

我们发现这两种定义方式是不同的：一个是类型修饰符（typmod）方式，一个是约束的方式：

Table "public.pois\_ny"

Column

|

Type

|

Modifiers

-----------+-----------------------+------------------------------------------------------

gid

poi\_name

| integer

| text

| not null default nextval(‘pois\_ny\_gid\_seq’::regclass)

|

cat

| character varying(20) |

geom

| geometry(Point,4326)

|

geom\_2160 | geometry

|

Indexes:

"pois\_ny\_pkey" PRIMARY KEY, btree (gid)

Check constraints:

"enforce\_dims\_geom\_2160" CHECK (st\_ndims(geom\_2160) = 2)

"enforce\_geotype\_geom\_2160" CHECK (geometrytype(geom\_2160) = ‘POINT’::text

OR geom\_2160 IS NULL)

"enforce\_srid\_geom\_2160" CHECK (st\_srid(geom\_2160) = 2160)

在 geometry\_columns视图中，这两种方式都成功地注册了：

SELECT f\_table\_name, f\_geometry\_column, srid, type

FROM geometry\_columns

WHERE f\_table\_name = ‘pois\_ny’;

f\_table\_name | f\_geometry\_column | srid | type

-------------+-------------------+------+-------

pois\_ny

pois\_ny

| geom

| geom\_2160

| 4326 | POINT

| 2160 | POINT

然而——如果我们创建这样创建一个视图：

CREATE VIEW vw\_pois\_ny\_parks AS

SELECT \*

FROM pois\_ny

WHERE cat=‘park’;

SELECT f\_table\_name, f\_geometry\_column, srid, type

FROM geometry\_columns

WHERE f\_table\_name = ‘vw\_pois\_ny\_parks’;

基于类型修饰符（typmod）方式创建的视图的列在geometry\_columns视图中成功注册了，但基于约束的却没有注册成功：

f\_table\_name

| f\_geometry\_column | srid |

type

------------------+-------------------+------+----------

vw\_pois\_ny\_parks | geom

vw\_pois\_ny\_parks | geom\_2160

| 4326 | POINT

| 0 | GEOMETRY

在未来的PostGIS版本中，可能会改变这样的结果，但目前来说，要想将视图的基于约束创建的列在geometry\_columns视图中注册成功。你可以这样做：

DROP VIEW vw\_pois\_ny\_parks;

CREATE VIEW vw\_pois\_ny\_parks AS

SELECT gid, poi\_name, cat

, geom

, geom\_2160::geometry(POINT,2160) As geom\_2160

FROM pois\_ny

WHERE cat=‘park’;

PostGIS 2.2.0dev 开发手册

41 / 686

SELECT f\_table\_name, f\_geometry\_column, srid, type

FROM geometry\_columns

WHERE f\_table\_name = ’vw\_pois\_ny\_parks’;

f\_table\_name

| f\_geometry\_column | srid | type

------------------+-------------------+------+-------

vw\_pois\_ny\_parks | geom

vw\_pois\_ny\_parks | geom\_2160

| 4326 | POINT

| 2160 | POINT

**4.3.5**

**确保使用兼容OpenGIS的几何类型**

PostGIS 兼容Open Geospatial Consortium (OGC)的 OpenGIS 规范。同样，许多PostGIS函数需要或者更准确地说，会假定它们要处理的数据是简单的而且是有效的。例如PostGIS不支持计算一个外部带孔的多边形的面积，或者从一个非简单的边界创建一个多边形对象。

根据OGC规范，一个简单几何对象指没有异常几何点，没有自相交或者自相切，主要是0或1维几何对象（比如[MULTI]POINT, [MULTI]LINESTRING）。从另一个角度来说，几何数据类型的有效性，主要是指2维几何对象(比如 [MULTI]POLYGON)，并且定义了一个有效多边形的特征集合。每一个几何类的描述，包括进一步描述几何简单性和有效性的具体条件。

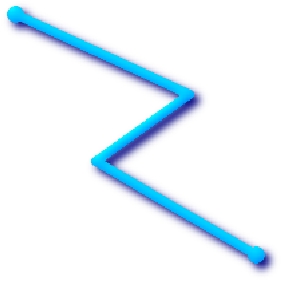
POINT 点类型是一种继承了一个0维几何类型的类型

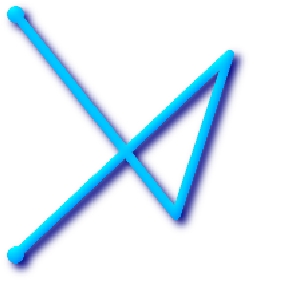
MULTIPOINT 的简单性是指没有任意两个点重复的

LINESTRING的简单性是指该对象不存在两个重复的点（除非是终点，并且是一个线性环的终点，该环是封闭的）

(a)

(b)





PostGIS 2.2.0dev 开发手册

42 / 686

(c)

(d)

(a) 和 (c) 是简单的LINESTRINGs, (b) 和 (d)不是

A MULTILINESTRING is simple only if all of its elements are simple and the only intersection between any two elements

occurs at POINTs that are on the boundaries of both elements.

