

Trabalhando com pontos e linhas

De Juliano Ignácio

Não, não é nenhum artigo sobre corte e costura, ou ainda, sobre desenhos (como vemos no Paint do Windows). Trata-se de tipos de dados que constam no PostgreSQL específicos para operações geométricas.

O PostgreSQL tem condições de ser utilizado para construir poderosas soluções que exijam tratamento científico, onde, tais recursos são dificilmente encontrados em bancos de dados comerciais (Oracle e DB2 têm esses recursos). Este artigo dará uma pequena idéia de como tratar dados geográficos (Geo-Data) com o PostgreSQL, um assunto cada vez mais importante: o geoprocessamento (GIS).

Dados geométricos podem ser armazenados com o auxílio de alguns tipos de dados do PostgreSQL: point, line, box, path, polygon e circle.

point

O ponto é o ponto fundamental (desculpem, não resisti ao trocadilho) para o tratamento de objetos geométricos. Ele pode ser manuseado fácil e eficientemente pelo usuário. Um ponto (neste caso) é definido por dois valores: o primeiro é o valor da coordenada do eixo X, o segundo é o valor da coordenada do eixo Y. Ambos os valores são armazenados internamente no banco de dados PostgreSQL como valores do tipo ponto-flutuante (8 bytes), portanto, um ponto requer 16 bytes de armazenamento.

<pre>CREATE TABLE tabpontos(posicao point); INSERT INTO tabpontos(posicao) VALUES ('1,2'); INSERT INTO tabpontos(posicao) VALUES ('1,3');</pre>	<pre>SELECT * FROM tabpontos; posicao ----- (1,2) (1,3) (2 rows)</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------

Um detalhe muito importante é que ao inserirmos os pontos, estes devem estar entre apóstrofes, caso contrário, um erro sobre tipos diferentes de dados será informado.

line

Uma linha é definida através de dois pontos (ou seja, quatro valores ou dois pares de coordenadas). O primeiro ponto determina o início da linha e, o segundo, determina seu término. Portanto, para armazenar uma linha serão utilizados 32 bytes. Existem dois tipos definidos para linha: line e lseg. O tipo line ainda não está totalmente implementado, usaremos então o lseg.

<pre>CREATE TABLE tablinhas(segmento lseg); INSERT INTO tablinhas(segmento) VALUES ('1,4,3,5'); INSERT INTO tablinhas(segmento) VALUES ('((1,4),(3,5))'); INSERT INTO tablinhas(segmento) VALUES ('[(2,6),(4,7)]');</pre>	<pre>SELECT * FROM tablinhas; segmento ----- [(1,4),(3,5)] [(1,4),(3,5)] [(2,6),(4,7)] (3 rows)</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------

A utilização de colchetes e parênteses facilita a leitura ao inserir os pontos da linha.

box

O tipo box é utilizado para armazenar um retângulo (ou quadrado), definido por dois pontos (exatamente como a linha), porém, esta linha, define a diagonal deste retângulo. Portanto, o espaço destinado ao armazenamento de um retângulo é igual ao de uma linha, 32 bytes.

<pre>CREATE TABLE tabarearec(area box); INSERT INTO tabarearec(area) VALUES ('7,5,2,3'); INSERT INTO tabarearec(area) VALUES ('((1,1),(8,0))');</pre>	<pre>SELECT * FROM tabarearec; area ----- (7,5),(2,3) (8,1),(1,0) (2 rows)</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------

Aqui, somente a utilização de parênteses é permitida, os colchetes não.

path

O **path** é um caminho, ou seja, uma sequência de pontos que pode ser aberta ou fechada. Fechada significa que o último ponto do caminho retorna ao início da sequência. O tamanho (ou espaço de armazenamento) de um **path** é dinâmico, ou seja, são utilizados 4 bytes de início e 32 bytes para cada nó que for armazenado. O PostgreSQL também fornece algumas funções especiais para verificar se o caminho é aberto ou fechado: `popen()` e `pclose()` forçam que o caminho deva ser aberto ou fechado respectivamente; `isopen()` e `isclosed()` fazem a verificação.

<pre>CREATE TABLE tabtrilha(trajeto path); INSERT INTO tabtrilha(trajeto) VALUES ('(1,2), (5,3)'); INSERT INTO tabtrilha(trajeto) VALUES ('(1,3), (-3,1), (7,0)');</pre>	<pre>SELECT * FROM tabtrilha; trajeto ----- ((1,2),(5,3)) ((1,3),(-3,1),(7,0)) (2 rows)</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------

polygon

O polígono é um caminho fechado por definição.

<pre>CREATE TABLE tabarea(area polygon); INSERT INTO tabarea(area) VALUES ('(1,3), (4,16), (0,23)'); INSERT INTO tabarea(area) VALUES ('(2,7), (5,-15), (1,20)');</pre>	<pre>SELECT * FROM tabarea; area ----- ((1,3),(4,16),(0,23)) ((2,7),(5,-15),(1,20)) (2 rows)</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------

circle

Um círculo consiste de um ponto central e seu raio, e precisa de 24 bytes para ser armazenado.

<pre>CREATE TABLE tabareacirc(area circle); INSERT INTO tabareacirc(area) VALUES ('10,12,10'); INSERT INTO tabareacirc(area) VALUES ('12,19,5');</pre>	<pre>SELECT * FROM tabareacirc; area ----- <(10,12),10> <(12,19),5> <(2,9),0> (3 rows)</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------

O tipo **circle** não aceita o uso de parênteses para uma melhor definição visual e, além disso, o valor do raio SEMPRE deve ser maior ou igual a zero.

