

Banco de Dados Geográficos

Teoria e Prática com o PostgreSQL

Dr. Gilberto Ribeiro de Queiroz
gilberto.queiroz@inpe.br

Instituto Nacional de Pesquisas Espaciais - INPE
São José dos Campos, SP, Brasil

Janeiro de 2020

Prefácio

Este documento faz parte do material de apoio do curso de verão em Geoinformática e Geospatial Data Science oferecido pelo Instituto Nacional de Pesquisas Espaciais (INPE). O curso realizado entre os dias 27 e 31 de Janeiro de 2020, é coordenado pelos professores Dr. Gilberto Ribeiro de Queiroz e Dr. Rafael Duarte Coelho Dos Santos e colaboração das professoras Dra. Lúbia Vinhas e Dra. Karine Reis Ferreira.

Nessas notas de aula, tentamos ilustrar o uso das funcionalidades presentes em um servidor típico de bancos de dados, como o PostgreSQL, para trabalhar no *domínio geoespacial*. A aula sobre tipos geométricos e operações espaciais tem como objetivo fornecer os fundamentos teóricos e a prática necessária para lidar com a componente geoespacial em sistemas de bancos de dados e em linguagens de programação. Para tal apresentamos diversos exemplos em SQL com o PostgreSQL e PostGIS e trechos de código em Python.

A aula sobre programação do lado servidor de bancos de dados têm como objetivo fornecer ao aluno conceitos e técnicas de programação em bancos de dados que o auxilie em atividades de pesquisa e desenvolvimento, principalmente, para aqueles que precisarão lidar com grandes quantidades de dados. Essa parte das notas de aulas são baseadas no manual do servidor PostgreSQL, disponíveis livremente e com seções com maiores detalhes do que os apresentados aqui:

- <https://www.postgresql.org/docs/current/static/index.html>
- <https://www.postgresql.org/docs/current/static/server-programming.html>

Espera-se que o aluno aproveite a oportunidade no curso para aprofundar seus conhecimentos sobre bancos de dados, além de exercitar esse conhecimento adquirido em um ambiente real como o PostgreSQL.

Atenciosamente,

Gilberto Ribeiro de Queiroz

São José dos Campos, 27 de Janeiro de 2020

This work is licensed under a Creative Commons “Attribution-NonCommercial-ShareAlike 3.0 Unported” license.



Sumário

1 Tipos Geométricos e Operações Espaciais	1
1.1 Tipos Geométricos	3
1.2 Relacionamentos Espaciais	6
1.2.1 Matriz de 9-intersecções Estendida Dimensionalmente	6
1.2.2 Operador <code>Relate</code>	12
1.2.3 Relacionamentos Espaciais Nomeados	13
1.3 PostGIS Geometry	23
1.3.1 Carregando a Extensão PostGIS	23
1.3.2 Well-Known Text (WKT)	24
1.3.3 Operadores Métricos	25
1.4 Tipos Geométricos em Python	28
1.4.1 Instalação	29
1.4.2 Tipos Geométricos	30
2 Programação do Lado Servidor no PostgreSQL	39
2.1 PL/pgSQL	40
2.1.1 Comentários	43
2.1.2 Estruturas Condicionais	44
2.1.3 Mensagens e Exceções	46
2.1.4 Tipos dos Parâmetros e Tipo de Retorno	49
2.1.5 Execução de Comandos SQL	50
2.1.6 Retornando Conjuntos ou Tuplas	59
2.1.7 Comandos de Repetição	62
2.1.8 Blocos	67

2.2	Triggers	68
2.3	Extensões	74
2.4	Considerações Finais	75
2.5	Exercícios	77

Lista de Tabelas

Listas de Figuras

1.1	Representação de feições geográficas através de objetos geométricos	2
1.2	Diagrama de classes do modelo geométrico da <i>OGC Simple Feature</i> . Fonte: [1].	3
1.3	Exemplos dos tipos geométricos da <i>OGC-SFS</i>	4
1.4	Matriz de 9-Intersecções Estendida Dimensionalmente.	6
1.5	Interior, fronteira e exterior dos diversos tipos geométricos.	7
1.6	Configuração espacial de dois objetos geométricos A e B	9
1.7	Relacionamento espacial definido por A e B segundo a matriz de intersecções.	10
1.8	Intersecção entre os componentes dos objetos A e B	12
1.9	DE-9IM para o relacionamento Equals	15
1.10	A e B são geometrias espacialmente iguais.	15
1.11	DE-9IM para o relacionamento Disjoint	16
1.12	A e B são geometrias espacialmente disjuntas.	16
1.13	DE-9IM para o relacionamento Touches	17
1.14	A e B são geometrias que se tocam.	18
1.15	DE-9IM para o relacionamento Crosses para os casos P/L, P/A, ou L/A. . .	18
1.16	DE-9IM para o relacionamento Crosses para o caso L/L.	18
1.17	A e B são geometrias que se cruzam.	19
1.18	DE-9IM para o relacionamento Within	19
1.19	A geometria A está dentro da geometria B	20
1.20	DE-9IM para o relacionamento Contains	20
1.21	DE-9IM para o relacionamento Overlaps nos casos P/P ou A/A.	21
1.22	DE-9IM para o relacionamento Overlaps no caso L/L.	21
1.23	A geometria A sobrepõe a geometria B	22

1.24 DE-9IM para o relacionamento <code>Intersects</code>	22
1.25 DE-9IM para o relacionamento <code>Intersects</code>	23
1.26 DE-9IM para o relacionamento <code>Intersects</code>	23
1.27 DE-9IM para o relacionamento <code>Intersects</code>	23
1.28 Área polígono	25
1.29 Comprimento da linha	26
1.30 Geometria <i>A</i> e <i>B</i>	27

Listas de Códigos

1.1	Relacionamento espacial entre os objetos <i>A</i> e <i>B</i>	13
1.2	Relacionamento espacial entre os objetos <i>A</i> e <i>B</i>	13
1.3	Para criar um novo banco de dados no PostgreSQL.	23
1.4	Habilitando a extensão PostGIS.	24
1.5	Informações de configurações da sua extensão PostGIS.	24
1.6	Representação ponto de coordenadas x=1 e y=8 WKT	24
1.7	Representação Linha três vértices	24
1.8	Representação coleção de pontos	24
1.9	Representação coleção de linhas	25
1.10	Representação coleção de polígonos	25
1.11	Área polígonos	26
1.12	Perímetro polígonos	26
1.13	Comprimento da linha	27
1.14	Distância entre duas geometrias	27
1.15	Instalação da biblioteca shapely.	29
1.16	Importando a biblioteca shapely.	29
1.17	Importando a classe Point.	30
1.18	Construtor de um objeto Point.	30
1.19	Acessando as coordenadas do Point.	31
1.20	Comprimento de um Point.	31
1.21	Área de um Point.	31
1.22	Fronteira de um Point.	31
1.23	Importando a classe LineString.	31
1.24	Construtor de um objeto LineString.	32
1.25	Acessando as coordenadas da LineString.	32
1.26	Comprimento, área e fronteira da LineString.	32
1.27	Acessando as coordenadas da LineString.	32
1.28	Acessando as coordenadas da LineString.	33
1.29	Importando a classe LinearRing.	33
1.30	Construtor de um objeto LinearRing.	33
1.31	Comprimento, área e fronteira de um objeto LineString.	33
1.32	Importando a classe Polygon.	34
1.33	Construtor de um objeto Polygon.	34

1.34 Anel externo e anéis internos de um objeto Polygon.	34
1.35 Comprimento, área e fronteira de um objeto Polygon.	35
1.36 Importando a classe MultiPoint.	35
1.37 Construtor de um objeto MultiPoint.	35
1.38 Propriedade geom MultiPoint.	35
1.39 Importando a classe MultiPoint.	36
1.40 Construtor de um objeto MultiLineString.	36
1.41 Propriedade geom MultiLineString	36
1.42 Importando a classe MultiPolygon.	36
1.43 Construtor de um objeto MultiPolygon.	37
1.44 Construtor de um objeto MultiPolygon com polígonos intermediários.	37

List of Algorithms

Capítulo 1

Tipos Geométricos e Operações Espaciais

A forma de modelar e representar os fenômenos geográficos no computador depende de sua percepção na forma de **entidades discretas** (objetos) ou **campos contínuos**.

Quando lidamos com fenômenos onde temos um valor definido para uma ou mais variáveis de observação em toda localização possível do espaço, estamos compreendendo tal fenômeno como um *campo contínuo*. Elevação, temperatura de superfície, risco de incêndio na vegetação, e radiância da superfície são exemplos de campos contínuos.

Quando percebemos o fenômeno em questão por objetos com fronteiras bem definidas e pertencentes a uma certa categoria, estamos compreendendo esse fenômeno como *entidades discretas*. Unidades de conservação estadual e federal, organização territorial, arruamento, trechos rodoviários, escolas, hospitais, linhas de transmissão de energia elétrica, são alguns exemplos de entidades discretas.

Para representar os dados dessas duas formas de conceitualização do espaço geográfico, em geral, utilizamos a **representação matricial** para fenômenos modelados como campos contínuos, e a **representação vetorial** para entidades discretas.

Nesta parte do curso estamos interessados na representação vetorial dos dados.

As entidades codificadas usando dados vetoriais são usualmente chamadas de **feições** (ou *features*). Nesse contexto, uma feição pode ser representada computacionalmente por diversas características, as quais chamamos de **atributos da feição**. Um **atributo** possui um nome, sendo associado a um determinado **tipo de dado**, como um número, uma sequência de caracteres (texto), ou uma data.

Além dos atributos alfanuméricos, uma feição é descrita por um ou mais **atributos geométricos**, associados a um tipo de dado geométrico. Um **tipo de dado geométrico** é capaz de representar elementos geométricos primitivos tais como pontos, linhas e polígonos ou coleções desses elementos.

A Figura 1.1 apresenta alguns tipos de objetos geográficos representados por feições com representações geométricas de pontos (hidrelétricas e termoelétricas), linhas (logradouro) e polígonos (municípios brasileiros).

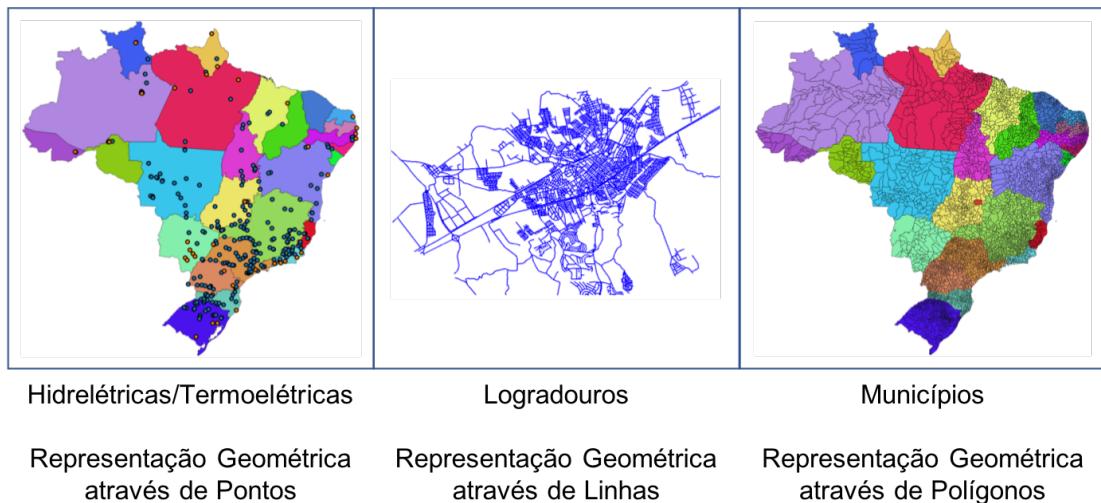


Figura 1.1: Representação de feições geográficas através de objetos geométricos.

1.1 Tipos Geométricos

Atualmente, o modelo geométrico e as operações espaciais encontradas nos diversos sistemas geoespaciais são baseados na especificação conhecida por **OGC Simple Feature** [1]. Essa especificação padroniza o nome e as definições dos tipos geométricos bem como a semântica das operações espaciais, em especial, os relacionamentos espaciais (ou topológicos). Iremos nos referir a essa especificação com a sigla **OGC-SFS**.

A Figura 1.2, abaixo, apresenta o modelo geométrico definido na *OGC-SFS*.

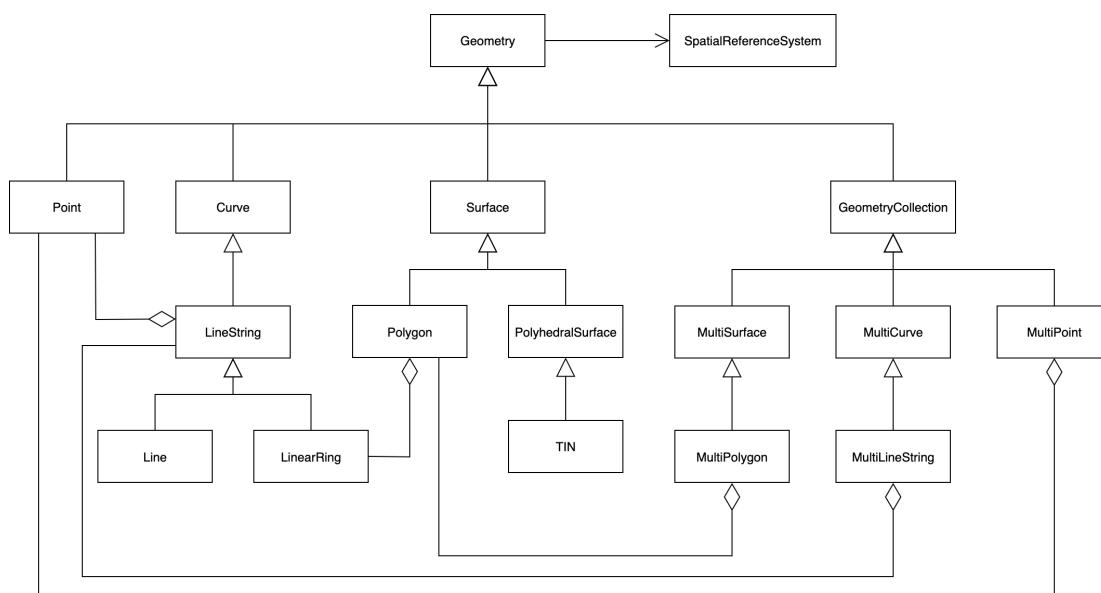


Figura 1.2: Diagrama de classes do modelo geométrico da *OGC Simple Feature*.
Fonte: [1].

Como pode ser observado, são definidas classes para representações de objetos geométricos na forma de pontos, curvas, superfícies e coleções geométricas. Além disso, todos os tipos geométricos estão associados a um **sistema de referência espacial**, que é usado para descrever o sistema de coordenadas no qual o objeto geométrico encontra-se definido.