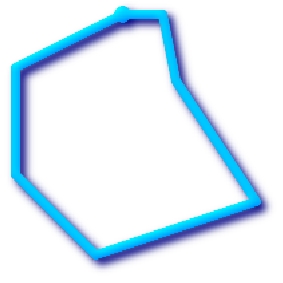
(e)

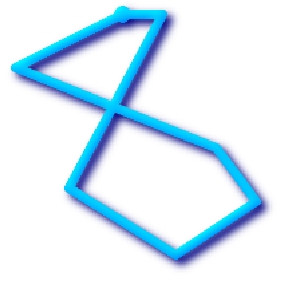
(f)

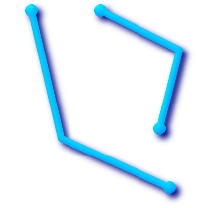
(g)

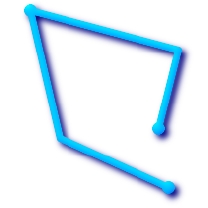
(e) 和 (f) 是简单的MULTILINESTRINGs, (g) 不是

按照定义，POLYGON 对象总是简单的。如果边界（由外环和内环组成）里面没有两个环相交。POLYGON的边界可能在某个点相切（不是相切在一个线上）。一个多边形的可能没有切线或尖峰，并且内部环必完全在外部环里面。











PostGIS 2.2.0dev 开发手册

43 / 686

(h)

(k)

(i)

(l)

(j)

(m)

(h) and (i) are valid POLYGONs, (j-m) cannot be represented as single POLYGONs, but (j) and (m) could be represented as

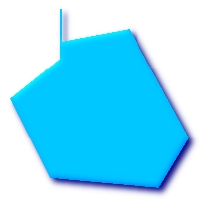
a valid MULTIPOLYGON.

一个MULTIPOLYGON是有效的当且仅当其包含的所有基本对象元素都是有效的，任何两个内部的基本元素都不能有完全相交，任何2个元素的边界可能接触，但仅在有限数量的点。













PostGIS 2.2.0dev 开发手册

44 / 686

(n)

(o)

(n) and (o) are not valid MULTIPOLYGONs. (p), however, is valid.

大多数GEOS库实现的函数都依赖于这样的假定：你的几何对象是按照OpenGIS Simple Feature Speciﬁcation要求的那样，是有效的。为了检验几何对象的简单性或者有效性，你可以使用函数ST\_IsSimple()和ST\_IsValid()

-- Typically, it doesn’t make sense to check

-- for validity on linear features since it will always return TRUE.

-- But in this example, PostGIS extends the definition of the OGC IsValid

-- by returning false if a LineString has less than 2 \*distinct\* vertices.

gisdb=# SELECT

ST\_IsValid(’LINESTRING(0 0, 1 1)’),

ST\_IsValid(’LINESTRING(0 0, 0 0, 0 0)’);

st\_isvalid | st\_isvalid

------------+-----------

t

|

f

默认情况下，PostGIS不对几何类型数据做有效性检查，因为对于复杂查询，检查这种有效性会耗费很多CPU时间，特别是多边形对象。如果你不相信自己的数据源是有效的，你可以手动给表添加约束用于约束插入数据的有效性：

ALTER TABLE mytable

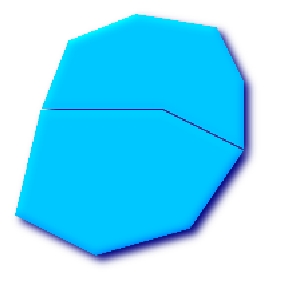
ADD CONSTRAINT geometry\_valid\_check

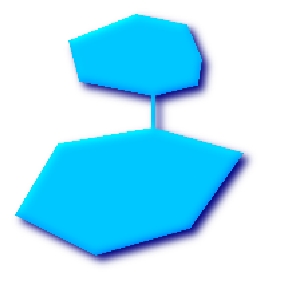
CHECK (ST\_IsValid(the\_geom));

如果你在调用PostGIS函数处理有效的几何类型数据时候，遇到一个奇怪的错误，比如"GEOS Intersection() threw an error!" 或 "JTS Intersection() threw an error!"，你可能已经发现了PostGIS中的一个错误，或者是PostGIS所依赖库的错误，这时候你应该联系PostGIS开发者。同样如果PostGIS函数对于有效的输入，却返回一个无效的几何数据，你也应该联系PostGIS的开发者。

**注意**

严格遵守OGC规范的几何对象不包括Z或M 值。ST\_IsValid()函数不会把更高维度的几何对象认为是无效的！调用函数AddGeometryColumn()会添加一个检查几何对象的约束。因此在这里指定维度为2足够了







PostGIS 2.2.0dev 开发手册

45 / 686

**4.3.6**

**九交模型——Dimensionally Extended 9 Intersection Model (DE-9IM)**

译者注：这是一个九元组加维数扩展的空间拓扑关系数学描述框架

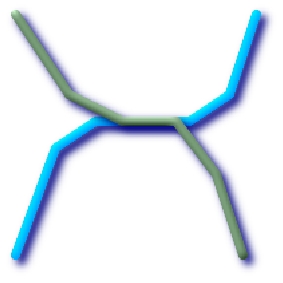
有时候典型的空间谓词比如(ST\_Contains, ST\_Crosses, ST\_Intersects, ST\_Touches, ...)不足以提供足够的空间过滤条件。

译者注：空间谓词是指比较两个空间对象关系并返回一个布尔值做为结果，它表明了两个空间对象之间关系。如：是否相交、是否互相包含等等。OGC提出了Disjoint/Intersect、Equals、Within/Contains、Overlaps、BBOX等多种空间谓词过滤器（其实就是各种空间关系函数啦）。

例如，考虑一个表示道路网络的线性数据集。GIS分析师也许会有这样的任务：区分所有相交于一条线，而不是一个点的路段，这可能表示一些业务上的规则。这种情况下函数ST\_Crosses 不能完全提供必要的空间过滤，因为对线性对象，只有他们相交于一个点的时候，它才会返回true。

有一个两步解决的方案可能是：首先使用函数(ST\_Intersection)对一对有空间交互(ST\_Intersects函数来判定)路段进行实际的相交，然后把交集的几何类型（使用函数ST\_GeometryType来获得）与类型’LINESTRING’ 进行比较(需要正确处理交集返回 [MULTI]POINTs, [MULTI]LINESTRING这样的GEOMETRY COLLECTION情况.).