OPERADORES PARA DADOS GEOMÉTRICOS

Primeira regra: lembre-se de usar o operador `~=` no lugar de `=` sempre, caso contrário receberá uma mensagem de erro. Por exemplo:

```
SELECT * FROM tabpontos WHERE posicao ~= '(1,2)';
```

Segundo: existem operadores para cálculo entre dados geométricos. Por exemplo:

Distância entre 2 pontos	<pre>SELECT '(1,1)::point <-> '(3,9)::point;</pre>
Adição de 2 pontos	<pre>SELECT '(2,1)::point + '(4,8)::point;</pre>
Subtração de 2 pontos	<pre>SELECT '(3,1)::point - '(5,7)::point;</pre>
Multiplicação de 2 pontos	<pre>SELECT '(4,1)::point * '(6,6)::point;</pre>
Divisão de 2 pontos	<pre>SELECT '(5,1)::point / '(7,5)::point;</pre>
Divisão de 2 pontos	<pre>SELECT '(5,1)::point / '(7,5)::point;</pre>
Intersecção de 2 segmentos	<pre>SELECT '((0,0),(10,1))::lseg # '((-4,5),(10,-3))::lseg;</pre>
O ponto mais próximo	<pre>SELECT '(10,1)::point ## '((-4,5),(10,-3))::lseg;</pre>

Terceiro: existem operadores para análise dos pontos. Por exemplo:

Pontos perpendiculares na horizontal	<pre>SELECT '(6,1)::point ?- '(8,4)::point;</pre>
Pontos perpendiculares na vertical	<pre>SELECT '(6,1)::point ? '(8,4)::point;</pre>
O segundo ponto está à esquerda do primeiro	<pre>SELECT '(-8,0)::point << '(0,0)::point;</pre>
O segundo ponto está abaixo do primeiro	<pre>SELECT '(0,-9)::point <^ '(0,0)::point;</pre>
O segundo ponto está à direita do primeiro	<pre>SELECT '(6,0)::point >> '(0,0)::point;</pre>
O segundo ponto está acima do primeiro	<pre>SELECT '(0,5)::point >^ '(0,0)::point;</pre>

Quarto: existem diversos outros operadores para trabalhar com dados geométricos (`&&`, `&<`, `&>`, `?`, `#`, `@-@`, `?||`, `@`, `@@`), procure maiores detalhes na documentação do PostgreSQL, ou execute `/do` em seu `psql` para ver todos os operadores.

Link do original:

http://imasters.uol.com.br/artigo/1004/postgresql/trabalhando_com_pontos_e_linhas/imprimir/

Tipos geométricos

Os tipos de dado geométricos representam objetos espaciais bidimensionais. A [Tabela 8-16](#) mostra os tipos geométricos disponíveis no PostgreSQL. O tipo mais fundamental, o ponto, forma a base para todos os outros tipos.

Tabela 8-16. Tipos geométricos

Nome	Tamanho de Armazenamento	Descrição	Representação
point	16 bytes	Ponto no plano	(x,y)
line	32 bytes	Linha infinita (não totalmente implementado)	((x1,y1),(x2,y2))
lseg	32 bytes	Segmento de linha finito	((x1,y1),(x2,y2))
box	32 bytes	Caixa retangular	((x1,y1),(x2,y2))
path	16+16n bytes	Caminho fechado (semelhante ao polígono)	((x1,y1),...)
path	16+16n bytes	Caminho aberto	[(x1,y1),...]
polygon	40+16n bytes	Polígono (semelhante ao caminho fechado)	((x1,y1),...)
circle	24 bytes	Círculo	<(x,y),r> (centro e raio)

Está disponível um amplo conjunto de funções e operadores para realizar várias operações geométricas, como escala, translação, rotação e determinar interseções, conforme explicadas na [Seção 9.10](#).

Pontos

Os pontos são os blocos de construção bidimensionais fundamentais para os tipos geométricos. Os valores do tipo point são especificados utilizando a seguinte sintaxe:

```
( x , y )  
x , y
```

onde x e y são as respectivas coordenadas na forma de números de ponto flutuante.

Segmentos de linha

Os segmentos de linha (lseg) são representados por pares de pontos. Os valores do tipo lseg são especificado utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )  
( x1 , y1 ) , ( x2 , y2 )  
x1 , y1 , x2 , y2
```

onde (x1,y1) e (x2,y2) são os pontos das extremidades do segmento de linha.

Caixas

As caixas são representadas por pares de pontos de vértices opostos da caixa. Os valores do tipo box são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )  
  ( x1 , y1 ) , ( x2 , y2 )  
    x1 , y1      , x2 , y2
```

onde (x1,y1) e (x2,y2) são quaisquer vértices opostos da caixa.

As caixas são mostradas utilizando a primeira sintaxe. Os vértices são reordenados na entrada para armazenar o vértice direito superior e, depois, o vértice esquerdo inferior. Podem ser especificados outros vértices da caixa, mas os vértices esquerdo inferior e direito superior são determinados a partir da entrada e armazenados.

Caminhos

Os caminhos são representados por listas de pontos conectados. Os caminhos podem ser *abertos*, onde o primeiro e o último ponto da lista não são considerados conectados, e *fechados*, onde o primeiro e o último ponto são considerados conectados.

Os valores do tipo path são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )  
[ ( x1 , y1 ) , ... , ( xn , yn ) ]  
  ( x1 , y1 ) , ... , ( xn , yn )  
  ( x1 , y1      , ... , xn , yn )  
    x1 , y1      , ... , xn , yn
```

onde os pontos são os pontos das extremidades dos segmentos de linha que compõem o caminho. Os colchetes ([]) indicam um caminho aberto, enquanto os parênteses (()) indicam um caminho fechado.

Os caminhos são mostrados utilizando a primeira sintaxe.

Polígonos

Os polígonos são representados por uma lista de pontos (os vértices do polígono). Provavelmente os polígonos deveriam ser considerados equivalentes aos caminhos fechados, mas são armazenados de forma diferente e possuem um conjunto próprio de rotinas de suporte.

Os valores do tipo polygon são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )  
  ( x1 , y1 ) , ... , ( xn , yn )  
  ( x1 , y1      , ... , xn , yn )  
    x1 , y1      , ... , xn , yn
```

onde os pontos são os pontos das extremidades dos segmentos de linha compondo a fronteira do polígono.

Os polígonos são mostrados utilizando a primeira sintaxe.

Círculos

Os círculos são representados por um ponto central e um raio. Os valores do tipo circle são especificado utilizando a seguinte sintaxe:

```
< ( x , y ) , r >  
( ( x , y ) , r )  
  ( x , y ) , r  
    x , y , r
```

onde (x,y) é o centro e r é o raio do círculo.

Os círculos são mostrados utilizando a primeira sintaxe.

Fonte: <http://pgdocptbr.sourceforge.net/pg80/datatype-geometric.html>

Funções e operadores geométricos

Os tipos geométricos point, box, lseg, line, path, polygon e circle possuem um amplo conjunto de funções e operadores nativos para apoiá-los, mostrados na [Tabela 9-30](#), na [Tabela 9-31](#) e na [Tabela 9-32](#).