Os objetos geométricos definidos por essa hierarquia de classes pode existir no espaço R^2 , R^3 ou R^4 . Geometrias no R^2 possuem pontos com valores de coordenadas em x e y . Geometrias no R^3 possuem pontos com valores de coordenadas em x , y e z ou x , y e m . Geometrias no R^4 possuem pontos com valores de coordenadas em x , y , z e m . Em geral, a coordenada m representa algum tipo de medida.

A Figura 1.3 ilustra graficamente objetos associados aos tipos geométricos representados pelas classes do diagrama da Figura 1.2.

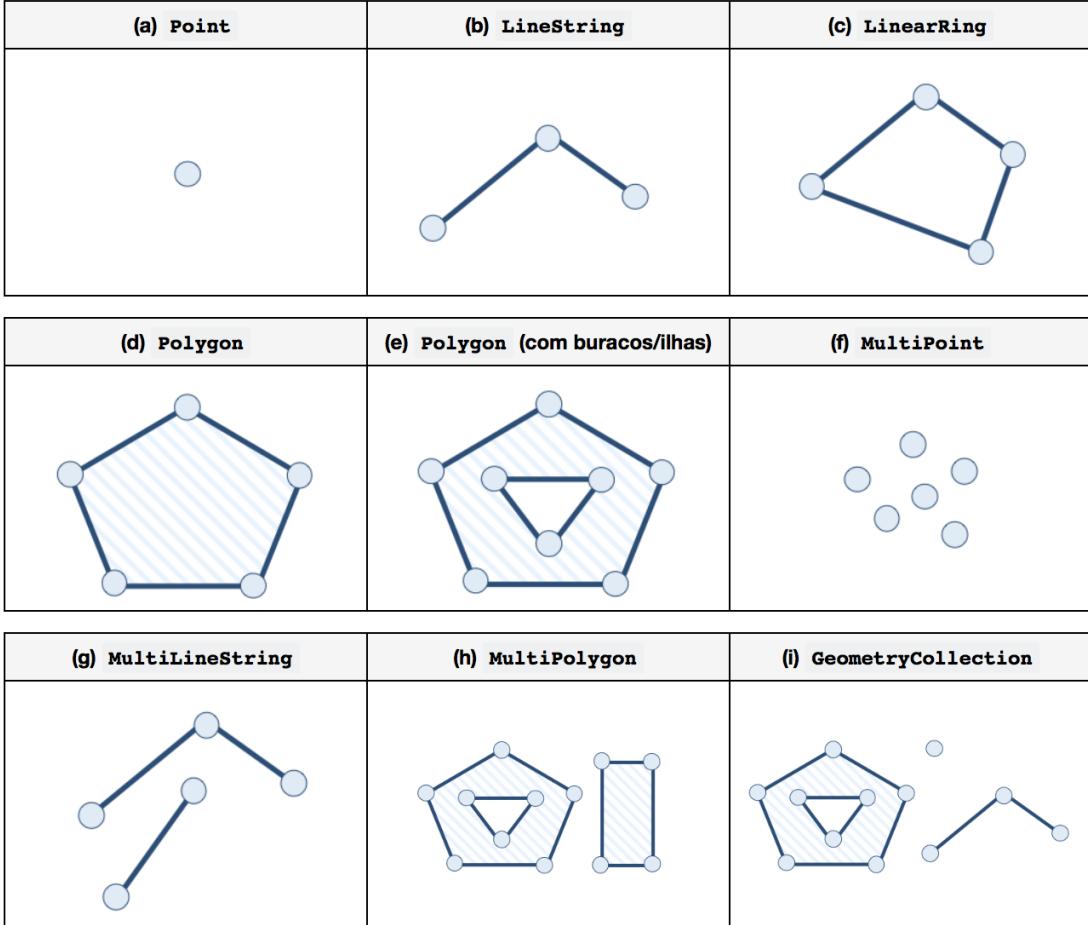


Figura 1.3: Exemplos dos tipos geométricos da *OGC-SFS*.

O tipo **Point** (Figuras 1.2 e 1.3a) representa pontos no espaço R^2 , R^3 ou R^4 . Um ponto é um objeto geométrico 0-dimensional (dimensão topológica), isto é, não possui comprimento, largura, altura, ou volume, representando uma única localização no sistema de coordenadas. A fronteira de um ponto é o conjunto vazio. Em geral, utilizamos esse tipo de geometria para representar atributos de feições associadas a ocorrências ou eventos, como incidência de crimes ou doenças.

O tipo **Curve** (Figura 1.2) representa a imagem contínua de uma linha. Uma curva é um objeto geométrico 1-dimensional (dimensão topológica), isto é, possui comprimento mas não possui largura, altura, ou volume. Em geral, utilizamos elementos geométricos

das subclasses de `Curve` para representar entidades lineares tais como rodovias, linhas de transmissão de energia elétrica, dutos, arruamentos, entre outras. Uma curva pode conter coordenadas com m ou z .

A subclasse `LineString` representa linhas com interpolação linear entre pontos consecutivos (Figuras 1.2 e 1.3b). A fronteira de uma **linha aberta** é definida como sendo os pontos extremos dessa linha. O primeiro ponto é chamado de *ponto inicial (start point)* e o último ponto da sequência, *ponto final (end point)*.

A subclasse `LinearRing` representa linhas fechadas, denominadas anéis, cujo ponto inicial e final são coincidentes (Figuras 1.2 e 1.2c). A fronteira de um anel é definida como o conjunto vazio. Essa classe é o bloco básico para construção de polígonos (classe `Polygon`).

O tipo `Surface` representa objetos geométricos 2-dimensionais (dimensão topológica), isto é, objetos que possuem área (largura e altura) mas não possuem volume. Esse tipo geométrico pode ser utilizado para representar entidades discretas como áreas de cultivo, unidades de conservação florestal, divisões territoriais, entre outras.

A subclasse `Polygon` representa polígonos que podem ser formados por um anel externo e zero ou mais anéis internos (buracos ou ilhas). A Figura 1.3d representa um polígono formado apenas por um anel, que é o anel externo. O polígono mostrado na Figura 1.3e representa um polígono formado por um anel externo e um anel interno. A fronteira de um polígono é definida como sendo o conjunto de todos os anéis que o delimitam.

O diagrama da Figura 1.2 ainda contém classes que representam coleções de geometrias. As classes `MultiPoint`, `MultiLineString` e `MultiPolygon` representam, respectivamente, coleções homogêneas de pontos (Figura 1.3f), linhas (Figura 1.3g) e polígonos (Figura 1.3h). A classe `GeometryCollection` representa coleções geométricas formadas por qualquer combinação de outros elementos geométricos, inclusive das coleções homogêneas. A coleção heterogênea mostrada na Figura 1.3i é composta de um polígono, uma linha e um ponto.

Em geral, o tipo `GeometryCollection` é introduzido nos sistemas para acomodar o resultado de operações espaciais complexas sobre os tipos de geometria elementares. Por exemplo, o resultado da operação de interseção entre dois polígonos pode resultar em um conjunto de pontos, linhas e polígonos, de forma que é necessário um contêiner especial para acomodar esse resultado.

O documento da *OGC-SFS* define também o conjunto de operações sobre os tipos geométricos do diagrama da Figura 1.2. A seção a seguir irá detalhar as operações que possibilitam determinar os relacionamentos espaciais entre objetos espaciais.

1.2 Relacionamentos Espaciais

Entre as operações definidas na *OGC-SFS*, existe um conjunto que merece uma atenção especial: os **operadores topológicos**. Esses operadores são extensivamente utilizados na construção de consultas espaciais envolvendo o relacionamento espacial entre objetos geográficos.

1.2.1 Matriz de 9-intersecções Estendida Dimensionalmente

Os relacionamentos espaciais são definidos com base no paradigma da **Matriz de 9-Intersecções Estendida Dimensionalmente (DE-9IM)** proposto por [2]. A abordagem básica desse método consiste em comparar dois objetos geométricos através de testes com a intersecção entre seus interiores, fronteiras e exteriores. Desta forma, podemos construir uma matriz 3×3 como a mostrada abaixo:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	$\dim(I(A) \cap I(B))$	$\dim(I(A) \cap F(B))$	$\dim(I(A) \cap E(B))$
<i>Fronteira(A)</i>	$\dim(F(A) \cap I(B))$	$\dim(F(A) \cap F(B))$	$\dim(F(A) \cap E(B))$
<i>Exterior(A)</i>	$\dim(E(A) \cap I(B))$	$\dim(E(A) \cap F(B))$	$\dim(E(A) \cap E(B))$

Figura 1.4: Matriz de 9-Intersecções Estendida Dimensionalmente.

Na matriz acima, $I(A)$, $F(A)$ e $E(A)$ refere-se, respectivamente, ao interior, fronteira e exterior do objeto A . De maneira análoga, $I(B)$, $F(B)$ e $E(B)$ refere-se, respectivamente, ao interior, fronteira e exterior do objeto B .

A dimensionalidade máxima dos objetos resultantes da intersecção dos componentes avaliados, $\dim(x)$, pode ser:

- \emptyset : Caso os componentes não tenham intersecção. Adotaremos o valor -1 para este caso.
- 0: Se a intersecção dos componentes resulta em um ponto.
- 1: Se a intersecção dos componentes resulta em alguma curva.
- 2: Se a intersecção dos componentes resulta em alguma superfície. Adotaremos também o termo **Área** ou **Região** como sinônimos.

A Figura 1.5 ilustra cada um dos componentes dos tipos de objetos geométricos definidos na *OGC-SFS*.

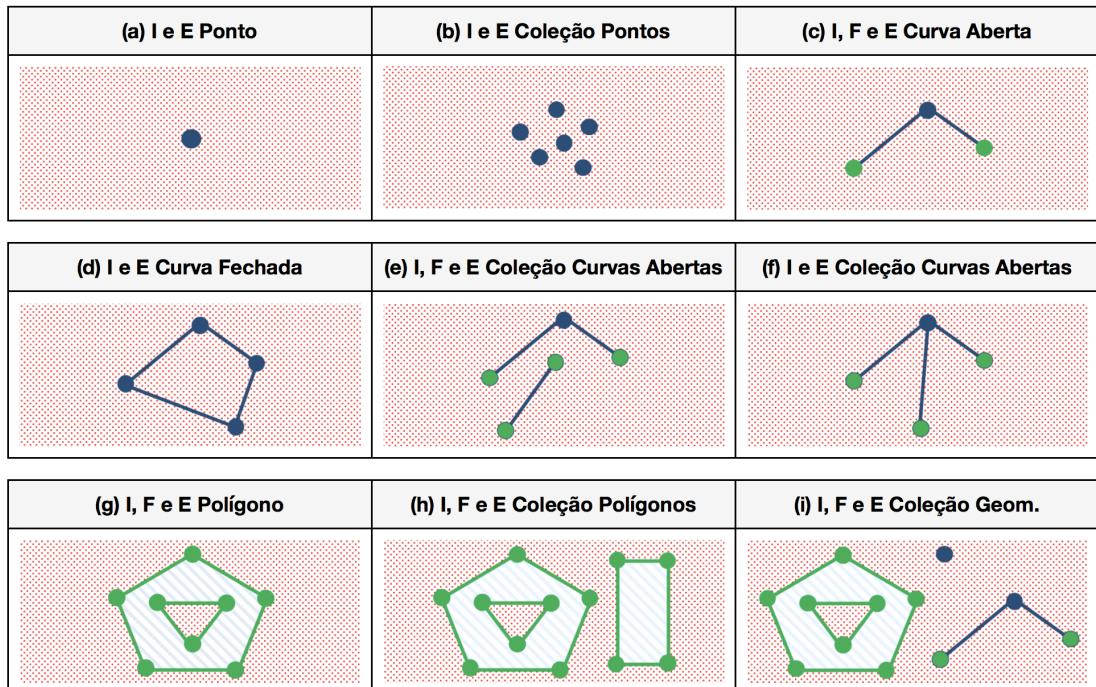


Figura 1.5: Interior, fronteira e exterior dos diversos tipos geométricos.

A fronteira de um objeto geométrico é formada por um conjunto de objetos geométricos de uma dimensão abaixo do objeto em questão.

Assim, a fronteira de um objeto do tipo `Point` é o conjunto vazio, uma vez que um ponto é um objeto 0-dimensional (dimensão topológica). Observe na Figura 1.5a que no caso de um ponto, temos apenas a definição de seu interior, destacado em azul escuro, e seu exterior, representado pela região em vermelho. Portanto, o interior de um ponto é um objeto 0-dimensional e seu exterior, um objeto 2-dimensional.

Um objeto geométrico do tipo `MultiPoint`, que também é 0-dimensional, tem sua fronteira definida como o conjunto vazio. O interior é formado pelos próprios pontos da coleção, destacados em azul escuro na Figura 1.5b, e o exterior é região destacada em vermelho nessa figura.

A fronteira de uma curva aberta (`Curve`) consiste nos pontos inicial e final dessa curva. A Figura 1.5c mostra uma exemplo de curva aberta, onde os pontos destacados em verde formam a fronteira dessa curva. Portanto, a fronteira de uma curva aberta é um conjunto de objetos geométricos 0-dimensional. O interior de uma curva aberta é formado pelos infinitos pontos de sua imagem, exceto os pontos da fronteira. Na Figura 1.5c, a linha em azul escuro corresponde ao interior da curva, que é um objeto geométrico 1-dimensional. O exterior da curva é representado pela região em vermelho que não contém os pontos da fronteira e do interior da curva.

A fronteira de uma curva fechada (`Curve`) é o conjunto vazio. A Figura 1.5d mostra um exemplo de curva fechada. Nessa figura, podemos observar que o interior é formado por todos os pontos da curva, incluindo os pontos inicial e final, formando um objeto geométrico 1-dimensional. O exterior de uma curva fechada é uma região, um objeto 2-dimensional, destacado em vermelho nessa figura.

A fronteira de uma coleção de curvas (`MultiCurve`) consiste nos pontos na fronteira de um número ímpar de elementos da curva. A Figura 1.5e mostra um exemplo de coleção de curvas abertas. Repare que os pontos destacados em verde formam a fronteira dessa curva, enquanto os pontos sobre as imagens das linhas, destacados em azul escuro, forma seus interiores e seu exterior é destacado pela região em vermelho. A Figura 1.5f ilustra a regra de que os pontos da fronteira devem pertencer a um número ímpar de elementos da curva.

Uma coleção de curvas (**MultiCurve**) é dita fechada se todos os seus elementos são fechados, consequentemente, a fronteira de uma coleção de curvas fechadas é o conjunto vazio.

A fronteira de um polígono (**Polygon**) consiste no seu conjunto de anéis (**LinearRing**). A Figura 1.5g ilustra esse caso. Repare que o exterior de um polígono com buraco é desconectado.