我们需要一个更优雅、更快速的解决方案。



PostGIS 2.2.0dev 开发手册

46 / 686

第二个(理论性的)例子是一个GIS分析师试图找到所有与湖的边界相交于一条线的码头并且码头的一端在岸上。换句话说，一个码头在湖里面，但不完全在一个湖里面，和湖的边界相交于一条线，码头两端都完全在湖的边界上。分析师可能需要使用空间谓词的联合使用来分离查找结果：

· ST\_Contains(lake, wharf) = TRUE

· ST\_ContainsProperly(lake, wharf) = FALSE

· ST\_GeometryType(ST\_Intersection(wharf, lake)) = ’LINESTRING’

· ST\_NumGeometries(ST\_Multi(ST\_Intersection(ST\_Boundary(wharf), ST\_Boundary(lake)))) = 1

... (无需多说，这个非常复杂)

请参考Dimensionally Extended 9 Intersection Model（ DE-9IM ）

**4.3.6.1**

**理论知识**

根据OpenGIS Simple Features Implementation Speciﬁcation for SQL规范“比较两个几何对象关系的基本方法是对这两个几何对象的内部与内部之间、边界与边界之间、外部与外部之间进行相交试验，而区分两个几何对象之间的关系要根据相交矩阵来区分。”

Boundary

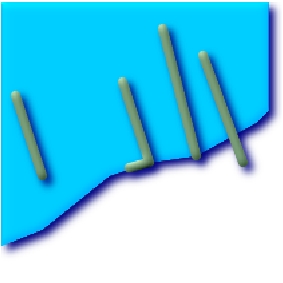
geometry的The boundary of a geometry is the set of geometries of the next lower dimension.对于Point对象，维度为0，其边界也是空集。LINESTRING的边界是两个端点。对于POLYGON对象，其边界是线条类对象，这些线条组成了POLYGON的内环和外环。

Interior

一个几何对象的内部是指移除几何对象边界后几何对象左边的点，称之为内部。对于点来说，点的内部就是点自身。对于LINESTRING来说，其内部就是起点和终点之间的真实点。对于POLYGON多边形来说，其内部就是多边形内部的面。

Exterior

几何对象的外部是一个宇宙，都是同一个面，不是几何对象的内部也不在几何对象的边界上。



PostGIS 2.2.0dev 开发手册

47 / 686

给定输入几何对象a， I(a), B(a), and E(a) 分别表示a的内部、边界、外部，那么a和b的相交矩阵如下：

Dim(a)是函数ST\_Dimension 计算得到的a的维度，相交结果为{0,1,2,T,F,\*}

· 0 => 点

· 1 => 线

· 2 => 面

· T => {0,1,2}，表示上面1,2,3都有

· F => 空集

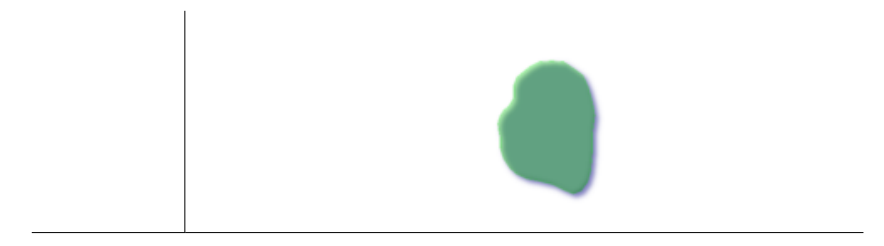
· \* => 不考虑

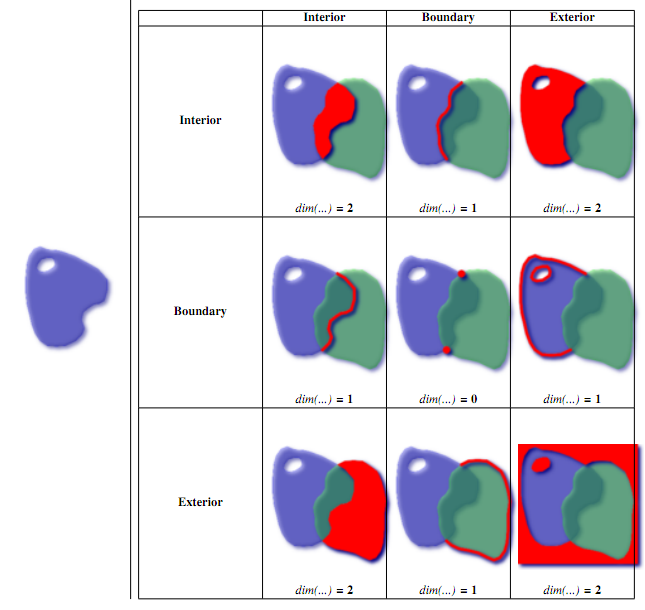
直观上看，对于两个叠加的多边形对象，看起来是这样：

|  |  |  |  |
| --- | --- | --- | --- |
|  | Interior | Boundary | Exterior |
| Interior | dim( I(a) ∩ I(b) ) | dim( I(a) ∩ B(b) ) | dim( I(a) ∩ E(b) ) |
| Boundary | dim( B(a) ∩ I(b) ) | dim( B(a) ∩ B(b) ) | dim( B(a) ∩ E(b) ) |
| Exterior | dim( E(a) ∩ I(b) ) | dim( E(a) ∩ B(b) ) | dim( E(a) ∩ E(b) ) |

PostGIS 2.2.0dev 开发手册

48 / 686





从左到右读，从上到下读，维度矩阵表示为：’212101212’.