Tabela 9-30. Operadores geométricos

Operador	Descrição	Exemplo
+	Translação	box '((0,0),(1,1))' + point '(2.0,0)'
-	Translação	box '((0,0),(1,1))' - point '(2.0,0)'
*	Escala/rotação	box '((0,0),(1,1))' * point '(2.0,0)'
/	Escala/rotação	box '((0,0),(2,2))' / point '(2.0,0)'
#	Ponto ou caixa de interseção	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Número de pontos do caminho ou do polígono	# '((1,0),(0,1),(-1,0))'
@-@	Comprimento ou circunferência	@-@ path '((0,0),(1,0))'
@@	Centro	@@ circle '((0,0),10)'
##	Ponto mais próximo do primeiro operando no segundo operando	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Distância entre	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Se sobrepõem?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
&<	Não se estende à direita de?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Não se estende à esquerda de?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<	Está à esquerda?	circle '((0,0),1)' << circle '((5,0),1)'
>>	Está à direita?	circle '((5,0),1)' >> circle '((0,0),1)'
<^	Está abaixo?	circle '((0,0),1)' <^ circle '((0,5),1)'

Operador	Descrição	Exemplo
>^	Está acima?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Se intersectam?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	É horizontal?	?- lseg '((-1,0),(1,0))'
?-	São alinhados horizontalmente?	point '(1,0)' ?- point '(0,0)'
?	É vertical?	? lseg '((-1,0),(1,0))'
?	São alinhados verticalmente	point '(0,1)' ? point '(0,0)'
?-	São perpendiculares?	lseg '((0,0),(0,1))' ?- lseg '((0,0),(1,0))'
?	São paralelos?	lseg '((-1,0),(1,0))' ? lseg '((-1,2),(1,2))'
~	Contém?	circle '((0,0),2)' ~ point '(1,1)'
@	Está contido ou sobre?	point '(1,1)' @ circle '((0,0),2)'
~=	O mesmo que?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Tabela 9-31. Funções geométricas

Função	Tipo retornado	Descrição	Exemplo
area(object)	double precision	área	area(box '((0,0),(1,1))')
box_intersect(box, box)	box	caixa de interseção	box_intersect(box '((0,0),(1,1))', box '((0.5,0.5),(2,2))')
center(object)	point	centro	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diâmetro do círculo	diameter(circle '((0,0),2.0)')
height(box)	double precision	tamanho vertical da caixa	height(box '((0,0),(1,1))')
isclosed(path)	boolean	é um caminho fechado?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	é um caminho aberto?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	comprimento	length(path '((-1,0),(1,0))')
npoints(path)	integer	número de pontos	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	integer	número de pontos	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	converte o caminho em caminho fechado	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	converte o caminho em caminho aberto	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	raio do círculo	radius(circle '((0,0),2.0)')
width(box)	double	tamanho horizontal da caixa	width(box '((0,0),(1,1))')

Função	Tipo retornado	Descrição	Exemplo
	precision		

Tabela 9-32. Funções de conversão de tipo geométrico

Função	Tipo retornado	Descrição	Exemplo
box(circle)	box	círculo em caixa	box(circle '((0,0),2.0)')
box(point, point)	box	pontos em caixa	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polígono em caixa	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	caixa em círculo	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	centro e raio em círculo	circle(point '(0,0)', 2.0)
lseg(box)	lseg	diagonal de caixa em segmento de linha	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	ponto em segmento de linha	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	point	polígono em caminho	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	constrói ponto	point(23.4, -44.5)
point(box)	point	centro da caixa	point(box '((-1,0),(1,0))')
point(circle)	point	centro do círculo	point(circle '((0,0),2.0)')
point(lseg)	point	centro do segmento de linha	point(lseg '((-1,0),(1,0))')
point(lseg, lseg)	point	intersecção	point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')
point(polygon)	point	centro do polígono	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	caixa em polígono de 4 pontos	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	círculo em polígono de 12 pontos	polygon(circle '((0,0),2.0)')
polygon(npts, circle)	polygon	círculo em polígono de npts-pontos	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	caminho em polígono	polygon(path '((0,0),(1,1),(2,0))')

É possível acessar os dois números que compõem um point como se este fosse uma matriz com os índices 0 e 1. Por exemplo, se t.p for uma coluna do tipo point, então `SELECT p[0] FROM t` retorna a coordenada X, e `UPDATE t SET p[1] = ...` altera a coordenada Y. Do mesmo modo, um valor do tipo box ou lseg pode ser tratado como sendo uma matriz contendo dois valores do tipo point.

As funções area operam sobre os tipos box, circle e path. A função area somente opera sobre o tipo de dado path se os pontos em path não se intersectarem. Por exemplo, não opera sobre o path

'((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))':PATH, entretanto opera sobre o path visualmente idêntico '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))':PATH. Se o conceito de path que intersecta e que não intersecta estiver confuso, desenhe os dois caminhos acima lado a lado em uma folha de papel gráfico.

<http://pgdocptbr.sourceforge.net/pg80/functions-geometry.html>

Capítulo de E-book Online:

Chapter: Geometric Data Types

PostgreSQL supports six data types that represent two-dimensional geometric objects. The most basic geometric data type is the `POINT`?as you might expect, a `POINT` represents a point within a two-dimensional plane.

A `POINT` is composed of an x-coordinate and a y-coordinate?each coordinate is a `DOUBLE PRECISION` number.

The `LSEG` data type represents a two-dimensional line segment. When you create a `LSEG` value, you specify two points?the starting `POINT` and the ending `POINT`.

A `BOX` value is used to define a rectangle?the two points that define a box specify opposite corners.

A `PATH` is a collection of an arbitrary number of `POINTS` that are connected. A `PATH` can specify either a closed path or an open path. In a closed path, the beginning and ending points are considered to be connected, and in an open path, the first and last points are not connected. PostgreSQL provides two functions to force a `PATH` to be either open or closed: `POPEN()` and `PCLOSE()`. You can also specify whether a `PATH` is open or closed using special literal syntax (described later).

A `POLYGON` is similar to a closed `PATH`. The difference between the two types is in the supporting functions.