A fronteira de uma coleção de polígonos (**MultiPolygon**) consiste no conjunto de anéis (**LinearRing**) de seus polígonos (**Polygon**) (Figura 1.5h).

A fronteira de um conjunto de objetos geométricos heterogêneos (**GeometryCollection**) cujo os interiores dos seus elementos são disjuntos, consiste nos objetos geométricos das fronteiras desses elementos obedecendo à regra de que os pontos devem pertencer a um número ímpar de elementos (Figura 1.5i).

Vamos considerar os objetos geométricos apresentados na Figura 1.6.

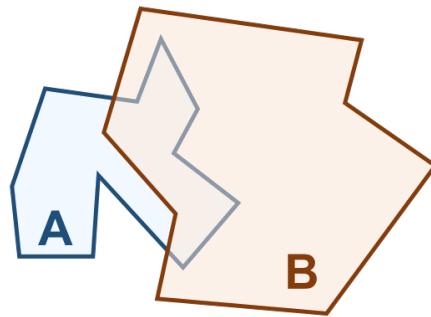


Figura 1.6: Configuração espacial de dois objetos geométricos A e B .

Podemos estabelecer o seguinte:

- $I(A)$: área mostrada em azul claro.
- $F(A)$: formada por um único anel (externo), em azul escuro.
- $E(A)$: é definido como toda a região do espaço que não compreenda a fronteira e o interior de A .

De maneira análoga:

- $I(B)$: área mostrada em laranja claro.
- $F(B)$: formada por um único anel (externo), em laranja escuro.
- $E(B)$: é definido como toda a região do espaço que não compreenda a fronteira e o interior de B .

O relacionamento espacial definido por A e B segundo a matriz de intersecções é a seguinte:

	$Interior(B)$	$Fronteira(B)$	$Exterior(B)$
$Interior(A)$	2	1	2
$Fronteira(A)$	1	0	1
$Exterior(A)$	2	1	2

Figura 1.7: Relacionamento espacial definido por A e B segundo a matriz de intersecções.

A matriz acima foi definida considerando:

- Os interiores de A e B são regiões, isto é, objetos 2-dimensional. A intersecção desses dois componentes forma uma nova região, a área em vermelho na Figura 1.8a, que também é um objeto 2-dimensional. Logo, a $\dim(I(A) \cap I(B)) = 2$.
- A fronteira de B é formada por um anel (laranja), um objeto 1-dimensional. A intersecção entre $I(A)$ e a $F(B)$ forma uma linha, destacada em vermelho na Figura 1.8b, que é um objeto 1-dimensional. Logo, a $\dim(I(A) \cap F(B)) = 1$.
- O exterior de B é uma região. A intersecção do $I(A)$ e o $E(B)$ forma a área destacada em vermelho da Figura 1.8c. Logo, a $\dim(I(A) \cap E(B)) = 2$.
- A fronteira de A é formada por um anel (azul). A intersecção entre $F(A)$ e o $I(B)$ forma uma linha, destacada em vermelho na Figura 1.8d, que é um objeto 1-dimensional. Logo, a $\dim(F(A) \cap I(B)) = 1$.
- A intersecção entre as fronteiras de A e B são os pontos mostrados em vermelho na Figura 1.8e. Como um ponto é um objeto 0-dimensional, a $\dim(F(A) \cap F(B)) = 0$.

- A intersecção entre a fronteira de A e o exterior de B forma a linha mostrada na Figura 1.8f. Logo, a $\dim(F(A) \cap E(B)) = 1$.
- A intersecção entre exterior de A e interior de B forma uma área (Figura 1.8g). Logo, a $\dim(E(A) \cap I(B)) = 2$.
- A intersecção entre exterior de A e fronteira de B forma a linha mostrada na Figura 1.8h. Logo, a $\dim(E(A) \cap F(B)) = 1$.
- A intersecção entre os exteriores de A e B formam uma área (Figura 1.8i). Logo, a $\dim(E(A) \cap E(B)) = 2$.

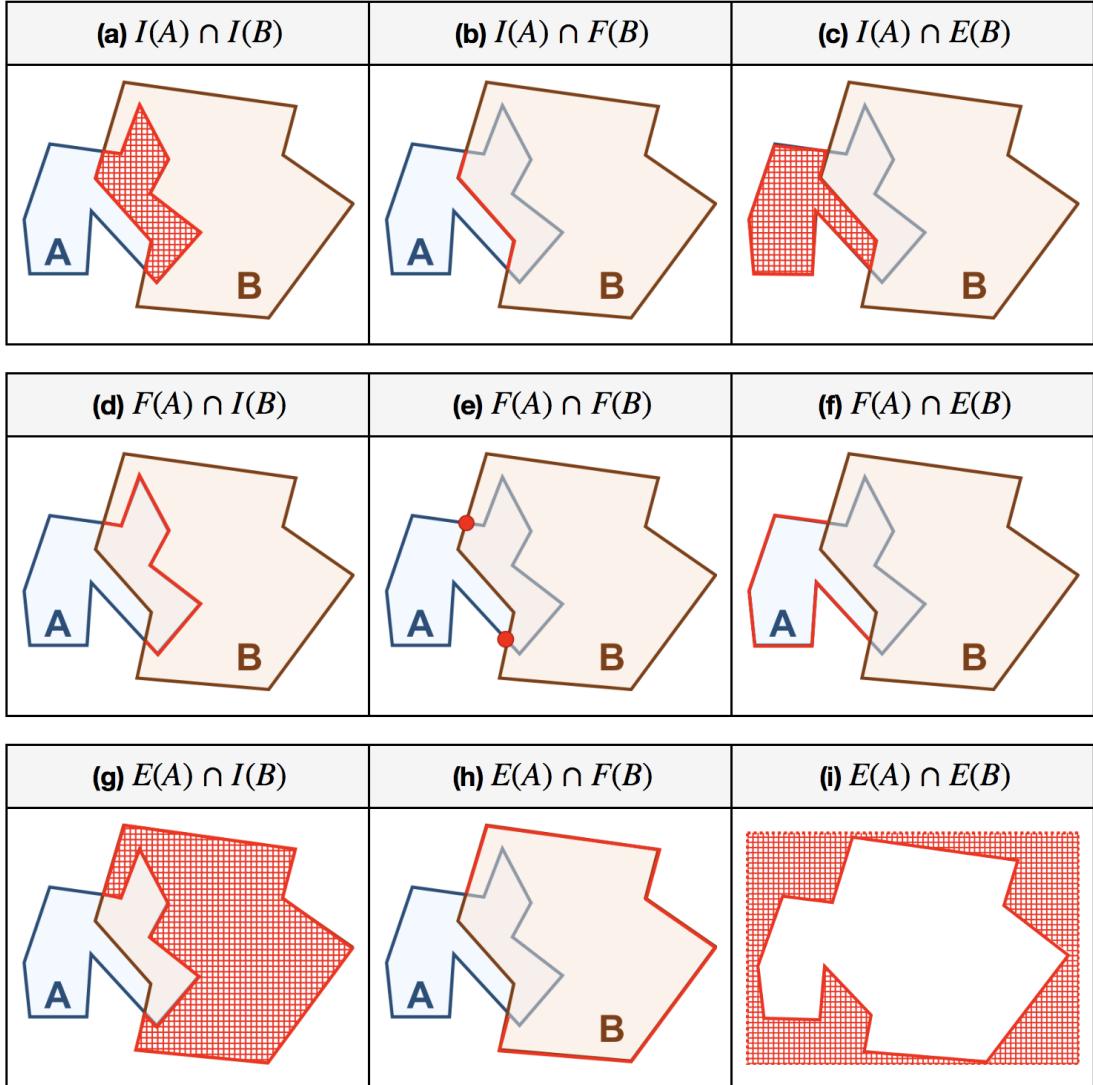


Figura 1.8: Intersecção entre os componentes dos objetos A e B .

1.2.2 Operador Relate

Em geral, os ambientes computacionais que dão suporte a *OGC-SFS*, introduzem uma função chamada **Relate** (ou *ST_Relate*) que permite determinar a matriz de intersecções ou testar se dois objetos satisfazem um determinado padrão dessa matriz. Assim, tanto em Python quanto em C++, ou Java, ou em SQL, é possível testar o relacionamento espacial entre dois objetos.

O exemplo abaixo mostra como seria o uso dessa função para saber o relacionamento espacial entre dois polígonos hipotéticos:

Listagem 1.1 Relacionamento espacial entre os objetos A e B .

```
A = Polygon(...)
B = Polygon(...)

M = Relate(A, B)
```

Nesse caso, M é a matriz resultante do relacionamento espacial entre A e B . Se A e B forem os polígonos mostrados na Figura 1.6, teríamos M codificado como a seguinte string: "212101212".

O próximo exemplo mostra como seria a utilização da função `Relate` para saber se dois objetos A e B satisfazem um padrão específico da matriz de intersecções:

Listagem 1.2 Relacionamento espacial entre os objetos A e B .

```
A = Polygon(...
B = Polygon(...)

M = "212101212"

resultado = Relate(A, B, M)
```

Nesse caso, M é a matriz com o padrão que desejamos verificar se os objetos A e B satisfazem. Se A e B forem os polígonos mostrados na Figura 1.6, essa função retorna o valor booleano `True`.

1.2.3 Relacionamentos Espaciais Nomeados

Para facilitar a construção de predicados topológicos nas consultas espaciais existe um conjunto de 8 operadores com nomes bem definidos que são baseados na sobrecarga de alguns padrões da matriz de intersecções. Os operadores nomeados são: *Equals*, *Disjoint*,

Intersects, Touches, Crosses, Within, Contains, e Overlaps. Esses operadores comparam pares de geometrias e são definidos utilizando os seguintes valores para as células da matriz de intersecções:

- **T:** indica que $\dim(x) \in \{0, 1, 2\}$, ou seja, $x \neq \emptyset$. Em outras palavras, o resultado da intersecção é um objeto x que pode ser qualquer elemento geométrico (ponto, curva ou superfície).
- **F:** indica que $\dim(x) = \emptyset$, ou seja, $x = \emptyset$. Em outras palavras, o resultado da intersecção é um conjunto vazio. Adotaremos que $\dim(x) = \emptyset \implies \dim(x) = -1$, de maneira que usaremos -1 quando necessário para indicar que a intersecção é vazia.
- *****: indica que $\dim(x) \in \{-1, 0, 1, 2\}$, ou seja, não importa se há ou não intersecção.
- **0:** indica que $\dim(x) = 0$, ou seja, que a dimensão máxima do elemento geométrico resultante da intersecção é zero, isto é, um ponto.
- **1:** indica que $\dim(x) = 1$, ou seja, que a dimensão máxima do elemento geométrico resultante da intersecção é uma curva.
- **2:** indica que $\dim(x) = 2$, ou seja, que a dimensão máxima do elemento geométrico resultante da intersecção é uma superfície (ou área).

Além disso, em alguns casos diferenciamos os padrões pelos tipos de objetos testados. Usamos a seguinte nomenclatura para os tipos de objetos:

- **Puntuais** ou P: pontos e coleções homogêneas de pontos.
- **Curvas** ou L: linhas, anéis, ou coleções homogêneas de linhas.
- **Superfícies** ou A: polígonos ou coleções homogêneas de polígonos.

$\text{Equals}(\text{Geometry}, \text{Geometry}) \rightarrow \text{bool}$

Retorna verdadeiro (`True`) se as duas geometrias são espacialmente iguais. Neste caso, a matriz de intersecção para as duas geometrias deve satisfazer o seguinte padrão:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>T</i>	<i>F</i>	<i>F</i>
<i>Fronteira(A)</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>Exterior(A)</i>	<i>F</i>	<i>F</i>	<i>T</i>

Figura 1.9: DE-9IM para o relacionamento `Equals`.

Pela matriz acima, o relacionamento `Equals` garante que os interiores das duas geometrias tenham intersecção e que os interiores e fronteiras de um objeto não tenha intersecção com o exterior do outro.

A Figura 1.10 apresenta alguns pares de objetos que satisfazem o relacionamento espacial `Equals`, de acordo com o padrão mostrado acima:

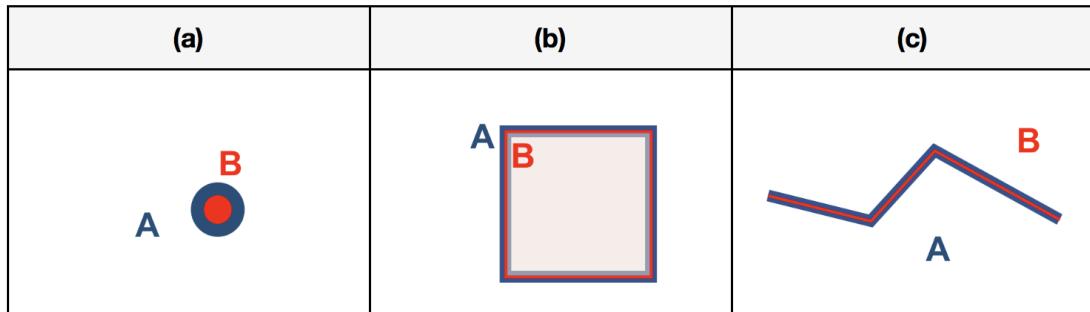


Figura 1.10: *A* e *B* são geometrias espacialmente iguais.

Observação: No caso dos pares de geometrias serem curvas/curvas ou superfícies/superfícies, mesmo que o número de pontos desses objetos sejam diferentes, o relacionamento `Equals` pode ser verdadeiro.

Disjoint(Geometry, Geometry) → bool

Retorna verdadeiro (`True`) se as duas geometrias não possuem qualquer tipo de interação espacial, isto é, se os interiores e fronteiras não possuem qualquer intersecção, exceto com os exteriores. Neste caso, a matriz de intersecção para as duas geometrias deve satisfazer o seguinte padrão:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>F</i>	<i>F</i>	*
<i>Fronteira(A)</i>	<i>F</i>	<i>F</i>	*
<i>Exterior(A)</i>	*	*	*

Figura 1.11: DE-9IM para o relacionamento `Disjoint`.

A matriz acima define que no relacionamento `Disjoint` nem a fronteira, nem o interior de um objeto possui intersecção com a fronteira ou interior do outro objeto. Observe que o uso do padrão * nas demais células indica que nenhuma outra condição importa para determinar este relacionamento.