我们第一个例子中，表示两条线相交于一条线这样的关系矩阵是： ’1\*1\*\*\*1\*\*’

-- Identify road segments that cross on a line

SELECT a.id

FROM roads a, roads b

WHERE a.id != b.id

AND a.geom && b.geom

AND ST\_Relate(a.geom, b.geom, ’1\*1\*\*\*1\*\*’);

表示码头部分地处于湖的海岸线上这样的关系矩阵是：’102101FF2’

-- Identify wharfs partly on a lake’s shoreline

SELECT a.lake\_id, b.wharf\_id

FROM lakes a, wharfs b

WHERE a.geom && b.geom

AND ST\_Relate(a.geom, b.geom, ’102101FF2’);

PostGIS 2.2.0dev 开发手册

49 / 686

更多信息，请参考：

· OpenGIS Simple Features Implementation Speciﬁcation for SQL (1.1版本规范，2.1.13.2节)

· [Dimensionally Extended Nine-Intersection Model (DE-9IM) by Christian Strobl](http://gis.hsr.ch/wiki/images/3/3d/9dem_springer.pdf)

· [GeoTools: Point Set Theory and the DE-9IM Matrix](http://docs.geotools.org/latest/userguide/library/jts/dim9.html)

· Encyclopedia of GIS By Hui Xiong

**4.4**

**加载 GIS (矢量) 数据**

一旦你创建了空间表，你就做好了将GIS数据上传到数据库的准备。目前，有两种版本把数据弄到PostGIS/PostgreSQL 数据库中：使用格式化的SQL语句或使用shp文件加载/导出工具

**4.4.1**

**使用SQL加载数据**

如果你能把你的数据转换成text文本方式描述，那么把数据加载到数据库的最简单办法就是使用SQL的方式。就像Oracle和其他支持SQL的数据库一样，可以把以SQL "INSERT"语句方式表示的语句，在数据库终端中执行，实现大批量的数据插入。以一个上传好的数据文件为例（比如road.sql这个文件），文件内容可能是这样的：

BEGIN;

INSERT INTO roads (road\_id, roads\_geom, road\_name)

VALUES (1,ST\_GeomFromText(’LINESTRING(191232 243118,191108

INSERT INTO roads (road\_id, roads\_geom, road\_name)

VALUES (2,ST\_GeomFromText(’LINESTRING(189141 244158,189265

INSERT INTO roads (road\_id, roads\_geom, road\_name)

VALUES (3,ST\_GeomFromText(’LINESTRING(192783 228138,192612

INSERT INTO roads (road\_id, roads\_geom, road\_name)

VALUES (4,ST\_GeomFromText(’LINESTRING(189412 252431,189631

INSERT INTO roads (road\_id, roads\_geom, road\_name)

VALUES (5,ST\_GeomFromText(’LINESTRING(190131 224148,190871

INSERT INTO roads (road\_id, roads\_geom, road\_name)

VALUES (6,ST\_GeomFromText(’LINESTRING(198231 263418,198213

243242)’,-1),’Jeff Rd’);

244817)’,-1),’Geordie Rd’);

229814)’,-1),’Paul St’);

259122)’,-1),’Graeme Ave’);

228134)’,-1),’Phil Tce’);

268322)’,-1),’Dave Cres’);

COMMIT;

使用psql命令，数据库可以很容易批量导入到PostgreSQL中：

psql -d [database] -f roads.sql

**4.4.2**

**shp2pgsql: 使用ESRI Shapeﬁle文件加载工具**

译者注：这实际上是一个转换工具，它把shp文件转换成sql文件，然后需要psql命令将数据插入进去。

shp2pgsql 工具把ESRI Shape 文件转化成SQL语句，以便用于插入到PostGIS/PostgreSQL 数据库中，shp文件可以是包含geometry类型数据，也可以是包含geography类型的数据。该工具有几种操作模式，以不同的命令行参数来区分。

除了shp2pgsql命令行方式，还有一个shp2pgsql-gui图形化界面的方式，支持大多数命令行模式，但却是一个不需要脚本加载的更容易的方式，或者你是一个PostGIS新手，这样也是更简单方法。这个也可以当做PgAdminIII的一个插件使用。

(c|a|d|p) 这些参数是互相排斥的：

-c 创建一个新表，并将shp数据插入进去。这是默认的模式

-a 将shp文件数据以增量方式插入到数据表中。注意：如果要使用这种模式加载许多个文件包含的数据，这些文件必须有相同的列和相同的数据类型。

-d 创建存放shp文件数据的表前，删除掉该表。

PostGIS 2.2.0dev 开发手册

50 / 686

-p 只生成创建表的SQL语句，不用加载实际的数据。如果你需要完全把创建表和数据加载两步分开，那可以使用这个模式。

-? 展示帮助内容

-D ：使用PostgreSQL "dump" 格式输出数据。该参数可以和-a, -c 或 -d其中之一联合使用。它比通过INSERT方式插入数据要快得多。对于海量数据，可以使用这个参数。