A center `POINT` and a (`DOUBLE PRECISION`) floating-point radius represent a `CIRCLE`.

Table 2.18 summarizes the geometric data types.

Syntax for Literal Values

When you enter a value for geometric data type, keep in mind that you are working with a list of two-dimensional points (except in the case of a `CIRCLE`, where you are working with a `POINT` and a radius).

A single `POINT` can be entered in either of the following two forms:

`'(x, y)'`

`' x, y '`

The `LSEG` and `BOX` types are constructed from a pair of `POINTS`. You can enter a pair of `POINTS` in any of the following formats:

`'((x1, y1), (x2, y2))'`

`'(x1, y1), (x2, y2)'`


```
'x1, y1, x2, y2'
```

The `PATH` and `POLYGON` types are constructed from a list of one or more `POINTS`. Any of the following forms is acceptable for a `PATH` or `POLYGON` literal:

```
'(( x1, y1 ), ..., ( xn, yn ))'
```

```
'( x1, y1 ), ..., ( xn, yn )'
```

```
'( x1, y1, ..., xn, yn )'
```

```
'x1, y1, ..., xn, yn'
```

You can also use the syntax `'[(x1, y1), ..., (xn, yn)]'` to enter a `PATH` literal: A `PATH` entered in this form is considered to be an open `PATH`.

A `CIRCLE` is described by a central point and a floating point radius. You can enter a `CIRCLE` in any of the following forms:

```
'< ( x, y ), r >'
```

```
'(( x, y ), r )'
```

```
'( x, y ), r'
```

```
'x, y, r'
```

Notice that the surrounding single quotes are required around all geometric literals?in other words, geometric literals are entered as string literals. If you want to create a geometric value from individual components, you will have to use a geometric conversion function. For example, if you want to create a `POINT` value from the results of some computation, you would use:

```
POINT( 4, 3*height )
```

The `POINT(DOUBLE PRECISION x, DOUBLE PRECISION y)` function creates a `POINT` value from two `DOUBLE PRECISION` values. There are similar functions that you can use to create any geometric type starting from individual components. Table 2.19 lists the conversion functions for geometric types.

Table 2.19. Type Conversion Operators for the Geometric Data Types






Result Type	Meaning
<code>POINT</code>	<code>POINT(DOUBLE PRECISION x, DOUBLE PRECISION y)</code>
<code>LSEG</code>	<code>LSEG(POINT p1, POINT p2)</code>
<code>BOX</code>	<code>BOX(POINT p1, POINT p2)</code>
<code>PATH</code>	<code>PATH(POLYGON poly)</code>

Result Type	Meaning
POLYGON	POLYGON(PATH path)
	POLYGON(BOX b)
	yields a 12-point polygon
	POLYGON(CIRCLE c)
	yields a 12-point polygon
	POLYGON(INTEGER n, CIRCLE c)
	yields an n point polygon
CIRCLE	CIRCLE(BOX b)
	CIRCLE(POINT radius, DOUBLE PRECISION point)

Sizes and Valid Values

Table 2.20 lists the size of each geometric data type.

Table 2.20. Geographic Data Type Storage Requirements

Type	Size (in bytes)
POINT	16 (2  sizeof DOUBLE PRECISION)
LSEG	32 (2  sizeof POINT)
BOX	32 (2  sizeof POINT)
PATH	4+(32  number of points)[4]
POLYGON	4+(32  number of points) ^[4]
CIRCLE	24 (sizeof POINT + sizeof DOUBLE PRECISION)

[4] The size of a PATH or POLYGON is equal to $4 + (\text{sizeof LSEG} \times \text{number of segments})$.

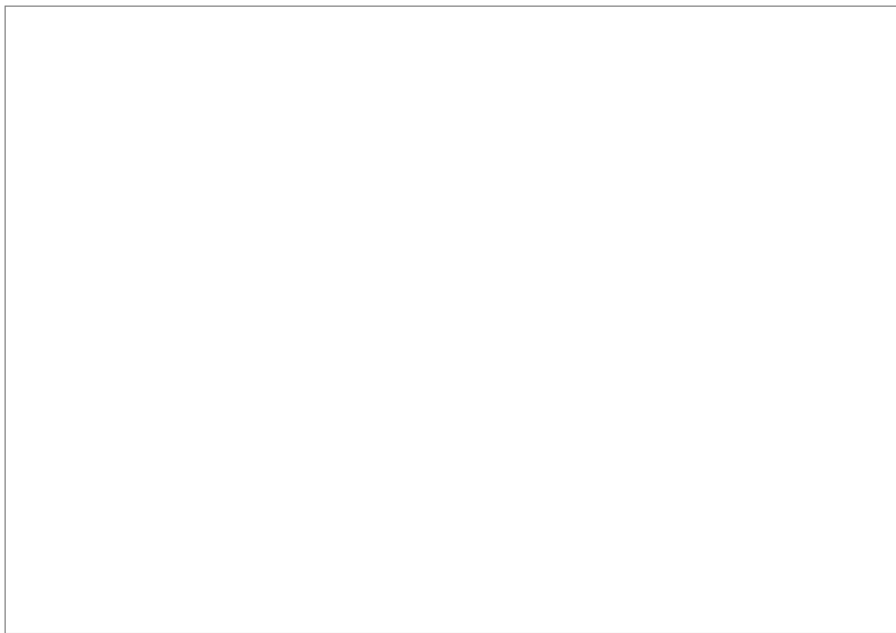


Supported Operators

PostgreSQL features a large collection of operators that work with the geometric data types. I've divided the geometric operators into two broad categories (transformation and proximity) to make it a little easier to talk about them.

Using the transformation operators, you can translate, rotate, and scale geometric objects. The `+` and `-` operators translate a geometric object to a new location. Consider Figure 2.1, which shows a `BOX` defined as `BOX(POINT(3,5), POINT(1,2))`.

Figure 2.1. `BOX(POINT(3,5), POINT(1,2))`.



If you use the `+` operator to add the `POINT(2,1)` to this `BOX`, you end up with the object shown in Figure 2.2.

Figure 2.2. Geometric translation.



You can see that the x-coordinate of the `POINT` is added to each of the x-coordinates in the `BOX`, and the y-coordinate of the `POINT` is added to the y-coordinates in the `BOX`. The `-` operator works in a similar fashion: the x-coordinate of the `POINT` is subtracted from the x-coordinates of the `BOX`, and the y-coordinate of the `POINT` is subtracted from each y-coordinate in the `BOX`.