A Figura 1.12 apresenta alguns pares de objetos que satisfazem o relacionamento espacial `Disjoint`, de acordo com o padrão mostrado acima:

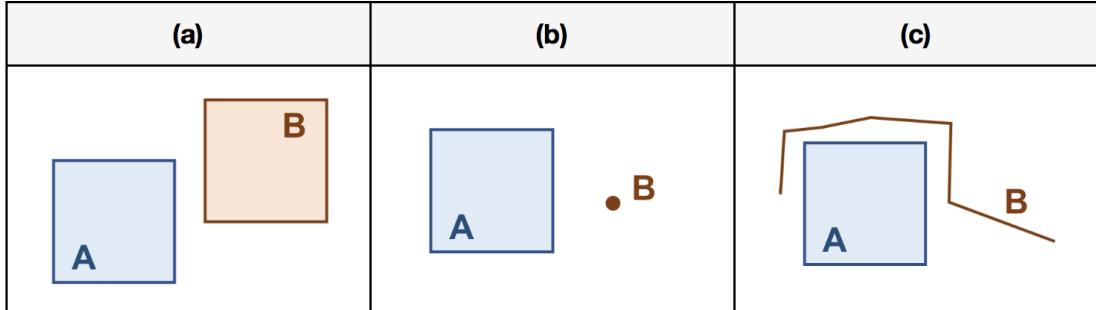


Figura 1.12: *A* e *B* são geometrias espacialmente disjuntas.

$\text{Touches}(\text{Geometry}, \text{Geometry}) \rightarrow \text{Bool}$

Retorna verdadeiro (**True**) se as duas geometrias se tocam. Este relacionamento espacial é definido para os casos onde os pares de objetos geométricos correspondem a A/A, L/L, L/A, P/A e P/L. Este relacionamento não se aplica no caso dos objetos serem do tipo P/P.

Se os pares de objetos satisfizerem um dos três padrões mostrados abaixo, dizemos que a geometria *A* toca a geometria *B*:

		<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>F</i>	<i>T</i>	*	
<i>Fronteira(A)</i>	*	*	*	
<i>Exterior(A)</i>	*	*	*	

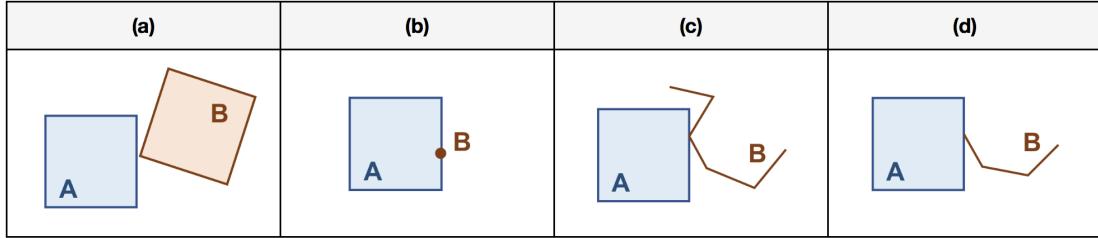
		<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>F</i>	*	*	
<i>Fronteira(A)</i>	<i>T</i>	*	*	
<i>Exterior(A)</i>	*	*	*	

		<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>F</i>	*	*	
<i>Fronteira(A)</i>	*	<i>T</i>	*	
<i>Exterior(A)</i>	*	*	*	

Figura 1.13: DE-9IM para o relacionamento **Touches**.

Os três padrões acima estabelecem que os objetos não podem ter intersecção entre seus interiores, mas podem ter intersecção entre as fronteiras ou entre a fronteira de um e o interior do outro.

A Figura 1.14 apresenta alguns pares de objetos que satisfazem o relacionamento espacial **Touches**, de acordo com os padrões mostrados acima:

Figura 1.14: A e B são geometrias que se tocam.

Crosses(*Geometry*, *Geometry*) \rightarrow *bool*

Retorna verdadeiro (**True**) se a primeira geometria cruza a segunda. Este relacionamento espacial é definido para os casos P/L, P/A, L/L e L/A.

O resultado da intersecção entre duas geometrias deve gerar uma outra de dimensão um nível menor do que a maior dimensão das geometrias envolvidas e os interiores devem ter intersecção.

Nos casos P/L, P/A, ou L/A:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>T</i>	*	<i>T</i>
<i>Fronteira(A)</i>	*	*	*
<i>Exterior(A)</i>	*	*	*

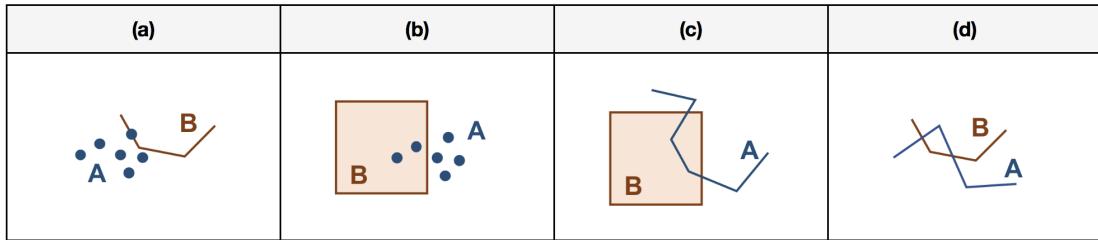
Figura 1.15: DE-9IM para o relacionamento *Crosses* para os casos P/L, P/A, ou L/A.

No caso L/L:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	0	*	*
<i>Fronteira(A)</i>	*	*	*
<i>Exterior(A)</i>	*	*	*

Figura 1.16: DE-9IM para o relacionamento *Crosses* para o caso L/L.

A Figura 1.17 apresenta alguns pares de objetos que satisfazem o relacionamento espacial *Crosses*, de acordo com os padrões mostrados acima:

Figura 1.17: A e B são geometrias que se cruzam.

$Within(Geometry, Geometry) \rightarrow bool$

Retorna verdadeiro (`True`) se a primeira geometria está dentro da segunda e se elas não são iguais.

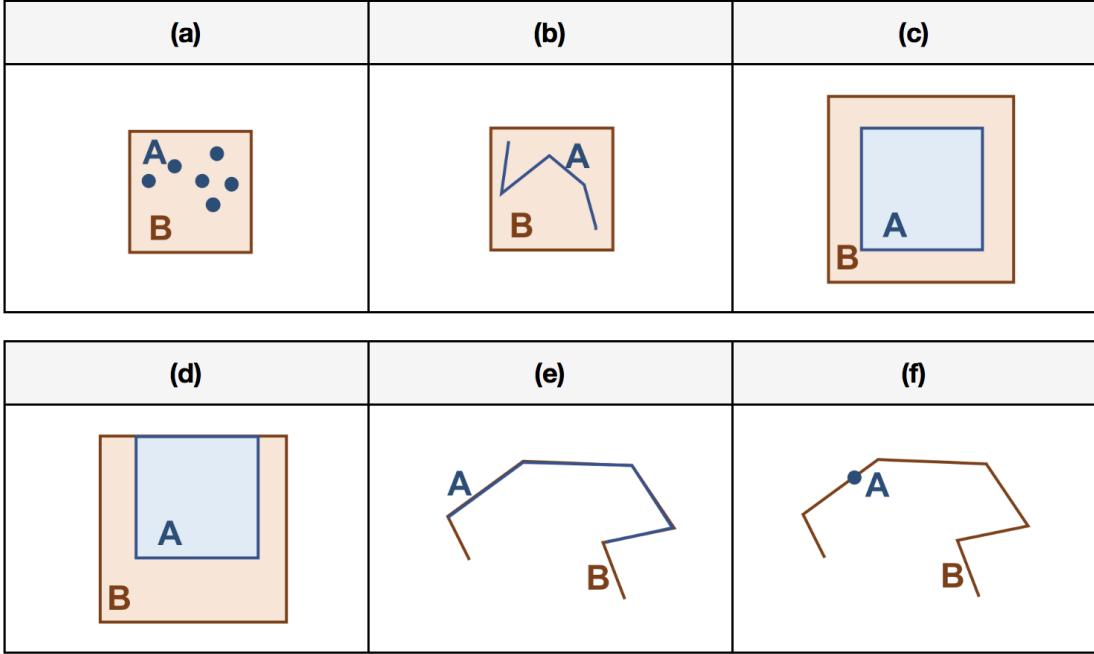
O padrão da matriz para este relacionamento espacial é o seguinte:

		<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>T</i>	*	<i>F</i>	
<i>Fronteira(A)</i>	*	*	<i>F</i>	
<i>Exterior(A)</i>	*	*	*	

Figura 1.18: DE-9IM para o relacionamento `Within`.

A matriz acima define que o predicado `Within` retorna `True` quando os interiores das duas geometrias possuem intersecção e quando o interior e a fronteira do objeto A não têm intersecção com o exterior de B .

A Figura 1.19 apresenta alguns pares de objetos que satisfazem o relacionamento espacial `Within`, de acordo com os padrões mostrados acima:

Figura 1.19: A geometria A está dentro da geometria B .
 $Contains(GeometryA, GeometryB) \rightarrow \text{bool}$

Retorna verdadeiro (`True`) se a primeira geometria contém a segunda e se elas não são iguais. Este relacionamento espacial é o inverso de `Within`, de maneira que temos:

$$Contains(A, B) \iff Within(B, A)$$

A matriz de intersecções do predicado `Contains` é a seguinte:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>T</i>	*	*
<i>Fronteira(A)</i>	*	*	*
<i>Exterior(A)</i>	<i>F</i>	<i>F</i>	*

Figura 1.20: DE-9IM para o relacionamento `Contains`.

Na matriz acima podemos observar que o interior das duas geometrias devem ter intersecção e que a fronteira e interior da segunda (B) não pode ter intersecção com o exterior da primeira (A).

$\text{Overlaps}(\text{Geometry}, \text{Geometry}) \rightarrow \text{bool}$

Retorna verdadeiro (**True**) se a primeira geometria sobrepõe a segunda. Este relacionamento espacial é definido para os casos A/A, L/L e P/P. Em outras palavras, este relacionamento só é válido para geometrias de mesma dimensão e se a intersecção entre elas resultar em um objeto de mesma dimensão.

No caso P/P ou A/A, o predicado **Overlap** retorna **True** se o interior das duas geometria possuem intersecção e se o interior de uma faz intersecção com o exterior da outra e vice-versa. Este relacionamento é definido através do seguinte padrão:

		<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>T</i>	*	<i>T</i>	
<i>Fronteira(A)</i>	*	*	*	
<i>Exterior(A)</i>	<i>T</i>	*	*	

Figura 1.21: DE-9IM para o relacionamento **Overlaps** nos casos P/P ou A/A.

No caso L/L, a intersecção deve resultar em um objeto 1-dimensional. Assim, temos o seguinte padrão de matriz:

		<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	1	*	<i>T</i>	
<i>Fronteira(A)</i>	*	*	*	
<i>Exterior(A)</i>	<i>T</i>	*	*	

Figura 1.22: DE-9IM para o relacionamento **Overlaps** no caso L/L.

A Figura 1.23 apresenta alguns pares de objetos que satisfazem o relacionamento espacial **Overlaps**, de acordo com os padrões mostrados acima:

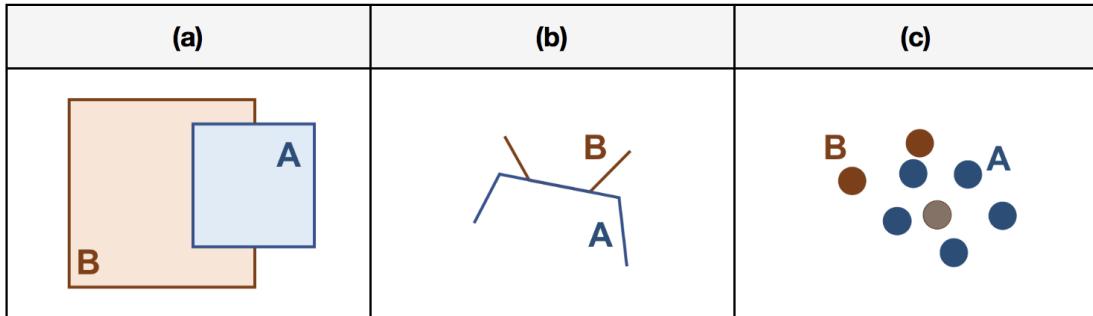


Figura 1.23: A geometria A sobrepõe a geometria B .

Intersects(Geometry, Geometry) → bool

Retorna verdadeiro (`True`) se as duas geometrias tiverem algum tipo de interação espacial, isto é, se os interiores e fronteiras tiverem qualquer intersecção. Este relacionamento espacial é definido como sendo a negação do relacionamento `Disjoint`:

$$\text{Intersects}(A, B) \iff \neg \text{Disjoint}(A, B)$$

Na verdade esse relacionamento diz que as geometrias A e B podem se tocar, ou cruzar, ou sobrepor, ou uma estar contida uma na outra, ou serem iguais. Define-se este relacionamento para facilitar a escrita de expressões com predicados espaciais.

Portanto, o predicado `Intersects` retorna `True` se os interiores das duas geometrias se intersectam:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	<i>T</i>	*	*
<i>Fronteira(A)</i>	*	*	*
<i>Exterior(A)</i>	*	*	*

Figura 1.24: DE-9IM para o relacionamento `Intersects`.

Se as fronteiras das duas geometrias se intersectam:

Se o interior da primeira intersecta a fronteira da segunda:

Ou se a fronteira da primeira intersecta o interior da segunda:

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	*	*	*
<i>Fronteira(A)</i>	*	T	*
<i>Exterior(A)</i>	*	*	*

Figura 1.25: DE-9IM para o relacionamento `Intersects`.

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	*	T	*
<i>Fronteira(A)</i>	*	*	*
<i>Exterior(A)</i>	*	*	*

Figura 1.26: DE-9IM para o relacionamento `Intersects`.

	<i>Interior(B)</i>	<i>Fronteira(B)</i>	<i>Exterior(B)</i>
<i>Interior(A)</i>	*	*	*
<i>Fronteira(A)</i>	T	*	*
<i>Exterior(A)</i>	*	*	*

Figura 1.27: DE-9IM para o relacionamento `Intersects`.