-s [<FROM\_SRID%gt;:]<SRID> ：根据指定的SRID值，创建并在表中填充数据。你可以根据给定的FROM \_SRID值，指定任意shp格式文件，这样几何数据会被重投影到目的SRID值。

FROM\_SRID 值不能用-D参数来指定

-k：保持标识符名称的大小写格式(比如列, schema 和字段). 注意：shp文件里面的字段名称都是大写的。

-i ：强制将所有的整型值转换为32位的整型值，带上该参数，则不会创建64位的值，即便.dbf文件的签名头里面注明了这一点。

-I 在一个几何数据列上创建GiST索引

-S ：生成简单几何对象，而不是MULTI 几何数据。如果所有的几何对象真的都是简单的（比如MULTIPOLYGON 带有一个简单的环或者MULTIPOINT有个简单的点），那么才会成功生成简单几何对象。

-t <关于维度的参数>：强制将输出的几何对象加上指定的维度。使用如下字符串值来表明指定的维度类型： 2D, 3DZ, 3DM, 4D

如果输入的维度数没有指定的多，输出时候这些多加的维度坐标会用0填充。.如果输入的数据维度比指定的多，那么不在指定范围的维度将会被丢弃。

-w ：输出WKT格式描述的数据，而不是WKB的。注意：该参数可能由于精度损失导致坐标有些细微变化。

-e：不使用一个大事务提交数据，而是对于每执行一条插入语句就提交一次。对于一个SQL格式的数据文件来说，如果这个文件里面包含主要的都是“好的数据”并且有一些会导致的错误的“坏的数据”，那么插入数据时候可以将好的数据成功插入进去。（译者注：如果所有的插入语句都在一个事务中的话，一旦出现一条语句失败，整个事务都无法提交，导致数据无法插入进去）。注意：该参数不能和-D参数一起使用，因为-D参数使用的dump格式数据总是使用一个大事务来插入数据。

-W <数据文件编码>：指定输入数据文件（就是dbf后缀的文件）的编码格式。使用时候，所有的dbf数据列都会从指定的文件编码转换成UTF-8格式的。输出的SQL数据文件脚本会包含一条命令：SET CLIENT\_ENCODING to UTF8，这样后台就可以将数据从UTF-8转换成内部使用的任意数据库编码。

-N <NULL处理策略>：插入几何数据列值为NULL记录时候的处理策略，有跳过或者终止两种

-n ：-n参数只导入DBF格式的文件。如果你数据没有对应的shp格式文件（译者注：一般提供数据文件时候，会提供shp和dbf两种格式），那么这个命令hi自动切换到这个模式，并加载dbf格式的文件数据。因此设置这个参数意味着，虽然你有shp文件数据集，但你只想要属性数据而不需要几何特征数据。（译者注：shapefile文件属于矢量数据中无拓扑关系的数据类型。它至少包含3个文件，dbf、shp、shx 。其中：dbf文件时存储矢量数据的属性信息的；shp是存储矢量数据的空间信息的。shx是保留要素几何学特征的索引文件。）

-G ：在WGS84（SRID=4326）参考系中使用地理数据类型而不是几何数据类型。

-T <表空间参数>：指定新创建表的表空间。除非使用了-X参数，否则索引也会使用默认的表空间。PostgreSQL 文档有一个很好的关于何时使用自定义表空间的说明。

-X <表空间参数>：指定新创建表索引的表空间。这个参数适用于primary key索引（译者注：在PostgreSQL中primary key也可以看做一种索引，读者可以在where条件中加上primary key条件，然后查看执行计划），如果指定了-I参数，那么这个参数也适用于GiST空间索引。

使用shp2pgsql命令来创建一个输入数据文件并导入到数据库的步骤大致如下：

# shp2pgsql -c -D -s 4269 -i -I shaperoads.shp myschema.roadstable > roads.sql

# psql -d roadsdb -f roads.sql

使用UNIX管道功能可以在一个命令中完成shp文件的转化和数据导入：

# shp2pgsql shaperoads.shp myschema.roadstable | psql -d roadsdb

PostGIS 2.2.0dev 开发手册

51 / 686

**4.5**

**获取GIS数据**

无论使用SQL还是shp文件加载/导出工具，都可以把数据从数据库中抽取出来。在关于使用SQL方式的部分，我们会讨论一些可用的操作符来做bounding box和在空间表上的查询

**4.5.1**

**使用SQL获取数据**

从数据库中获取数据的最直接的方法是使用SQL语句的select查询方式，来减少查询的记录数和列的数量，然后把数据结果列存储在一个可解析的文本中。

db=# SELECT road\_id, ST\_AsText(road\_geom) AS geom, road\_name FROM roads;

road\_id | geom

| road\_name

--------+-----------------------------------------+-----------

1 | LINESTRING(191232 243118,191108 243242) | Jeff Rd

2 | LINESTRING(189141 244158,189265 244817) | Geordie Rd

3 | LINESTRING(192783 228138,192612 229814) | Paul St

4 | LINESTRING(189412 252431,189631 259122) | Graeme Ave

5 | LINESTRING(190131 224148,190871 228134) | Phil Tce

6 | LINESTRING(198231 263418,198213 268322) | Dave Cres

7 | LINESTRING(218421 284121,224123 241231) | Chris Way

(6 rows)

然而会有时候需要一些限制来减少返回的记录数。如果是属性数据列（就是非空间类型数据）的限制，直接正常使用SQL语法来查询非空间表即可。如果是空间数据限制，下面的运算符可以使用：