Using the `+` and `-` operators, you can move a `POINT`, `BOX`, `PATH`, or `CIRCLE` to a new location. In each case, the x-coordinate in the second operand (a `POINT`), is added or subtracted from each x-coordinate in the first operand, and the y-coordinate in the second operand is added or subtracted from each y-coordinate in the first operand.

The multiplication and division operators (`*` and `/`) are used to scale and rotate. The multiplication and division operators treat the operands as points in the complex plane. Let's look at some examples.

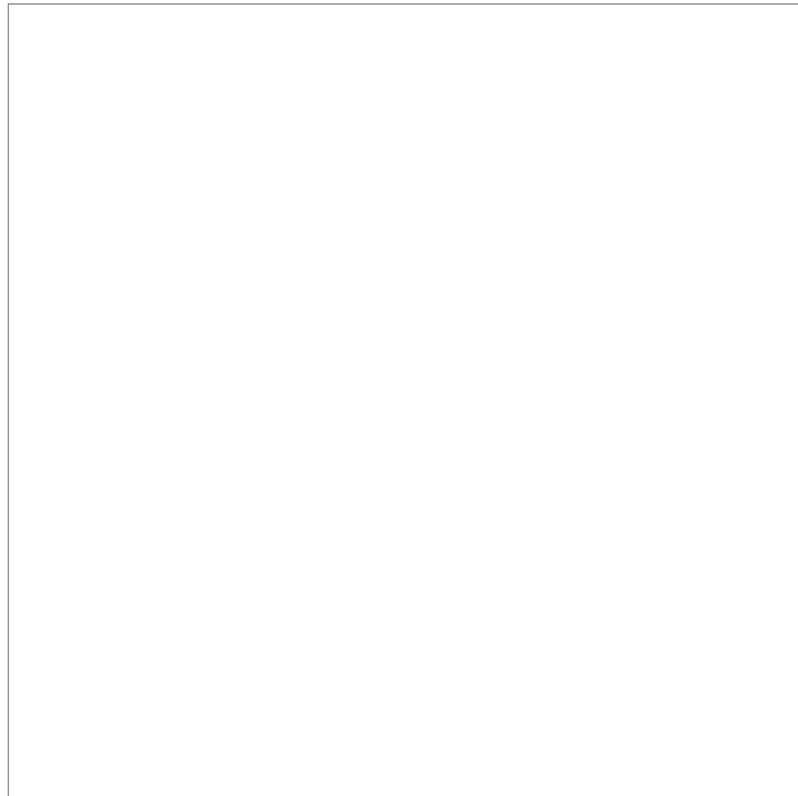
Figure 2.3 shows the result of multiplying `BOX (POINT (3, 2) , POINT (1, 1))` by `POINT (2, 0)`.

Figure 2.3. Point multiplication?scaling by a positive value.



You can see that each coordinate in the original box is multiplied by the x-coordinate of the point, resulting in `BOX (POINT (6, 4) , POINT (2, 2))`. If you had multiplied the box by `POINT (0.5, 0)`, you would have ended up with `BOX (POINT (1.5, 1) , POINT (0.5, 0.5))`. So the effect of multiplying an object by `POINT (x, 0)` is that each coordinate in the object moves away from the origin by a factor x . If x is negative, the coordinates move to the other side of the origin, as shown in Figure 2.4.

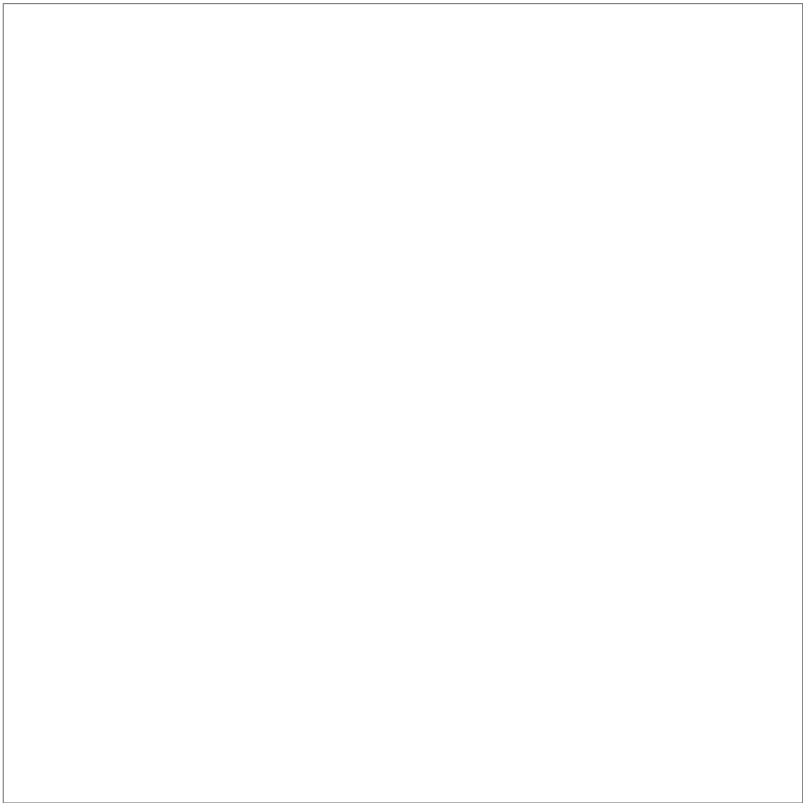
Figure 2.4. Point multiplication?scaling by a negative value.



You can see that the x-coordinate controls scaling. The y-coordinate controls rotation. When you

multiply any given geometric object by `POINT (0, y)`, each point in the object is rotated around the origin. When `y` is equal to 1, each point is rotated counterclockwise by 90° about the origin. When `y` is equal to -1, each point is rotated 90° about the origin (or 270°). When you rotate a point without scaling, the length of the line segment drawn between the point and origin remains constant, as shown in Figure 2.5.

Figure 2.5. Point multiplication?rotation.



You can combine rotation and scaling into the same operation by specifying non-zero values for both the x- and y-coordinates. For more information on using complex numbers to represent geometric points, see <http://www.clarku.edu/~djoyce/complex>.