1.3 PostGIS Geometry

1.3.1 Carregando a Extensão PostGIS

Criar um banco de dado `bdgeo` no PostgreSQL:

Listagem 1.3 Para criar um novo banco de dados no PostgreSQL.

```
CREATE DATABASE bdgeo TEMPLATE template1;
```

Carregar a extensão PostGIS no banco criado:

Listagem 1.4 Habilitando a extensão PostGIS.

```
CREATE EXTENSION postgis;
```

Para saber as configurações da sua extensão:

Listagem 1.5 Informações de configurações da sua extensão PostGIS.

```
SELECT postgis_full_version();
```

1.3.2 Well-Known Text (WKT)

Exemplo da representação geométrica para um ponto de coordenadas x = 1 e y = 8:

Listagem 1.6 Representação ponto de coordenadas x=1 e y=8 WKT

```
SELECT ST_GeomFromText('POINT(1 8)');
```

Representação de uma linha definida a partir de três vértices:

Listagem 1.7 Representação Linha três vértices

```
SELECT ST_GeomFromText('LINESTRING(1 5, 3 6, 4 5)');
```

Representação de uma coleção de pontos:

Listagem 1.8 Representação coleção de pontos

```
SELECT ST_GeomFromText('MULTIPOINT(1 8, 3 7, 4 9, 2 9)');
```

Representação de uma coleção de linhas:

Listagem 1.9 Representação coleção de linhas

```
SELECT ST_GeomFromText('MULTILINESTRING( (1 2, 3 3, 4 2),
                                         (4 3, 6 2) )');
```

Representação de uma coleção de polígonos:

Listagem 1.10 Representação coleção de polígonos

```
SELECT ST_GeomFromText('MULTIPOLYGON( ( (1 4, 2 6, 4 5, 3 4, 1 4) ),
                                         ( (1 1, 2 3, 5 4, 5 1, 1 1),
                                             (3 2, 4 3, 4 2, 3 2) ) )');
```

1.3.3 Operadores Métricos

Qual a área e perímetro do polígono abaixo? (**SUBSTITUIR AS IMAGENS***)

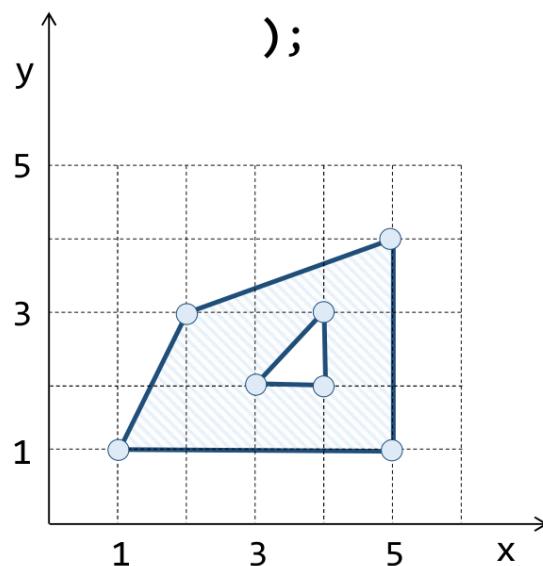


Figura 1.28: Área polígono

Listagem 1.11 Área polígonos

```
SELECT ST_Area(
    'POLYGON( (1 1, 2 3, 5 4, 5 1, 1 1),
                (3 2, 4 3, 4 2, 3 2) )'
);
```

Listagem 1.12 Perímetro polígonos

```
SELECT ST_Perimeter(
    ST_GeomFromText(
        'POLYGON( (1 1, 2 3, 5 4, 5 1, 1 1),
                    (3 2, 4 3, 4 2, 3 2) )'
    )
);
```

Qual o comprimento da linha mostrada na figura abaixo?

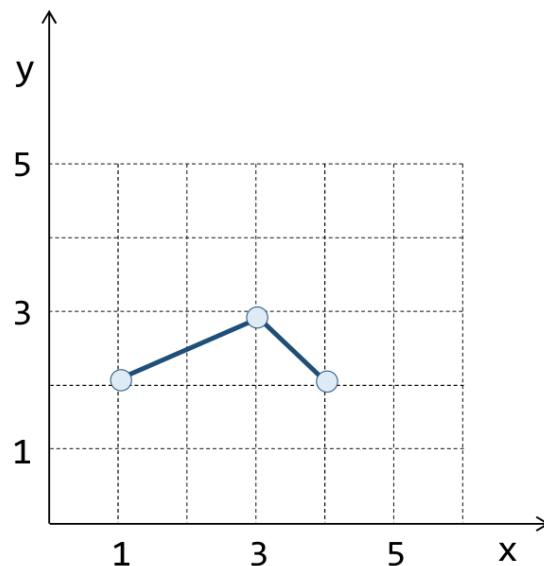


Figura 1.29: Comprimento da linha

Listagem 1.13 Comprimento da linha

```
SELECT ST_Length( 'LINESTRING( 1 2, 3 3, 4 2 )' );
```

Qual a distância entre as geometrias *A* e *B*?

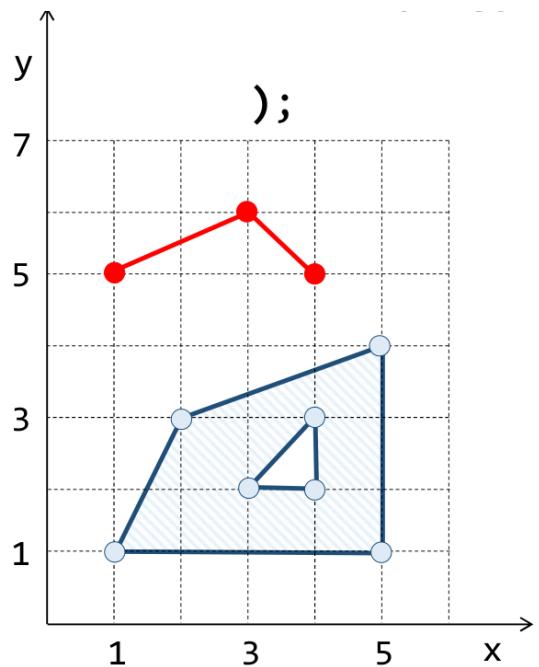


Figura 1.30: Geometria *A* e *B*

Listagem 1.14 Distância entre duas geometrias

```
SELECT ST_Distance(
    'LINESTRING( 1 5, 3 6, 4 5 )',
    'POLYGON( (1 1, 2 3, 5 4, 5 1, 1 1),
                (3 2, 4 3, 4 2, 3 2) )'
);
```

1.4 Tipos Geométricos em Python

Em Python, podemos utilizar as bibliotecas GDAL e Shapely para criação e manipulação de geometrias. A GDAL é uma biblioteca de software livre que fornece uma *camada de abstração de dados geoespaciais* que possibilita o desenvolvimento de aplicações que manipulam dados nos mais diferentes formatos e sistemas. A API (*Application Programming Interface* ou Interface de Programação de Aplicações) desta biblioteca encontra-se disponível para uso em Python através de um *binding* (ou *wrapper*), que fornece acesso às funcionalidades implementadas em C++. A GDAL é basicamente composta de quatro APIs:

- **GDAL:** Voltada para manipulação de dados matriciais (*raster*), com capacidade de leitura e escrita de diversos formatos de imagem de sensoriamento remoto, como GeoTIFF, HDF, e JPEG, entre outros. Esta parte da API contém objetos para manipulação das dimensões de uma imagem, para acesso de leitura e escrita de blocos nas bandas espectrais de uma imagem, acesso a metadados e manipulação de pirâmides de multi-resolução.
- **OGR:** Parte da API voltada para manipulação de dados em formatos vetoriais, tais como *ESRI Shapefile*, *Google KML* e *GeoJSON*. Apresenta os conceitos de camada de informação, feições, atributos alfanuméricos e geométricos.
- **OSR:** Voltada para a manipulação de projeções e sistemas de referência espacial.
- **GNM:** Acrônimo de *Geographic Network Model*, esta parte da API serve ao propósito de manipulação de redes geográficas.

A parte da biblioteca GDAL voltada para manipulação de dados vetoriais conhecida por OGR contém os tipos geométricos da OGC-SFS mostrados na Figura 1.2 (Seção 1.1). As operações espaciais disponíveis sobre os tipos geométricos da GDAL são implementadas através da biblioteca C++ GEOS (*Geometry Engine Open Source*). A GEOS implementa todos os operadores espaciais definidos na OGC-SFS, com destaque para as operações topológicas. Além disso, ela possui outras operações, como triangulação, diagrama de voronoi e estruturas de dados espaciais como árvores-R. Consequentemente, a GEOS é usada em diversos projetos de software livre, como TerraView, QGIS, PostGIS e muitos outros.

Apesar da GDAL ser uma biblioteca robusta e completa, sua API Python expõe os tipos e operações com um estilo de programação muito próximo da linguagem C. Por isso, existem outras bibliotecas em Python que tentam fornecer um estilo de programação mais próximo do universo Python.

A biblioteca Shapely é uma dessas bibliotecas, que fornece especificamente o modelo geométrico e as operações espaciais da OGC-SFS num estilo "*Pythonico*". Vale ressaltar que a Shapely utiliza a GEOS para representação dos tipos geométricos bem como para as operações espaciais.

Esta seção utilizará a biblioteca Shapely para ilustrar a manipulação de geometrias no espaço cartesiano com a linguagem Python.

1.4.1 Instalação

Para realizar a instalação da biblioteca `shapely` com o gerenciador de pacotes da Anaconda, ative o ambiente virtual no qual deseja instalar essa biblioteca e em seguida instale o pacote com o mesmo nome da biblioteca, como mostrado abaixo:

Listagem 1.15 Instalação da biblioteca shapely.

```
conda activate geospatial  
conda install shapely
```

Você deverá instalar a versão 1.6.4 ou superior.

Para verificar se sua instalação funcionou, importe a biblioteca e verifique a versão:

Listagem 1.16 Importando a biblioteca shapely.

```
import shapely  
print(shapely.__version__)
```

1.4.2 Tipos Geométricos

A biblioteca Shapely suporta os tipos geométricos `Point`, `LineString`, `LinearRing`, `Polygon`, `MultiPoint`, `MultiLineString`, `MultiPolygon` e `GeometryCollection` (Figura 1.2).

Como a biblioteca Shapely utiliza a GEOS, as operações suportadas não consideram a componente z em suas análises. Além disso, os tipos geométricos dessa biblioteca utilizam números em ponto flutuante para representação das coordenadas dos seus elementos geométricos.

Pontos (`Point`)

Para criar objetos do tipo `Point` é preciso importar esse tipo a partir da biblioteca `shapely` submódulo `geometry`:

Listagem 1.17 Importando a classe `Point`.

```
from shapely.geometry import Point
```

O construtor de um objeto `Point` aceita um par de valores `x`, `y` e, opcionalmente, um valor de `z`. Outra possibilidade é construir um ponto utilizando uma tupla:

Listagem 1.18 Construtor de um objeto `Point`.

```
pt = Point(5.0, 3.0)

pt = Point( ( 10.0, 5.0) )
```

As coordenadas de um ponto podem ser acessadas através do atributos `x` e `y`:

Listagem 1.19 Acessando as coordenadas do Point.

```
pt.x
```

```
pt.y
```

Comprimento do Ponto:

Listagem 1.20 Comprimento de um Point.

```
pt.length
```

Área do Ponto:

Listagem 1.21 Área de um Point.

```
pt.area
```

Fronteira de um ponto :

Listagem 1.22 Fronteira de um Point.

```
pt.boundary
```

Linhas (LineString)

Assim como o tipo Point, o tipo LineString deve ser importado da seguinte forma:

Listagem 1.23 Importando a classe LineString.

```
from shapely.geometry import LineString
```

O construtor de um objeto `LineString` aceita como argumento uma sequência de tuplas (x, y) ou (x, y, z) :

Listagem 1.24 Construtor de um objeto `LineString`.

```
line = LineString( [ (0, 0), (5, 2), (10, 9), (18, 10) ] )
```

As coordenadas x e y dos vértices da linha podem ser acessados na forma de *arrays*:

Listagem 1.25 Acessando as coordenadas da `LineString`.

```
line.xy
```

Comprimento, área e fronteira da linha podem ser acessadas respectivamente:

Listagem 1.26 Comprimento, área e fronteira da `LineString`.

```
line.length  
line.area  
line.boundary
```

Para acessar os elementos de uma linha podemos utilizar o atributo `coords`:

Listagem 1.27 Acessando as coordenadas da `LineString`.

```
for c in line.coords:  
    print(c)
```

Podemos usar a notação de `slice` de sequências com objetos do tipo `LineString`:

Listagem 1.28 Acessando as coordenadas da LineString.

```
line.coords[0:2]  
  
line.coords[1:]
```

Anel (LinearRing)

Para criar objetos do tipo `LinearRing` é preciso importar a classe `LinearRing`:

Listagem 1.29 Importando a classe LinearRing.

```
from shapely.geometry import LinearRing
```

O construtor de um objeto `LinearRing` aceita como argumento uma sequência de tuplas (x, y) ou (x, y, z) :

Listagem 1.30 Construtor de um objeto LinearRing.

```
anel = LinearRing( [ (0, 0), (10, 0), (10, 10), (0, 10), (0, 0) ] )
```

Observação: a sequência informada pode ser explicitamente fechada como no caso acima ou pode ser aberta. Neste último caso a sequência será implicitamente fechada.

Comprimento, área e fronteira da `LinearRing` podem ser acessadas respectivamente:

Listagem 1.31 Comprimento, área e fronteira de um objeto LineString.

```
anel.length  
  
anel.area  
  
anel.boundary
```

Polígonos (Polygon)

Para criar objetos do tipo `Polygon` é preciso importar a classe `Polygon`:

Listagem 1.32 Importando a classe `Polygon`.

```
from shapely.geometry import Polygon
```

O construtor de um objeto `Polygon` aceita dois argumentos. O primeiro, obrigatório, é uma sequência de tuplas (x, y) ou (x, y, z) que representa o anel externo do polígono. O segundo, opcional, é uma sequência de anéis e representa os anéis internos do polígono:

Listagem 1.33 Construtor de um objeto `Polygon`.

```
anel_externo = [ (0, 0), (10, 0), (10, 10), (0, 10), (0, 0) ]
anel_interno = [ (3, 3), (7, 3), (7, 7), (3, 7), (3, 3) ]
poly = Polygon(anel_externo, [anel_interno])
```

O anel externo pode ser acessado através do operador `exterior` e os anéis internos podem ser acessados através da propriedade `internors`:

Listagem 1.34 Anel externo e anéis internos de um objeto `Polygon`.

```
poly.exterior
poly.internors
poly.internors[0]
poly.internors[0].xy
```

Comprimento, área e fronteira de um `Polygon` podem ser acessadas respectivamente:

Listagem 1.35 Comprimento, área e fronteira de um objeto Polygon.

```
poly.length  
poly.area  
poly.area
```

Coleção de Pontos (MultiPoint)

Para criar objetos que representam coleções homogêneas de pontos é preciso importar a classe `MultiPoint`:

Listagem 1.36 Importando a classe MultiPoint.

```
from shapely.geometry import MultiPoint
```

O construtor de um `MultiPoint` recebe como argumento uma sequência de valores (x, y) ou (x, y, z):

Listagem 1.37 Construtor de um objeto MultiPoint.

```
mpt = MultiPoint( [ (0, 0), (5, 5), (10, 0), (10, 10), (0, 10) ] )
```

Os elementos das coleção podem ser acessados através da propriedade `geoms`:

Listagem 1.38 Propriedade geom MultiPoint.

```
for pt in mpt.geoms:  
    print(pt.x, pt.y)
```

Coleção de Linhas (MultiLineString)

Para criar objetos que representam coleções homogêneas de linhas é preciso importar a classe `MultiLineString`:

Listagem 1.39 Importando a classe MultiPoint.

```
from shapely.geometry import MultiLineString
```

O construtor de uma `MultiLineString` recebe como argumento uma sequência de linhas:

Listagem 1.40 Construtor de um objeto MultiLineString.

```
mline = MultiLineString( [ [ (0, 0), (8, 2), (13, 9) ],
                           [ (21, 11), (30, -1) ] ] )
```