&& ：这个操作符能够区分两个geometry对象的bounding box是否相交

ST\_OrderingEquals ：这个函数测试两个几何对象是否几何上相同。例如测试’POLYGON((0 0,1 1,1 0,0 0))’ 是否和 ’POLYGON((0 0,1 1,1 0,0 0))’相同（的确相同）

= ：该运算符有点太简单了，它只是检测两个几何对象的bounding box是否是相同的

现在你可以在查询中使用这些运算符。注意：当在SQL语句中指定了具体的几何对象或者box时候，你必须显式地通过使用函数ST\_GeomFromText()来把字符串描述的格式转换成实际的几何对象。SRID为312是匹配我们数据的参考SRID。例如：

SELECT road\_id, road\_name

FROM roads

WHERE ST\_OrderingEquals(roads\_geom , ST\_GeomFromText(’LINESTRING(191232 243118,191108

←

243242)’,312) ) ;

上面的查询会从roads表中返回匹配几何对象和给定的LINESTRING(191232 243118,191108 243242)一样的几何对象。运算符"&&"支持BOX3D或者GEOMETRY类型对象。当你使用的是GEOMETRY类型对象时候，它的bounding box回用于索引比较：

SELECT road\_id, road\_name

FROM roads

WHERE roads\_geom && ST\_GeomFromText(’POLYGON((...))’,312);

上面的查询会使用多边形的bounding box来进行比对以便加快查询

最常见的空间查询可能会是一个“框架”的查询，通过使用客户端软件的使用，如数据浏览器和Web地图工具，抓住带有数据的“图框”显示。需要用“BOX3D”对象表示该框架，查询语句如下：

SELECT ST\_AsText(roads\_geom) AS geom

FROM roads

WHERE

roads\_geom && ST\_MakeEnvelope(191232, 243117,191232, 243119,312);

注意：SRID值为312是为了指定凸包的投影系

PostGIS 2.2.0dev 开发手册

52 / 686

**4.5.2**

**使用批量导出工具（Dump数据工具）**

pgsql2shp是一个直接连接数据库然后导出表数据的工具，它会将表（也可以是一个查询）导出成Shape文件。基本用法是

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

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

这些命令参数是：

-f <ﬁlename>：将导出结果输出到指定文件

-h <host> ：数据库IP

-p <port> ：数据库监听端口

-P <password> ：连接数据库所需要的密码

-u <user>：连接数据库所需的用户名

-g <geometry column> ：指定写入到shape 文件中时候的列名

-b ：使用二进制游标，这会让导出更快，但是如果任何一个非几何类型的列没有转换成text格式，那么该参数无效

-r ：裸模式，不删除gid字段或者转义列名称。

-d ：为了后向兼容性，从老版本PostGIS（1.0.0版本之前）导成3D维度shape文件，以前的老版本默认输出2D对象文件。从PostGIS 1.0.0版本开始，会默认导出成3D格式。

-mfilename，重新把标识符映射为一个10个字符的名称。. 文件的行是由被单个空格分隔的两个符号组成，行首或者行尾空格。这些符号包括： VERYLONGSYMBOL SHORTONE ANOTHERVERYLONGSYMBOL SHORTER 等等。（译者注：不知道这个参数干嘛用的）

**4.6**

**创建索引**

索引让空间数据库的大量数据查询成为可能，没有索引，任何特性的查询都需要进行全表扫描。索引通过将数据丢到一个遍历树中以便快速找到一个指定的记录。PostgreSQL 默认支持三种索引类型：B-Tree索引，R-Tree索引和GiST索引：

· B-Tree索引：这种索引常用于可以在一个方向上排序的数据，比如数值，字母，日期，GIS数据不能在一个方向上进行排序（比如(0,0) ， (0,1) ， (1,0)三个对象哪个更大呢？) ，因此B-Tree索引对GIS数据无用。

· R-Trees 索引：这种索引把数据分隔成矩形，子矩形（sub-rectangles）和子子矩形（sub-sub rectangles，译者注：有时候这样的术语还不如不翻译）等等,。R-Tree索引适用于部分空间数类型，但是PostgreSQL的R-Tree索引实现不如GiST索引稳健

· GiST (Generalized Search Trees缩写)索引把数据分成这样几个部分：某一边的对象，重叠的对象，内部的对象，该索引适用范围很广，包括GIS数据。PostGIS使用的R-Tree索引是基于GiST实现的。(译者注：原文如此，找了公开资料也是这么说的，不过很纳闷既然R-Tree索引是基于GiST数据结构实现，为毛R-Tree索引还会出现不支持8K数据问题？估计GiST索引和GiST数据结构是两回事，R-Tree索引是基于GiST数据结构实现的，但不支持8K大的对象，而GiST索引却可以)

PostGIS 2.2.0dev 开发手册

53 / 686

**4.6.1**

**GiST 索引**

GiST 表示"Generalized Search Tree" ，是索引的一种常规形式。除了支持GIS索引外，GiST索引还可以加速所有不规则数据结构（整型数组，谱数据等等）的查询，而B-Tree索引在这样的情形却不怎么靠谱。

一旦一个GIS数据表有数千行记录，你应当创建一个索引来加速空间数据查询（除非你的所有查询都是在属性列（非几何类型列）上，这样的话你可以在属性列上面创建一个正常的索引）。

在一个geometry类型的列上面创建一个GiST索引方法如下：

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

上面的语法是用于创建2D索引的，要想创建一个PostGIS 2.0+版本支持的N维几何对象索引，方法如下：

CREATE INDEX [indexname] ON [tablename] USING GIST ([geometryfield] gist\_geometry\_ops\_nd);