Table 2.21 shows the valid combinations for geometric types and geometric operators.

Table 2.21. Transformation Operators for the Geometric Types

Data Types	Valid Operators (θ)
<code>POINT</code> θ <code>POINT</code>	<code>*</code> <code>+</code> <code>-</code> <code>/</code>
<code>BOX</code> θ <code>POINT</code>	<code>*</code> <code>+</code> <code>-</code> <code>/</code>
<code>PATH</code> θ <code>POINT</code>	<code>*</code> <code>+</code> <code>-</code> <code>/</code>
<code>CIRCLE</code> θ <code>POINT</code>	<code>*</code> <code>+</code> <code>-</code> <code>/</code>

The proximity operators allow you to determine the spatial relationships between two geometric objects.

First, let's look at the three containment operators. The `~` operator evaluates to `TRUE` if the left operand contains the right operand. The `@` operator evaluates to `TRUE` if the left operand is contained within the right operand. The `~=` returns `TRUE` if the left operand is the same as the right operand?two geographic objects are considered identical if the points that define the objects are identical (two circles are considered identical if the radii and center points are the same).

The next two operators are used to determine the distance between two geometric objects.

The `##` operator returns the closest point between two objects. You can use the `##` operator with the following operand types shown in Table 2.22.

Table 2.22. Closest-Point Operators

Operator	Description
<code>LSEG_a ## BOX_b</code>	Returns the point in <code>BOX_b</code> that is closest to <code>LSEG_a</code>
<code>LSEG_a ## LSEG_b</code>	Returns the point in <code>LSEG_b</code> that is closest to <code>LSEG_a</code>
<code>POINT_a ## BOX_b</code>	Returns the point in <code>BOX_b</code> that is closest to <code>POINT_a</code>
<code>POINT_a ## LSEG_b</code>	Returns the point in <code>LSEG_b</code> that is closest to <code>POINT_a</code>

The distance (`<->`) operator returns (as a `DOUBLE PRECISION` number) the distance between two geometric objects. You can use the distance operator with the operand types in Table 2.23.

Table 2.23. Distance Operators

Operator	Description (or Formula)
<code>BOX_a <-> BOX_b</code>	<code>((@ BOX_a) <-> (@ BOX_b))</code>
<code>CIRCLE_a <-> CIRCLE_b</code>	<code>((@ CIRCLE_a) <-> (@ CIRCLE_b))</code>
	?
	<code>(radius_a + radius_b)</code>
<code>CIRCLE_a <-> POLYGON_b</code>	0 if any point in <code>POLYGON_b</code> is inside <code>CIRCLE_a</code> otherwise, distance between center of <code>CIRCLE_a</code> and closest point in <code>POLYGON_b</code>
<code>LSEG_a <-> BOX_b</code>	<code>(LSEG ## BOX) <-> (LSEG ## (LSEG ## BOX))</code>
<code>LSEG_a <-> LSEG_b</code>	Distance between closest points (0 if <code>LSEG_a</code> intersects <code>LSEG_b</code>)

Operator	Description (or Formula)
$PATH_a <-> PATH_b$	Distance between closest points
$POINT_a <-> BOX_b$	$POINT_a <-> (POINT_a \ \#\# \ BOX_b)$
$POINT_a <-> CIRCLE_b$	$POINT_a <-> ((@@ \ CIRCLE_b) \ ? \ CIRCLE_b \ radius)$
$POINT_a <-> LSEG_b$	$POINT_a <-> (POINT_a \ \#\# \ LSEG_b)$
$POINT_a <-> PATH_b$	Distance between $POINT_a$ and closest points
$POINT_a <-> POINT_b$	$SQRT((\ POINT_a.x \ ? \ POINT_b.x)^2 + (\ POINT_a.y \ ? \ POINT_b.y)^2)$

Next, you can determine the spatial relationships between two objects using the left-of (<<), right-of (>>), below (<^), and above (>^) operators.

There are three overlap operators. && evaluates to TRUE if the left operand overlaps the right operand. The &> operator evaluates to TRUE if the leftmost point in the first operand is left of the rightmost point in the second operand. The &< evaluates to TRUE if the rightmost point in the first operand is right of the leftmost point in the second operand.

The intersection operator (#) returns the intersecting points of two objects. You can find the intersection of two BOXes, or the intersection of two LSEGs. The intersection of two BOXes evaluates to a BOX. The intersection of two LSEGs evaluates to a single POINT.

Finally, the ?# operator evaluates to TRUE if the first operand intersects with or overlaps the second operand.

The final set of geometric operators determines the relationship between a line segment and an axis, or the relationship between two line segments.

The ?- operator evaluates to TRUE if the given line segment is horizontal (that is, parallel to the x-axis). The ?| operator evaluates to TRUE if the given line segment is vertical (that is, parallel to the y-axis). When you use the ?- and ?| operators with a line segment, they function as prefix unary operators. You can also use the ?- and ?| operators as infix binary operators (meaning that the operator appears between two values), in which case they operate as if you specified two points on a line segment.

The ?-| operator evaluates to TRUE if the two operands are perpendicular. The ?|| operator evaluates to TRUE if the two operands are parallel. The perpendicular and parallel operators can be used only with values of type LSEG.

The final geometric operator (@@) returns the center point of an LSEG, PATH, BOX, POLYGON, or

CIRCLE.

Table 2.24. Proximity Operators for the Geometric Types

Data Types	Valid Operators (θ)
POINT θ POINT	<-> << <^ >> >^ ?- ? @
POINT θ LSEG	## <-> @
POINT θ BOX	## <-> @
POINT θ PATH	<-> @
POINT θ POLYGON	@
POINT θ CIRCLE	<-> @
LSEG θ LSEG	# ## < <-> <= <> = > >= ?# ?- ?
LSEG θ BOX	## <-> ?# @
BOX θ POINT	* + - /
BOX θ BOX	# && &< &> < <-> << <= <^ = > >= >> >^ ?# @ ~ ~=
PATH θ POINT	* + - / ~
PATH θ PATH	+ < <-> <= = > >= ?#
POLYGON θ POINT	~
POLYGON θ POLYGON	&& &< &> <-> >> << @ ~ ~=
CIRCLE θ POINT	* + - / ~
CIRCLE θ POLYGON	<->
CIRCLE θ CIRCLE	&& &< &> > <-> << <= <> <^ = > >= >> >^ @ ~ ~=

Table 2.25 summarizes the names of the proximity operators for geometric types.