Os elementos das coleção podem ser acessados através da propriedade `geoms`:

Listagem 1.41 Propriedade geom MultiLineString

```
for line in mline.geoms:
    print(line)
```

Coleção de Polígonos (MultiPolygon)

Para criar objetos que representam coleções homogeneas de polígonos é preciso importar a classe `MultiPolygon`:

Listagem 1.42 Importando a classe MultiPolygon.

```
from shapely.geometry import MultiPolygon
```

O construtor de um MultiPolygon recebe como argumento uma sequência de polígonos:

Listagem 1.43 Construtor de um objeto MultiPolygon.

```
mpoly = MultiPolygon(
    [
        [
            [
                [ (0, 0), (16, 0), (16, 10), (0, 10), (0, 0) ],
                [
                    [ (3, 1), (7, 1), (5, 7), (3, 1) ],
                    [ (8, 1), (12, 1), (10, 9), (8, 1) ]
                ]
            ],
            [
                [
                    [ (20, 0), (25, 0), (22, 10), (20, 0) ],
                    []
                ]
            ]
        ]
    )
)
```

O mesmo polígono acima pode ser construído de forma mais clara criando os polígonos intermediários e depois criando a coleção:

Listagem 1.44 Construtor de um objeto MultiPolygon com polígonos intermediários.

```
shell_1 = LinearRing( [ (0, 0), (16, 0), (16, 10), (0, 10), (0, 0) ] )

hole_11 = LinearRing( [ (3, 1), (7, 1), (5, 7), (3, 1) ] )
hole_12 = LinearRing( [ (8, 1), (12, 1), (10, 9), (8, 1) ] )

poly_1 = Polygon( shell_1, [ hole_11, hole_12 ] )

# Definição do segundo polígono
shell_2 = LinearRing( [ (20, 0), (25, 0), (22, 10), (20, 0) ] )

poly_2 = Polygon(shell_2)

mpoly = MultiPolygon( [ poly_1, poly_2 ] )
```

Capítulo 2

Programação do Lado Servidor no PostgreSQL

Atualmente, os SGBD-R são muito mais do que apenas sistemas de armazenamento e gerenciamento de dados, correspondendo a verdadeiras plataformas de desenvolvimento de sistemas com funcionalidades que vão muito além da simples execução de comandos SQL (DDL ou DML). Sistemas como Oracle [3], IBM DB2 [4] e PostgreSQL [5] são verdadeiros *frameworks* de desenvolvimento de aplicações.

Em comum, esses SGBDs permitem construir *funções* ou *procedimentos* que são armazenados dentro do próprio SGBD¹ e que são capazes de executar tarefas com alto grau de complexidade. Essas funções podem ser expressas em linguagens de programação tradicionais, como C, Java, Python ou até mesmo em R. No entanto, a maneira mais usual de expressarmos essas funções é através de linguagens procedurais baseadas em dialetos SQL, como as linguagens PL/pgSQL do PostgreSQL e a PL/SQL da Oracle.

Outra funcionalidade disponível nesses SGBDs é a definição de *ações* que sejam invocadas automaticamente sempre que modificarmos o conteúdo de uma tabela. Essas *ações* ou *triggers* são expressas, em geral, da mesma forma que os procedimentos armazenados, em um dialeto SQL. Embora em PostgreSQL seja possível criar *triggers* na Linguagem C.

¹Essas funções são chamadas de *Procedimentos Armazenados* ou *Funções Definidas pelo Usuário* (*User Defined Functions* ou UDF)

Porém, uma das capacidades menos exploradas nos cursos de bancos de dados, mas fundamental para o desenvolvimento de certos tipos de aplicações, como as geográficas, é a de criação de *extensões* do SGBD. As *extensões* funcionam como uma espécie de *plugin*, possibilitando a inclusão de novos tipos de dados, novas funções e operadores sobre esses novos tipos, extensão dos mecanismos de indexação existentes para os novos tipos e, integração de bibliotecas de software construídas em outras linguagens. No caso do PostgreSQL este mecanismo é responsável por possibilitar a criação da extensão geográfica PostGIS.

Este capítulo irá mostrar como criar novas funcionalidades para um servidor de bancos de dados PostgreSQL. Começaremos pela criação de funções escritas em PL/pgSQL que serão executadas dentro do processo do servidor de bancos de dados, quando invocadas por aplicações clientes. Em seguida, iremos trabalhar com a criação de *triggers*, que servirão ao propósito de materializar relacionamentos espaciais entre objetos geográficos. Por fim, iremos explorar um dos mecanismos mais poderosos do PostgreSQL, que é a sua capacidade de criação de extensões.

2.1 PL/pgSQL

A Linguagem PL/pgSQL² permite que comandos SQL sejam executados dentro de uma linguagem procedural, isto é, de uma linguagem com comandos de decisão (*if-then-else*) e repetição (laços *do-while*), e que facilita o trabalho de manipulação do resultado da execução de consultas.

²PL vem de *Programming Language*.

A sintaxe geral de funções em PL/pgSQL é a seguinte^{3,4}:

```
CREATE OR REPLACE FUNCTION nome-da-função(parâmetros)
RETURNS tipo-retorno
AS
$$
[<<rótulo>>]
[DECLARE
    lista-variáveis;]
BEGIN
    comandos;
[EXCEPTION
    [WHEN condição THEN
        comandos;
    ...]]
END [rótulo];
$$
LANGUAGE plpgsql;
```

³Para maiores detalhes da sintaxe da linguagem PL/pgSQL, consulte [6].

⁴As partes entre [e] são opcionais.

Vamos começar criando uma função chamada `my_distance` capaz de computar a distância euclideana entre dois pontos. O código desta função é mostrado a seguir:

```
CREATE OR REPLACE FUNCTION my_distance(first GEOMETRY,
                                         second GEOMETRY)
RETURNS NUMERIC
AS
$$
DECLARE
    dx NUMERIC DEFAULT 0.0;
    dy NUMERIC DEFAULT 0.0;
    d  NUMERIC DEFAULT 0.0;
BEGIN
    dx := ST_X(first) - ST_X(second);
    dy := ST_Y(first) - ST_Y(second);

    d := sqrt(power(dx, 2) + power(dy, 2));

    RETURN d;
END;
$$
LANGUAGE plpgsql;
```

As funções PL/pgSQL podem ser usadas em qualquer parte das consultas SQL onde uma função comum possa ser incluída. Uma vez criada, uma função pode ser chamada de dentro de uma consulta submetida tanto do lado cliente quanto do lado servidor. No entanto, a execução da função ocorrerá sempre do lado servidor.

Para executar a função `my_distance` podemos utilizar uma consulta do tipo `SELECT` usando uma das duas construções:

```
SELECT my_distance(
    ST_GeometryFromText('POINT(0 0)', 4326),
    ST_GeometryFromText('POINT(1 1)', 4326));
```

ou:

```
SELECT *
FROM my_distance(
    ST_GeometryFromText('POINT(0 0)', 4326),
    ST_GeometryFromText('POINT(1 1)', 4326));
```

2.1.1 Comentários

Uma boa prática de programação, consiste em documentar as funções. Em PL/pgSQL, os comentários de linha começam com um duplo hífen (-), enquanto comentários de bloco utilizam as marcações /* e */.

O trecho de código abaixo mostra como incluir alguns comentários na definição da função `my_distance`.

```
/*
Descrição: Função que computa a distância euclideana
entre dois pontos.

Parâmetros:
- first: Ponto no espaço bidimensional.
- second: Ponto no espaço bidimensional.

Retorno: um valor numérico que representa a distância
euclideana entre os pontos informados.

*/
CREATE OR REPLACE FUNCTION my_distance(first GEOMETRY,
                                         second GEOMETRY)
RETURNS NUMERIC
AS
$$
DECLARE
    dx NUMERIC DEFAULT 0.0;
    dy NUMERIC DEFAULT 0.0;
    d  NUMERIC DEFAULT 0.0;
```

```

BEGIN
    dx := ST_X(first) - ST_X(second);
    dy := ST_Y(first) - ST_Y(second);

    -- sqrt: operação raiz quadrada
    d := sqrt(power(dx, 2) + power(dy, 2));

    RETURN d;
END;
$$
LANGUAGE plpgsql;

```

2.1.2 Estruturas Condicionais

Em PL/pgSQL temos dois tipos de comandos condicionais: *if-then-else* e *case-when*.

A sintaxe de comandos *if-then-else* é a seguinte:

```

IF condição THEN
    comandos;
[ELSIF condição THEN
    comandos;]
[ELSE
    comandos;]
END IF;

```

Comandos *case-when* possuem a seguinte sintaxe:

```

CASE expressão
    WHEN expressão [, expressão [ ... ]] THEN
        comandos;
    [WHEN expressão [, expressão [ ... ]] THEN
        comandos;
    ...
]
```

```
[ELSE
    comandos;]
END CASE;
```

No exemplo da função `my_distance`, se considerarmos que a distância entre os pontos só pode ser computada caso eles se encontrem no mesmo sistema de referência espacial, poderíamos reescrever a função da seguinte forma:

```
CREATE OR REPLACE FUNCTION my_distance(first GEOMETRY,
                                         second GEOMETRY)
RETURNS NUMERIC
AS
$$
DECLARE
    dx NUMERIC DEFAULT 0.0;
    dy NUMERIC DEFAULT 0.0;
    d  NUMERIC DEFAULT 0.0;
BEGIN
    IF(ST_SRID(first) <> ST_SRID(second)) THEN
        return NULL;
    END IF;

    dx := ST_X(first) - ST_X(second);
    dy := ST_Y(first) - ST_Y(second);

    d := sqrt(power(dx, 2) + power(dy, 2));

    RETURN d;
END;
$$
LANGUAGE plpgsql;
```

Repare que agora, se chamarmos a função para um par de pontos em diferentes sistemas de coordenadas, receberemos um valor nulo (`NULL`) como retorno:

```
SELECT my_distance(
    ST_GeometryFromText('POINT(0 0)', 4326),
    ST_GeometryFromText('POINT(1 1)', 29193));
```

No entanto, ao invés de retornarmos um valor `NULL` o mais adequado seria interromper a execução da função indicando algum tipo ou condição de erro. A seção a seguir mostra como podemos proceder nestes casos.

2.1.3 Mensagens e Exceções

O comando `RAISE` pode ser usado para reportar mensagens, que são associadas a níveis de severidade: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, e `EXCEPTION`.

O nível `EXCEPTION` lança uma exceção (ou erro), que aborta a execução da função, abortando também a transação na qual a função se encontra. Os outros cinco níveis apenas emitem uma mensagem com diferentes prioridades⁵.

Vamos utilizar o comando `RAISE` junto com testes condicionais para verificar se os argumentos informados à função `my_distance` correspondem ao tipo geométrico `ST_Point` e, se esses pontos encontram-se no mesmo sistema de referência espacial. A nova definição da função `my_distance` pode ser vista abaixo.

```
CREATE OR REPLACE FUNCTION my_distance(first GEOMETRY,
                                         second GEOMETRY)
RETURNS NUMERIC
AS $$

DECLARE
    dx NUMERIC DEFAULT 0.0;
    dy NUMERIC DEFAULT 0.0;
    d  NUMERIC DEFAULT 0.0;

BEGIN
    IF((ST_GeometryType(first) <> 'ST_Point') OR
```

⁵Existem variáveis de configuração do servidor que controlam se as mensagens são escritas no *log* do próprio servidor. Veja no manual do PostgreSQL as variáveis `log_min_messages` e `client_min_messages` para maiores detalhes

```

(ST_GeometryType(second) <> 'ST_Point')) THEN
    RAISE EXCEPTION 'Tipos geométricos inválidos!';
END IF;

IF(ST_SRID(first) <> ST_SRID(second)) THEN
    RAISE EXCEPTION 'Pontos com SRIDs diferentes!';
END IF;

dx := ST_X(first) - ST_X(second);
dy := ST_Y(first) - ST_Y(second);

RAISE NOTICE 'dx: %', dx;
RAISE NOTICE 'dy: %', dy;

d := sqrt(power(dx, 2) + power(dy, 2));

RETURN d;
END;
$$
LANGUAGE plpgsql;

```

Desta vez, se não respeitarmos os tipos geométricos, a função irá produzir uma mensagem de erro e a função será interrompida:

```

SELECT my_distance(
    ST_GeometryFromText('LINESTRING(0 0, 1 1)', 4326),
    ST_GeometryFromText('POINT(1 1)', 4326));

```

ERROR: Tipos geométricos inválidos!

Da mesma forma, se informarmos pontos com diferentes SRIDs, uma mensagem será produzida informando o erro e a função será interrompida:

```
SELECT my_distance(
```

```
ST_GeometryFromText('POINT(0 0)', 29193),
ST_GeometryFromText('POINT(1 1)', 4326));
```

ERROR: Pontos com SRIDs diferentes!

Caso a função seja chamada com argumentos válidos, além do resultado, será produzida uma mensagem (NOTICE):

```
SELECT my_distance(
  ST_GeometryFromText('POINT(0 0)', 4326),
  ST_GeometryFromText('POINT(1 1)', 4326));
```

```
NOTICE: dx: -1
NOTICE: dy: -1
my_distance
-----
1.4142135623730950
(1 row)
```

Outro detalhe a ser notado no exemplo é o uso do caractere % dentro das mensagens:

```
RAISE NOTICE 'dx: %', dx;
RAISE NOTICE 'dy: %', dy;
```

No comando RAISE, o texto das mensagens podem conter o caractere especial %, que é substituído pelo valor de um literal ou variável informados após o texto da mensagem (neste caso as variáveis *dx* e *dy*).

Os níveis de severidade também estão associados a alguns comportamentos importantes. Por exemplo, RAISE NOTICE envia mensagem imediatamente, assim que a linha é executada. Já o comando RAISE NOTIFY aguarda a transação que contém a execução da função terminar, de forma que a mensagem pode não ser mostrada se a transação falhar e for desfeita (*rollback*).

2.1.4 Tipos dos Parâmetros e Tipo de Retorno

Todos os tipos de dados e operadores disponíveis em SQL também encontram-se disponíveis para a construção de funções PL/pgSQL. Desta forma, o tipo de um parâmetro de função e o tipo de valor de retorno pode ser qualquer um dos tipos manipulados pelo SGBD, incluindo: (a) tipos primitivos como NUMERIC, TEXT, TIMESTAMP; (b) tipos compostos, como os tipos definidos pelas estruturas das tabelas ou por comandos CREATE TYPE; (c) tipo registro (RECORD); (d) tipo SETOF; (e) tipo TABLE; e (f) tipos definidos pelo usuário, como os tipos GEOMETRY, GEOGRAPHY e RASTER do PostGIS. Funções que não retornam valores podem ser definidas com o tipo de retorno VOID.