创建空间索引是一种密集计算的工作：一个有百万记录的表，CPU为300MHz的Solaris机器上，创建GIS索引我们发现需要1个小时。创建完索引后，必须强制让PostgreSQL更新表的统计信息，这些统计信息可以用于查询计划的优化，更新方法如下：

VACUUM ANALYZE [table\_name] [(column\_name)];

-- This is only needed for PostgreSQL 7.4 installations and below

SELECT UPDATE\_GEOMETRY\_STATS([table\_name], [column\_name]);

GiST 索引相比较PostgreSQL中的R-Tree所有有两个优势。首先，GiST索引是NULL值安全的，这意味着用户可以在一个包含NULL值的列（包括几何类型列）上面创建索引。第二，GiST索引支持“无丢失”存储，这对于处理超过8K存储大小的大型GIS对象来说是很重要的。这种“无丢失”特性能够让PostgreSQL在索引中只存储一个对象的关键部分——比如对于GIS对象，只存储它的bounding box。在创建R-Tree索引的过程中，超过8K的GIS对象会引起创建失败。

**4.6.2 使用索引**

一般来说，索引能够隐式地加速获取数据：一旦创建了索引，查询优化器会决定什么时候使用索引信息来加速查询。不幸地是PostgreSQL不能尽可能有效地像使用其他索引一样使用GiST索引，这会导致有时候应该使用空间索引，却使用了全表扫描。

如果你发现空间索引(就这里来说就是字段索引)没有很好的使用，你可以按照下面的方式来做：

首先，确保表的记录数和值分布这样的统计信息已经被收集，以便查询优化器有更丰富的信息来决定是否使用索引。对于PostgreSQL 7.4及以下的版本来说，收集统计信息可以使用函数update\_geometry\_stats([table\_name, column\_name]) (compute distribution) 和命令 VACUUM ANALYZE [table\_name] [column\_name] (compute number of values)来做。从PostgreSQL 8.0 版本开始，执行命令VACUUM ANALYZE会将上面两个操作一步完成。你应该定期VACUUM你的数据库——许多PostgreSQL的DBA会经常定期执行VACUUM任务。

如果VACUUM操作没有解决问题，你可以强制通过SET ENABLE\_SEQSCAN=OFF 把全表扫描关了来使用索引。你应该有节制地使用这个命令，并且只在空间索引查询上使用。一般来说，查询优化器会比你在何时使用B-Tree索引上更懂。一旦你完成了查询，你应该把SET ENABLE\_SEQSCAN值改回ON，以便其他查询可以正常使用查询优化器。

**注意**

postgis 0.6版本不必把全表扫描关掉

·

如果你发现查询优化器对全表扫描和索引扫描的成本消耗估算是错误的，你可以减少配置文件中postgresql.conf 的random\_page\_cost 参数值，或者使用SET random\_page\_cost=#语句设置。改值的默认值为4，可以将这个值设置为1或2.减少这个值可以让查询优化器更倾向于使用索引。



PostGIS 2.2.0dev 开发手册

54 / 686

**4.7**

**复杂查询**

空间数据库功能必要的理由是因为需要在数据库内部进行查询，而这些是客户端GIS功能所需要的。要想高效使用PostGIS，你需要知道哪些空间函数是可用的，并且确保在空间表中有合适的索引以便提供好的查询性能。下面的这些样例中SRID值为312是用来展示样例的，实际开发中应该使用在spatial\_ref\_sys中真实的和你的数据匹配的SRID值。如果你的数据中没有指定空间参考系，你应该好好想想为毛这个数据没有SRID值，它应该有才对！

如果没有SRID的原因是你正在构建一个没有地理空间参考系的应用，比如分子内部建模或者在发生核灾难时候要把人类运送到火星的一个好位置上（译者注：这个开发可真会扯淡），那么省略掉SRID值吧，自己造一个参考系，然后把它插入到spatial\_ref\_sys 表中.

**4.7.1 充分利用索引**

构建查询时候，记住只有基于bounding box的运算符比如&&才能利用GiST索引。一些函数像ST\_Distance() 不能使用索引来优化查询。例如下面的在大表上的查询是非常慢的：

SELECT the\_geom

FROM geom\_table

WHERE ST\_Distance(the\_geom, ST\_GeomFromText(’POINT(100000 200000)’, 312)) < 100

这个查询查询出所有在点point (100000, 200000) 100个长度单位范围的所有记录。这个查询非常慢的原因是，它会计算出我们给定的点与表中的所有点的距离。我们可以通过&&运算符来减少所需的距离计算量。

SELECT the\_geom

FROM geom\_table

WHERE ST\_DWithin(the\_geom, ST\_MakeEnvelope(90900, 190900, 100100, 200100,312), 100)

这个查询同样结果的几何对象，但是这个查询会更高效。假定在字段the\_geom上面有一个GiST索引，查询优化器会识别到这个索引，并用来减少使用函数ST\_Distance进行计算的记录数。注意运算符&&隐含使用函数ST\_MakeEnvelope返回的几何对象是一个以给定点为原点的长宽为200个长度单位的正方形盒子——这个是我们的“查询盒子”。运算符&&使用索引来把要查询的几何对象集减少到其bounding box和我们查询盒子有叠加的几何对象。假定我们的“查询盒子”要比整个表小很多，那么这个办法会大幅度减少距离的计算量。

**变更的方式Change in Behavior**

从PostGIS 1.3.0版本起，大部分几何关系处理函数, 除了函数ST\_Disjoint 和ST\_Relate外，都隐含地包含了bounding box的叠加运算方式