Table 2.25. Geometric Proximity Operator Names

Data Types	Valid Operators (θ)
#	Intersection or point count(for polygons)
##	Point of closest proximity
<->	Distance Between
<<	Left of?
>>	Right of?
<^	Below?
>^	Above?
&&	Overlaps
&>	Overlaps to left
&<	Overlaps to right
?#	Intersects or overlaps
@	Contained in
~	Contains
~=	Same as
?-	Horizontal
?	Vertical
?-	Perpendicular
?	Parallel

Data Types

Valid Operators (θ)

@@

Center

Original em:

<http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+2.+Working+with+Data+in+PostgreSQL/Geometric+Data+Types/>

Outro capítulo de e-book online:

Using Geometric Data Types

PostgreSQL provides a set of data types you can use for storing geometric information efficiently. These data types are included in PostgreSQL's core distribution and are widely used and admired by many people. In addition to the data types themselves, PostgreSQL provides an index structure based on R-trees, which are optimized for performing spatial searching.

PHP has a simple interface for generating graphics. To generate database-driven indexes, you can combine PostgreSQL and PHP. In this section you will see how to implement the "glue" between PHP and PostgreSQL.

The goal of the next example is to implement an application that extracts points from a database and displays them in an image. To store the points in the database, you can create a table:

```
phpbook=# CREATE TABLE coord (comment text, data point);
CREATE
```

Then you can insert some values into the table:

```
phpbook=# INSERT INTO coord VALUES ('no comment', '20,20');
INSERT 19873 1
phpbook=# INSERT INTO coord VALUES ('no comment', '120,99');
INSERT 19874 1
phpbook=# INSERT INTO coord VALUES ('no comment', '137,110');
INSERT 19875 1
phpbook=# INSERT INTO coord VALUES ('no comment', '184,178');
INSERT 19876 1
```

In this example four records have been added to the table. Now that some data is in the database, you can start working on a script that generates the dynamic image:

```
<?php
```

```
include("objects/point.php");
```

```
header ("Content-type: image/png");
```

```
$im = @ImageCreate (200, 200)
      or die ("Cannot Generate Image");
```

```
$white = ImageColorAllocate ($im, 255, 255, 255);
```

```
$black = ImageColorAllocate ($im, 0, 0, 0);
```

```
ImageFill($im, 0, 0, $white);
```

```
# connecting to the database
```

```
$dbh = pg_connect("dbname=phpbook user=postgres");
```

```
if      (!$dbh)
```



```

        $this->y1 + $size/2,
        $color);
    }
}
?>

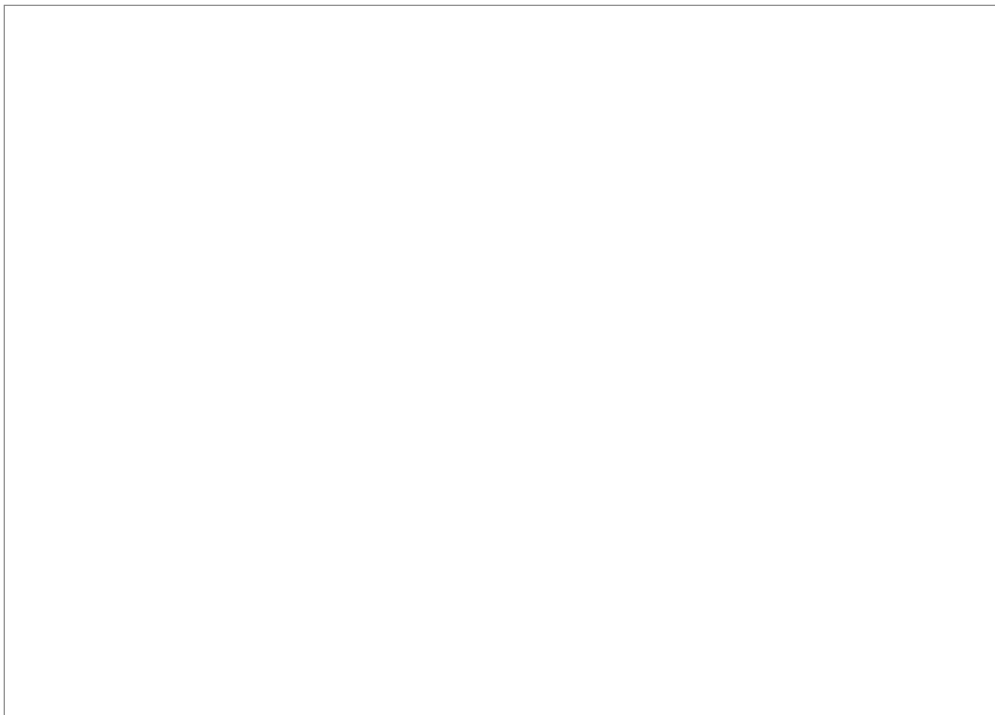
```

The class called `point` contains two variables. `$x1` contains the x coordinate of the point. `$y1` contains the y coordinate of the point. Let's take a closer look at the constructor.

One parameter has to be passed to the function. This parameter contains the data returned by PostgreSQL. In the next step the data is transformed and the two coordinates of the point are extracted from the input string. Finally the two values are assigned to `$this`.

In addition to the constructor, a function for drawing a point has been implemented. Because a point is too small, a filled rectangle is drawn. The coordinates of the rectangle are computed based on the data retrieved from the database by the constructor. [Figure 16.11](#) shows what comes out when running the script.

Figure 16.11. Dynamic points.