Os parâmetros das funções são nomeados com os identificadores \$1, \$2, ..., \$n. Nos exemplos anteriores, usamos nomes alternativos (ou *alias*) para renomear os parâmetros da função `my_distance`: `first` e `second`. Sem o uso de parâmetros nomeados, essa função seria definida da seguinte forma⁶:

```
CREATE OR REPLACE FUNCTION my_distance(GEOMETRY, GEOMETRY)
RETURNS NUMERIC
AS
$$
DECLARE
    dx NUMERIC DEFAULT 0.0;
    dy NUMERIC DEFAULT 0.0;
    d  NUMERIC DEFAULT 0.0;
BEGIN
    IF((ST_GeometryType($1) <> 'ST_Point') OR
       (ST_GeometryType($2) <> 'ST_Point')) THEN
        RAISE EXCEPTION 'Tipos geométricos inválidos!';
    END IF;

    IF(ST_SRID($1) <> ST_SRID($2)) THEN
        RAISE EXCEPTION 'Pontos com SRIDs diferentes!';
    END IF;
```

⁶Para remover a função `my_distance`, use o comando:
`DROP FUNCTION my_distance(geometry,geometry);`

```

dx := ST_X($1) - ST_X($2);
dy := ST_Y($1) - ST_Y($2);

d := sqrt(power(dx, 2) + power(dy, 2));

RETURN d;
END;
$$
LANGUAGE plpgsql;

```

Observação:

- Podemos declarar funções com o mesmo nome, porém essas funções deverão ter diferentes parâmetros. Essa capacidade é conhecida por *sobrecarga de funções* (*function overloading*).
- A saída de uma função pode ser realizada tanto na forma de valores de retorno quanto como parâmetros de saída (**OUT**). Consulte [6] para maiores informações sobre como trabalhar com parâmetros de saída.

2.1.5 Execução de Comandos SQL

Podemos utilizar comandos SQL com instruções válidas dentro de funções PL/pgSQL. Esses comandos podem utilizar nomes de variáveis definidas na própria função para compor o comando.

Para ilustrar a execução de comandos, vamos criar uma tabela chamada **pluviometros**:

```

CREATE TABLE pluviometros
(
    gid      SERIAL PRIMARY KEY,
    location GEOMETRY(POINT, 4326)
);

```

Comandos que não retornam resultados

Vamos criar uma função em PL/pgSQL que faça a inserção na tabela `pluviometros` tomando como entrada os valores de longitude e latitude onde se encontra instalado um pluviômetro:

```
CREATE OR REPLACE FUNCTION my_insert(longitude NUMERIC,
                                     latitude  NUMERIC)
RETURNS VOID
AS
$$
BEGIN

    INSERT INTO pluviometros(location)
    VALUES(
        ST_SetSRID(
            ST_MakePoint(longitude, latitude),
            4326));

    RETURN;
END;
$$
LANGUAGE plpgsql;
```

Agora, podemos chamar a função `my_insert` que irá montar a tupla a ser inserida na tabela `pluviometros`:

```
SELECT my_insert(-45.8872, -23.1791);
```

Repare que o comando `INSERT` sem a cláusula `RETURNING` não retorna nenhum resultado. Para realizar a avaliação de expressões ou consultas do tipo `SELECT` descartando o resultado, pode-se utilizar o comando `PERFORM`. Este comando é necessário, principalmente, quando se tem uma consulta do tipo `SELECT` cuja finalidade é apenas produzir um efeito colateral (*side-effect*).

```
PERFORM SELECT * FROM pluviometros ORDER BY gid;
```

Comandos que retornam uma única tupla como resultado

Podemos também recuperar o resultado de uma consulta que retorne uma única tupla, ou seja, um resultado que pode ser composto de um ou mais valores.

Como exemplo desse tipo de comando, vamos alterar a função `my_insert` para que ela retorne como resultado o valor gerado automaticamente pela sequência (SEQUENCE) associada à coluna `gid`, que é a chave primária da tabela `pluviometros`⁷.

```
CREATE OR REPLACE FUNCTION my_insert(longitude NUMERIC,
                                      latitude NUMERIC)
RETURNS pluviometros.gid%TYPE
AS
$$
DECLARE
    id pluviometros.gid%TYPE;
BEGIN

    INSERT INTO pluviometros(location)
    VALUES(
        ST_SetSRID(
            ST_MakePoint(longitude, latitude),
            4326))
    RETURNING gid INTO id;

    RETURN id;
END;
$$
LANGUAGE plpgsql;
```

Com essa modificação, ao invocarmos a função `my_insert` obtemos o novo identificador da linha criada:

⁷Provavelmente você terá que remover a definição da função `my_insert` realizada anteriormente. Para tal, utilize o seguinte comando:
`DROP FUNCTION my_insert(numeric,numeric);`

```
SELECT my_insert(-43.6419, -20.393);

my_insert
-----
2
(1 row)
```

Repare na declaração do tipo de retorno da função e no tipo da variável `id`. Utilizamos `%TYPE` para deduzir o tipo através de uma referência ao nome da coluna `gid` da tabela `pluviometros`.

Na prática, podemos associar o resultado de comandos SQL que produzam uma única tupla, a uma variável do tipo `RECORD`, tipos compostos (como os tipos definidos por tabelas e `%ROWTYPE`) ou listas de variáveis compatíveis com os tipos retornados. A sintaxe de cada comando é a seguinte:

```
SELECT ... INTO [STRICT] variável[, variável ...] FROM ...;
INSERT ... RETURNING expressões INTO [STRICT] variável[, variável ...];
UPDATE ... RETURNING expressões INTO [STRICT] variável[, variável ...];
DELETE ... RETURNING expressões INTO [STRICT] variável[, variável ...];
```

Caso a palavra `STRICT` seja omitida em comandos `SELECT`, a variável será associada ao primeiro valor retornado pelo comando ou a um valor nulo. Se a palavra `STRICT` for utilizada, a consulta deverá retornar uma única linha, caso contrário, será produzida uma exceção que irá interromper a execução da função. Para os comandos `INSERT`, `UPDATE` e `DELETE` usando a cláusula `RETURNING`, mesmo omitindo a palavra `STRICT`, será lançada uma exceção se mais de uma tupla for retornada⁸.

Uma variável que é automaticamente definida quando executamos um comando que retorna alguma tupla como resultado, é a variável `FOUND`. No caso de comandos que não usam a palavra `STRICT`, a próxima instrução da função pode verificar o valor desta variável

⁸A explicação dada pelo time de desenvolvimento do PostgreSQL é que neste caso não há a possibilidade de usar uma cláusula como `ORDER BY` que possibilite a escolha da linha afetada a ser considerada.

para saber se alguma tupla foi ou não retornada. Por exemplo, se quisermos saber se existe algum pluviômetro a uma certa distância de uma dada localização, podemos construir uma função como a seguinte⁹:

```

CREATE OR REPLACE FUNCTION existe_pluviometro(location GEOMETRY,
                                               distance NUMERIC)
RETURNS BOOL
AS
$$
DECLARE
    pluviometro pluviometros%ROWTYPE;
BEGIN
    SELECT * INTO pluviometro
    FROM pluviometros
    WHERE ST_DWithin(pluviometros.location,
                      existe_pluviometro.location,
                      distance);

    IF FOUND THEN
        RAISE NOTICE
            'Encontrado pelo menos um pluviômetro próximo: %',
            ST_AsText(pluviometro.location);

        RETURN TRUE;
    ELSE
        RAISE NOTICE
            'Não foi encontrado um único pluviômetro nas proximidades de: %',
            ST_AsText(existe_pluviometro.location);

        RETURN FALSE;
    END IF;
END;
$$

```

⁹Obviamente não precisamos de uma função para isso, além de podermos escrever uma consulta de agregação com o operador COUNT(*) .

```
LANGUAGE plpgsql;
```

Se quisermos saber da existência ou não de um pluviômetro num raio de 1.0 grau das localizações (-45, -23) e (-47, -25), podemos realizar a seguinte consulta:

```
SELECT existe_pluviometro(ST_SetSRID(ST_MakePoint(-45, -23), 4326), 1.0);
```

```
existe_pluviometro
-----
t
(1 row)
```

```
SELECT existe_pluviometro(ST_SetSRID(ST_MakePoint(-47, -25), 4326), 1.0);
```

```
NOTICE: Não foi encontrado um único pluviômetro nas proximidades de: POINT(-47 -25)
existe_pluviometro
-----
f
(1 row)
```

É muito comum usarmos a variável FOUND junto com testes condicionais para lançar uma exceção:

```
...
IF NOT FOUND THEN
    RAISE EXCEPTION 'mensagem: nenhuma linha encontrada!';
END IF;
...
```

No caso de comandos que usam a palavra STRICT, existem outras duas variáveis muito úteis:

- NO_DATA_FOUND: indica que nenhuma tupla foi retornada.
- TOO_MANY_ROWS: indica que mais do que uma tupla foi retornada.

Podemos utilizar um bloco para capturar os possíveis tipos de exceções, como mostrado na função abaixo, que verifica se existe apenas um único pluviômetro a uma certa distância de uma dada localização:

```

CREATE OR REPLACE FUNCTION unico_pluviometro(location GEOMETRY,
                                              distance NUMERIC)
RETURNS BOOL
AS
$$
DECLARE
    pluviometro pluviometros%ROWTYPE;
BEGIN
    SELECT * INTO STRICT pluviometro
        FROM pluviometros
        WHERE ST_DWithin(pluviometros.location,
                         unico_pluviometro.location,
                         distance);
    RETURN TRUE;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE NOTICE 'Nenhum pluviômetro encontrado próximo a: %',
                      ST_AsText(unico_pluviometro.location);
        RETURN FALSE;
    WHEN TOO_MANY_ROWS THEN
        RAISE NOTICE 'Vários pluviômetros encontrados próximo a: %',
                      ST_AsText(unico_pluviometro.location);
        RETURN FALSE;
END;
$$
LANGUAGE plpgsql;

```

Para testar a função acima, execute os comandos abaixo:

```
SELECT unico_pluviometro(ST_SetSRID(ST_MakePoint(-45, -23), 4326), 1.0);
```

```
unico_pluviometro
-----
t
(1 row)
```

```
SELECT unico_pluviometro(ST_SetSRID(ST_MakePoint(-47, -25.), 4326), 1.0);
```

```
unico_pluviometro
-----
f
(1 row)
```

Observação: atente-se ao fato de que a variável FOUND é sempre associada ao valor TRUE quando STRICT é utilizada e o comando retorna uma única tupla.

Comandos Dinâmicos

Outra facilidade para criação de funções em PL/pgSQL é a possibilidade de construirmos dinamicamente uma string representando um comando SQL para posterior execução. Vamos ilustrar o uso do comando EXECUTE com um exemplo que irá posicionar um pluviômetro em uma localização aleatória.

```
CREATE OR REPLACE FUNCTION random_insert()
RETURNS pluviometros.gid%TYPE
AS
$$
DECLARE
    id pluviometros.gid%TYPE;
    r NUMERIC;
    longitude NUMERIC;
    latitude NUMERIC;
```

```

pt GEOMETRY;
query TEXT;
BEGIN
    query := 'INSERT INTO pluviometros (location) VALUES($1) RETURNING gid';

    r := random();

    longitude := 360.0 * r - 180.0;
    latitude := 180.0 * r - 90.0;

    RAISE NOTICE 'Localização: (%, %)', longitude, latitude;

    pt := ST_SetSRID(ST_MakePoint(longitude, latitude), 4326);

    EXECUTE query INTO STRICT id USING pt;

    RETURN id;
END;
$$
LANGUAGE plpgsql;

```

Executando a função `random_insert` obtemos um resultado como:

```

SELECT random_insert();

NOTICE: Localização: (17.3685365170239600, 8.6842682585119800)

random_insert
-----
3
(1 row)

```

Observação: Os parâmetros como \$1 e \$2 só podem ser usados para os valores de atributos. Não é possível utilizá-los para substituir nomes de tabelas ou colunas. Neste caso, devemos concatenar o texto do comando ou usar a função `format()` com o caractere de substituição de nomes de tabelas e colunas (%I).

2.1.6 Retornando Conjuntos ou Tuplas

Podemos definir o tipo de retorno de uma função como sendo um conjunto de valores usando o construtor `SETOF` ou `TABLE`. Neste caso, ao invés de usar uma única instrução `RETURN`, utilizamos as instruções `RETURN NEXT` ou `RETURN QUERY`.

A função `generate_4pts` irá gerar um conjunto com quatro pontos sorteados de forma aleatória:

```
CREATE OR REPLACE FUNCTION generate_4pts()
RETURNS SETOF GEOMETRY
AS
$$
DECLARE
    pt GEOMETRY;
BEGIN
    pt := ST_SetSRID(ST_MakePoint(360.0 * random() - 180.0,
                                    180.0 * random() - 90.0), 4326);
    RETURN NEXT pt;

    pt := ST_SetSRID(ST_MakePoint(360.0 * random() - 180.0,
                                    180.0 * random() - 90.0), 4326);
    RETURN NEXT pt;

    pt := ST_SetSRID(ST_MakePoint(360.0 * random() - 180.0,
                                    180.0 * random() - 90.0), 4326);
    RETURN NEXT pt;

    pt := ST_SetSRID(ST_MakePoint(360.0 * random() - 180.0,
```

```

        180.0 * random() - 90.0), 4326);

RETURN NEXT pt;

RETURN;

END;
$$

LANGUAGE plpgsql;

```

Atente-se a um detalhe importante no código da função `generate_4pts`: ela termina com uma instrução `RETURN`.

Podemos invocar a função `generate_4pts` da seguinte forma:

```

SELECT ST_AsText(pt) AS geom FROM generate_4pts() AS pt;

      geom
-----
POINT(163.76629723236 85.4808924626559)
POINT(-55.0152626819909 55.0285322405398)
POINT(-1.04462971910834 -35.6856297980994)
POINT(-123.348454702646 31.6837284341455)
(4 rows)

```

Outra possibilidade de retornar um conjunto de valores é através do comando `RETURN QUERY`. A função `nearest_pluviometros` ilustra como podemos utilizar este comando:

```

CREATE OR REPLACE FUNCTION nearest_pluviometros(location GEOMETRY,
                                                distance NUMERIC)
RETURNS SETOF pluviometros
AS
$$
DECLARE
    q TEXT;
BEGIN
    q := 'SELECT * FROM pluviometros' ||

```

```

' WHERE ST_DWithin(location, $1, $2)';

RETURN QUERY EXECUTE q USING location, distance;

RETURN;

END;

$$

LANGUAGE plpgsql;

```

Invocando a função `nearest_pluviometros` obtemos o resultado mostrado abaixo:

```

SELECT gid, ST_AsText(location)
  FROM nearest_pluviometros(
    ST_SetSRID(ST_MakePoint(-45, -23), 4326),
    10.0);

      gid |      st_astext
-----+-----
      1 | POINT(-45.8872 -23.1791)
      2 | POINT(-43.6419 -20.393)

```

Outra possibilidade de escrita da função `nearest_pluviometros` seria sem a utilização do comando `EXECUTE`:

```

CREATE OR REPLACE FUNCTION nearest_pluviometros(location GEOMETRY,
                                                distance NUMERIC)
RETURNS SETOF pluviometros
AS
$$
BEGIN
  RETURN QUERY SELECT *
    FROM pluviometros
   WHERE ST_DWithin(pluviometros.location,
                    nearest_pluviometros.location,

```

```

        distance);

IF NOT FOUND THEN
    RAISE EXCEPTION 'Nenhum pluviometro nas proximidades: %', ST_AsText($1);
END IF;

RETURN;
END;
$$
LANGUAGE plpgsql;

```

2.1.7 Comandos de Repetição

PL/pgSQL possui diversos tipos de estruturas para laços de repetição: LOOP, WHILE, FOR e FOREACH. Além de comandos para desvio do fluxo de instruções: EXIT e CONTINUE.