**4.7.2 空间SQL样例**

本节所用的样例会使用两张表，一个是线性道路表，一个是多边形的市政边界。道路表bc\_roads的定义如下：

Column

| Type

| Description

------------+-------------------+-------------------

gid

| integer

| Unique ID

name

| character varying | Road Name

the\_geom

| geometry

| Location Geometry (Linestring)

市政边界表定义如下bc\_municipality:



PostGIS 2.2.0dev 开发手册

55 / 686

Column

| Type

| Description

-----------+-------------------+-------------------

gid

code

| integer

| integer

| Unique ID

| Unique ID

name

| character varying | City / Town Name

the\_geom

| geometry

| Location Geometry (Polygon)

1. 所有的道路以千米为长度单位，总长度是多少？

这个问题答案很简单，一个SQL就OK:

SELECT sum(ST\_Length(the\_geom))/1000 AS km\_roads FROM bc\_roads;

km\_roads

------------------

70842.1243039643

(1 row)

2. Prince George市有多少公顷？

该查询需要把查询时候的条件（该条件在bc\_municipality的字段name上）和一个空间计算结（面积计算）合起来：

SELECT

ST\_Area(the\_geom)/10000 AS hectares

FROM bc\_municipality

WHERE name = ’PRINCE GEORGE’;

hectares

------------------

32657.9103824927

(1 row)

3. 该省最大面积的城市是哪个？

该查询需要在查询条件中加入空间测量计算。有许多方法解决这个问题，但是最高效的是下面的方法：

SELECT

name,

ST\_Area(the\_geom)/10000 AS hectares

FROM

bc\_municipality

ORDER BY hectares DESC

LIMIT 1;

name

| hectares

---------------+-----------------

TUMBLER RIDGE

| 155020.02556131

(1 row)

注意：为了解决这个查询，我们必须计算每一个多边形的面积。如果要计算的面积对象很多，很重要的一点是要在要计算面积的列上面单独加上索引以便提高性能。通过逆序排序结果，然后使用PostgreSQL的LIMIT子句，我们可以很容易地找出最大的面积值，而不必使用像max（）这样的聚合函数

4. 完全包含在每个直辖市里面的道路长度分别是多少？

这是一个“空间join”的类型，因为我们把两个表的数据（做空间join）放到一起，但是使用空间相交的条件（空间包含）而不是通常在一个字段上进行的关联join：

PostGIS 2.2.0dev 开发手册

56 / 686

SELECT

m.name,

sum(ST\_Length(r.the\_geom))/1000 as roads\_km

FROM

bc\_roads AS r,

bc\_municipality AS m

WHERE

ST\_Contains(m.the\_geom,r.the\_geom)

GROUP BY m.name

ORDER BY roads\_km;

name

| roads\_km

----------------------------+------------------

SURREY

VANCOUVER

LANGLEY DISTRICT

BURNABY

PRINCE GEORGE

| 1539.47553551242

| 1450.33093486576

| 833.793392535662

| 773.769091404338

| 694.37554369147

...

这个查询会花费一点时间，因为表中的每一个道路的长度都要加起来（聚合）到最终的结果（对于我们的这个表大概有25W条记录）。对于较小的覆盖（几千条记录覆盖后只有几百条），查询响应会非常快。

5. 创建城市Prince George内所有道路的一个新表

这是一个叠加的样例，它使用两张表，输出一张新表，该新表是这两个表包含的几何对象空间切割的产物，和上面的空间join是不同的，这个查询实际上创造了新的几何对象。叠加可以看做是空间join的增强版，对于精确数据分析很有用。

CREATE TABLE pg\_roads as

SELECT

ST\_Intersection(r.the\_geom, m.the\_geom) AS intersection\_geom,

ST\_Length(r.the\_geom) AS rd\_orig\_length,

r.\*

FROM

bc\_roads AS r,

bc\_municipality AS m

WHERE

m.name = ’PRINCE GEORGE’ AND ST\_Intersects(r.the\_geom, m.the\_geom);

6. Victoria市的"Douglas St"大街长度是多少？

SELECT

sum(ST\_Length(r.the\_geom))/1000 AS kilometers

FROM

bc\_roads r,

bc\_municipality m

WHERE

r.name = ’Douglas St’ AND m.name = ’VICTORIA’

AND ST\_Contains(m.the\_geom, r.the\_geom) ;

kilometers

------------------

4.89151904172838

(1 row)

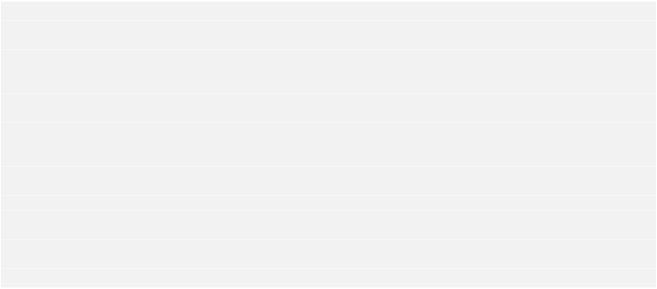
7. bc\_municipality表中含有孔的面积最大的多边形是什么？

SELECT gid, name, ST\_Area(the\_geom) AS area

FROM bc\_municipality

WHERE ST\_NRings(the\_geom) > 1

ORDER BY area DESC LIMIT 1;



PostGIS 2.2.0dev 开发手册

57 / 686

gid

| name

| area

-----+--------------+------------------

12 | SPALLUMCHEEN | 257374619.430216

(1 row)