Points are the easiest data structure provided by PostgreSQL and therefore the PHP class you have to implement is easy as well. A more complex yet important data structure is polygons. Polygons consist of a variable number of points, so the constructor as well as the drawing functions are more complex. Before taking a look at the code, you have to create a table and insert some data into it:

```

phpbook=# CREATE TABLE mypolygon (data polygon);
CREATE

```

In the next step you can add some data to the table:

```

phpbook=# INSERT INTO mypolygon VALUES ('10,10, 150,20, 120,160'::polygon);
INSERT 19902 1
phpbook=# INSERT INTO mypolygon VALUES ('20,20, 160,30, 120,160, 90,90');
INSERT 19903 1

```

Two records have been added to the table. Let's query the table to see how the data is returned by PostgreSQL:

```
phpbook=# SELECT * FROM mypolygon;
          data
-----
((10,10),(150,20),(120,160))
((20,20),(160,30),(120,160),(90,90))
(2 rows)
```

The target of the next piece of code is to transform the data returned by the database and make something useful out of it. The next listing contains a class called `polygon` used for processing polygons:

```
<?php

# working with polygons
class polygon
{
    var $x;
    var $y;
    var $number;

    # constructor
    function polygon($string)
    {
        $string = ereg_replace("\\(\\*)\"", "", $string);
        $tmp = explode(",", $string);
        $this->number = count($tmp);
        for ($i = 0; $i < $this->number; $i = $i + 2)
        {
            $this->x[$i/2] = $tmp[$i];
            $this->y[$i/2] = $tmp[$i + 1];
        }
    }

    # drawing a polygon
    function draw($image, $color)
    {
        for ($i = 0; $i < $this->number - 1; $i++)
        {
            imageline($image, $this->x[$i], $this->y[$i],
                      $this->x[$i + 1], $this->y[$i + 1], $color);
        }
        imageline($image, $this->x[$this->number - 1],
                  $this->y[$this->number - 1], $this->x[0],
                  $this->y[0], $color);
    }
}

?>
```

`$x` will be used to store a list of x coordinates and `$y` will contain the list of y coordinates extracted from the points defining the polygon. `$number` will contain the number of points a polygon consists of.

After defining the variables belonging to the polygon, the constructor has been implemented. With the help of regular expressions, the data returned by PostgreSQL is modified and can easily be transformed into an array of values. In the next step the values in the array are counted and the various coordinates are added to `$this->x` and `$this->y`.

In addition to the constructor, a function for drawing the polygon has been implemented. The function goes through all points of the polygon and adds lines to the image being generated.

Now that you have seen how to process polygons, you can take a look at the main file of the

application:

```
<?php

include("objects/polygon.php");
header ("Content-type: image/png");
$im = @ImageCreate (200, 200)
    or die ("Cannot Generate Image");

$white = ImageColorAllocate ($im, 255, 255, 255);
$black = ImageColorAllocate ($im, 0, 0, 0);
ImageFill($im, 0, 0, $white);

# connecting to the database
$dbh = pg_connect("dbname=phpbook user=postgres");
if (! $dbh)
{
    exit ("an error has occurred<br>");
}
# drawing border
$points = array(0,0, 0,199, 199,199, 199,0);
ImagePolygon($im, $points, 4, $black);

# selecting points
$sql = "SELECT data FROM mypolygon";
$result = pg_exec($dbh, $sql);
$rows = pg_numrows($result);

# processing result
for ($i = 0; $i < $rows; $i++)
{
    $data = pg_fetch_array ($result, $i);
    $mypoly = new polygon($data["data"]);
    $mypoly->draw($im, $black);
}

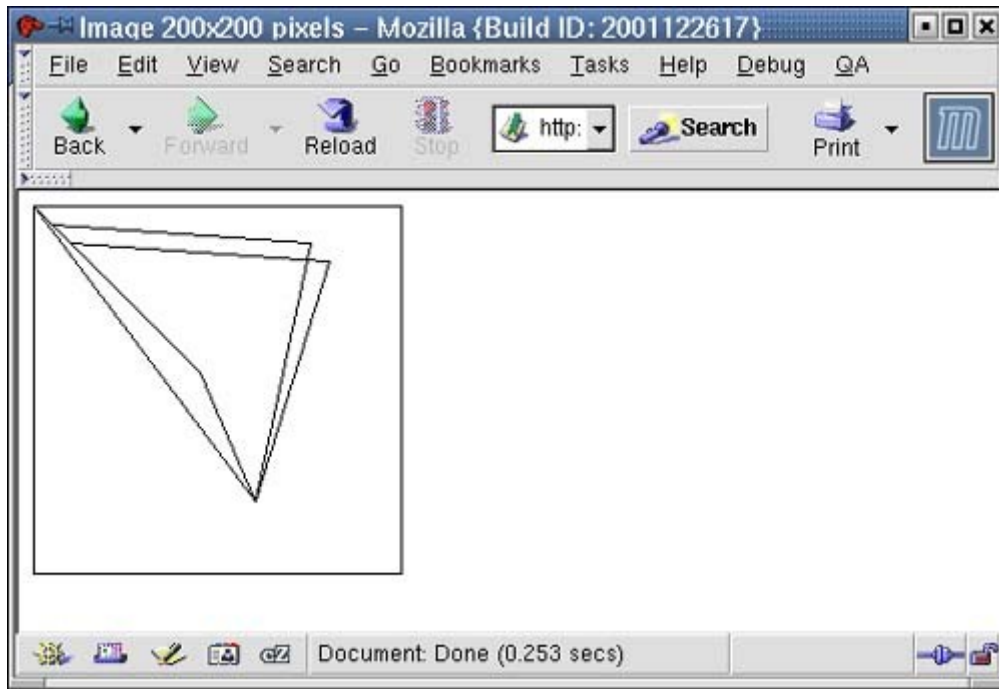
ImagePng ($im);

?>
```

First the library for processing polygons is included. In the next step an image is created, colors are allocated, the background of the image is painted white, and a connection to the database is established. After drawing the borders and retrieving all records from the database, the polygons are displayed using a loop. Inside the loop the constructor is called for every record returned by PostgreSQL. After the constructor, the `draw` function is called, which adds the polygon to the scenery.

If you execute the script, you will see the result as shown in [Figure 16.12](#).

Figure 16.12. Dynamic polygons.



Classes can be implemented for every geometric data type provided by PostgreSQL. As you have seen in this section, implementing a class for working with geometric data types is truly simple and can be done with just a few lines of code. The major part of the code consists of basic things such as creating images, allocating colors, or connecting to the database. The code that is used for doing the interaction with the database is brief and easy to understand. This should encourage you to work with PHP and PostgreSQL's geometric data types extensively.

Fonte: http://jlbtcd.eduunix.cn/index/html/php/Sams%20-%20PHP%20and%20PostgreSQL%20Advanced%20Web%20Programming/0672323826_ch16lev1sec2.html