FOR com variável de controle do tipo inteiro

Uma das variantes do laço tipo FOR possibilita iterar em uma variável do tipo inteiro. Como exemplo deste tipo de laço, vamos criar uma função para computar um certo número de pontos posicionados de forma aleatória. Este exemplo irá mostrar também como retornar um conjunto de registros. No caso, os registros terão o seguinte esquema: (INTEGER, GEOMETRY).

```

CREATE OR REPLACE FUNCTION random_pt_generator(npts NUMERIC)
RETURNS SETOF RECORD
AS $$

DECLARE
    tupla RECORD;
    longitude NUMERIC;
    latitude NUMERIC;
    pt GEOMETRY;
BEGIN

```

```

RAISE NOTICE 'Computando % pontos aleatórios...', npts;

FOR i IN 1..npts LOOP
    longitude := 360.0 * random() - 180.0;
    latitude := 180.0 * random() - 90.0;

    pt := ST_SetSRID(ST_MakePoint(longitude, latitude), 4326);

    tupla := (i, pt); -- ou: SELECT i, pt INTO tupla;

    RETURN NEXT tupla;

    IF (i % 1000) = 0 THEN
        RAISE NOTICE 'random_pt_generator: iteração %', i;
    END IF;
END LOOP;

RETURN;
END;
$$
LANGUAGE plpgsql;

```

A função `random_pt_generator` ilustra também um tipo de função que não pode ser utilizada somente na cláusula `SELECT` pois ela retorna um conjunto de tuplas. Neste caso, deve-se utilizar a função na cláusula `FROM`, mas com um cuidado especial, que é definir o tipo do conjunto retornado. Abaixo, mostramos como criar uma consulta que computa 5 pontos:

```

SELECT gid, ST_AsText(geom)
  FROM random_pt_generator(5) AS tabela(gid INTEGER, geom GEOMETRY);

NOTICE: Computando 5 pontos aleatórios...

gid |          st_astext
-----+

```

```

1 | POINT(25.8069811575115 4.40172334201634)
2 | POINT(-31.4398757368326 18.4167264495045)
3 | POINT(-42.4448257684708 -74.7917356621474)
4 | POINT(12.7397736534476 -52.6765340007842)
5 | POINT(87.8538537584245 -22.9236561711878)

(5 rows)

```

FOR sobre o resultado de uma consulta

Outro tipo de laço FOR pode ser empregado para iterar nas tuplas resultantes de uma consulta. Para ilustrar este tipo de laço, vamos construir uma função chamada `build_pt_table` que irá criar uma tabela geométrica com pontos gerados de forma aleatória usando como base nossa função `random_pt_generator`. Ao final esta função irá criar uma chave primária e um índice espacial sobre a coluna geométrica da tabela criada.

```

CREATE OR REPLACE FUNCTION build_pt_table(table_name TEXT,
                                         npts NUMERIC)
RETURNS VOID
AS
$$
DECLARE
    tupla RECORD;
    i      INTEGER DEFAULT 1;
BEGIN
    RAISE NOTICE 'Criando tabela %...', table_name;

    EXECUTE 'CREATE TABLE ' || table_name ||
        '(gid INTEGER, geom GEOMETRY(POINT,4326))';

    FOR tupla IN SELECT *
        FROM random_pt_generator(npts) AS tabela(gid INTEGER, geom GEOMETR
        LOOP
        EXECUTE 'INSERT INTO ' || table_name ||
            '(gid, geom) VALUES($1, $2)' USING tupla.gid, tupla.geom;

```

```
IF (i % 1000) = 0 THEN
    RAISE NOTICE 'Inseridos % tuplas!', i;
END IF;

i = i + 1;
END LOOP;

RAISE NOTICE 'Criando chave primária...';

EXECUTE format('ALTER TABLE %I ADD PRIMARY KEY(gid)', table_name);

RAISE NOTICE 'Criando índice espacial...';

EXECUTE 'CREATE INDEX spidx_' || table_name ||
'_geom ON ' || table_name || ' USING GIST(geom)';

RETURN;
END;
$$
LANGUAGE plpgsql;
```

Usando a função `build_pt_table`, vamos criar uma tabela chamada `pt10k` contendo 10.000 pontos:

```
SELECT build_pt_table('pt10k', 10000);

NOTICE: Criando tabela pt10k...
NOTICE: Computando 10000 pontos aleatórios...
...
NOTICE: Inseridos 1000 tuplas!
NOTICE: Inseridos 2000 tuplas!
NOTICE: Inseridos 3000 tuplas!
NOTICE: Inseridos 4000 tuplas!
NOTICE: Inseridos 5000 tuplas!
```

```

NOTICE: Inseridos 6000 tuplas!
NOTICE: Inseridos 7000 tuplas!
NOTICE: Inseridos 8000 tuplas!
NOTICE: Inseridos 9000 tuplas!
NOTICE: Inseridos 10000 tuplas!
NOTICE: Criando chave primária...
NOTICE: Criando índice espacial...
build_pt_table
-----

```

(1 row)

Outra forma útil deste tipo de FOR utiliza o comando EXECUTE:

```

[<<rótulo>>]
FOR variável-tupla IN EXECUTE query-string [USING expressão [, ... ]] LOOP
    comandos;
END LOOP [rótulo];

```

WHILE

```

[<<rótulo>>]
WHILE expressão-booleana LOOP
    comandos;
END LOOP [rótulo];

```

LOOP: repetições incondicionais

```

[<<rótulo>>]
LOOP
    comandos;
END LOOP [rótulo];

```

EXIT: interrompendo um laço ou bloco de comandos

```
EXIT [rótulo] [WHEN expressão-booleana];
```

CONTINUE: desviando o fluxo de uma laço

```
CONTINUE [rótulo] [WHEN expressão-booleana];
```

FOREACH: iterando sobre arrays

```
[<<rótulo>>]
FOREACH variável [SLICE número] IN ARRAY expressão-array LOOP
    comandos;
END LOOP [rótulo];
```

2.1.8 Blocos

Podemos criar e aninhar novos blocos dentro do bloco principal de uma função. Os blocos possuem a seguinte estrutura:

```
[<<rótulo>>]
DECLARE
    lista-variáveis;
BEGIN
    comandos;
[EXCEPTION
    WHEN condição THEN
        ...
]
END [rótulo];
```

A cláusula EXCEPTION permite que tratemos exceções lançadas durante a execução de uma função¹⁰.

¹⁰Conforme visto na 2.1.5 - Comandos que retornam uma única tupla como resultado.

2.2 Triggers

Uma restrição do tipo *check-constraint* permite definir regras básicas que devem ser asseguradas pelo SGBD quando realizamos operações de escrita sobre uma tabela. Para ilustrar o uso desse tipo de restrição, considere um empreendimento imobiliário de vendas de terrenos (ou lotes). Poderíamos criar uma tabela chamada `lotes` para armazenar as feições de cada terreno. Uma regra básica que poderia ser estabelecida, seria a de que o menor lote deva ter uma área mínima de 360m². A definição abaixo mostra como poderíamos construir essa tabela.

```
CREATE TABLE lotes
(
    gid SERIAL PRIMARY KEY,
    geom GEOMETRY(POLYGON, 0),
    CHECK(ST_Area(geom) >= 360.0)
);
```

Ao inserir a feição de um terreno com dimensões 12mx30m, respeitamos a regra imposta. Logo obteríamos uma mensagem de que a nova tupla foi adicionada à nossa tabela.:

```
INSERT INTO lotes (geom)
VALUES (ST_GeomFromText(
    'POLYGON((0 0, 12 0, 12 30, 0 30, 0 0))', 0));
```

Query returned successfully: one row affected, 16 ms execution time.

No entanto, se tentarmos adicionar uma nova tupla com dados de um terreno com dimensões de 10mx24m, fora da regra, o SGBD iria abortar a operação de inserção, mantendo assim a consistência da tabela com a regra estabelecida:

```
INSERT INTO lotes (geom)
VALUES (ST_GeomFromText(
    'POLYGON((12 0, 22 0, 22 24, 12 24, 12 0))', 0));
```

Suponha agora que queiramos incluir uma nova regra na nossa tabela a fim de evitar que um novo lote seja cadastrado de forma a ter sobreposição com algum outro lote já cadastrado. Neste caso, não é possível definir esta regra como uma restrição do tipo *check-constraint*. Precisamos de um mecanismo que possibilite expressar essa verificação do novo conteúdo sendo inserido na nossa tabela com dados que já existem na própria tabela.

Um mecanismo útil para estabelecer este tipo de restrição é conhecido por *trigger*, que é uma ação associada a uma tabela, invocada automaticamente sempre que modificamos o conteúdo dessa tabela¹¹.

No PostgreSQL, um *trigger* nada mais é do que uma função com uma notação especial e algumas facilidades para gerenciar o tipo de evento responsável por disparar esta função.

A função PL/pgSQL usada para definir o *trigger* não possui parâmetros e o tipo de retorno é do tipo **trigger**, como mostrado abaixo:

```
CREATE OR REPLACE FUNCTION nome-da-função-trigger()
RETURNS trigger
AS
$$
DECLARE
    lista-variáveis;
BEGIN
    comandos;
END;
$$ LANGUAGE plpgsql;
```

A definição do *trigger* é realizada através do comando CREATE TRIGGER, que possui a seguinte sintaxe:

CREATE [CONSTRAINT] TRIGGER nome-trigger

¹¹Aqui a palavra modificação pode ser traduzida em um dos comandos INSERT, UPDATE ou DELETE.

```
{BEFORE | AFTER | INSTEAD OF} {tipo-evento [OR ...]}

ON nome-tabela
[FROM referenced_table_name ]
[NOT DEFERRABLE |
 [DEFERRABLE] [INITIALLY IMMEDIATE | INITIALLY DEFERRED]]
[FOR [EACH] {ROW | STATEMENT}]
[WHEN (condition)]
EXECUTE PROCEDURE nome-função-trigger(argumentos)
```

No caso de tabelas, um *trigger* pode ser definido para execução antes (BEFORE) ou após (AFTER) qualquer operação de inserção (INSERT), atualização (UPDATE) ou remoção (DELETE). Também podemos definir que o *trigger* será disparado por múltiplos eventos, neste caso usamos um *ou-lógico* (OR) para encadear os tipos de eventos. Portanto, no comando acima, os tipos de eventos suportados são:

- INSERT.
- UPDATE [OF column_name [, ...]].
- DELETE.
- TRUNCATE.

Também podemos controlar se a função associada ao *trigger* será executada uma vez para cada linha modificada ou se será chamada uma única vez para todo o comando SQL que disparou o *trigger*.

Para um *trigger* ser executado para cada linha modificada usamos:

```
FOR EACH ROW EXECUTE PROCEDURE nome-da-função-trigger()
```

Para um *trigger* ser executado uma única vez para todo o comando SQL que o disparou, usamos:

```
FOR EACH STATEMENT EXECUTE PROCEDURE nome-da-função-trigger()
```

Uma função PL/pgSQL invocada como um *trigger* possui diversas variáveis especiais que são automaticamente criadas:

- **TG_OP**: uma constante string (TEXT) que diz o tipo de operação que disparou o *trigger*: 'INSERT', 'UPDATE', 'DELETE' ou 'TRUNCATE'.
- **TG_WHEN**: uma constante string (TEXT) que diz quando o *trigger* foi disparado: 'BEFORE', 'AFTER' ou 'INSTEAD OF'.
- **NEW**: variável do tipo registro (RECORD) contendo a nova tupla em operações de inserção (INSERT) e atualização (UPDATE) em *triggers*, para *triggers* definidos em nível de linha¹².
- **OLD**: variável do tipo registro (RECORD) contendo o valor da antiga tupla em operações de atualização (UPDATE) e remoção (DELETE), para *triggers* definidos em nível de linha¹³.

A função que implementa a ação do *trigger* deve retornar NULL ou um valor do tipo registro (ou tupla) com a mesma estrutura da tabela para a qual o *trigger* foi disparado.

Retomando o nosso exemplo do lote, podemos criar um *trigger* associado à tabela *lotes* que ajude a manter a integridade dos dados de forma a não haver sobreposição entre os lotes.

```
CREATE OR REPLACE FUNCTION verifica_overlap_lote()
RETURNS trigger
AS
$$
DECLARE
    lote lotes%ROWTYPE; -- ou: RECORD
BEGIN
    IF TG_OP = 'INSERT' THEN
        FOR lote IN SELECT *
```

¹²Essa variável não é definida em operações do tipo DELETE ou em *triggers* definidos em nível de comandos.

¹³Essa variável não é definida em operações do tipo INSERT ou em *triggers* definidos em nível de comandos.

```

        FROM lotes
        WHERE ST_Intersects(NEW.geom, geom) LOOP
    IF NOT ST_Touches(NEW.geom, lote.geom) THEN
        RAISE EXCEPTION 'Lote viola restrição de integridade espacial!';
    END IF;
END LOOP;

ELSIF TG_OP = 'UPDATE' THEN
    FOR lote IN SELECT *
        FROM lotes
        WHERE ST_Intersects(NEW.geom, geom)
        AND (lotes.gid != OLD.gid) LOOP
    IF NOT ST_Touches(NEW.geom, lote.geom) THEN
        RAISE EXCEPTION 'Lote viola restrição de integridade espacial!';
    END IF;
END LOOP;
END IF;

RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

Agora, precisamos definir o *trigger* sobre a tabela:

```

CREATE TRIGGER trigger_verifica_overlap_lote
BEFORE INSERT OR UPDATE
ON lotes
FOR EACH ROW EXECUTE PROCEDURE verifica_overlap_lote();

```

Se inserirmos um lote que não viole nossa restrição de integridade espacial, o trigger irá executar normalmente e retornará o registro com o valor da nova linha que será inserida na tabela:

```

INSERT INTO lotes
(geom)

```

```

VALUES (
    ST_GeomFromText(
        'POLYGON((12 0, 24 0, 24 30, 12 30, 12 0))',
        0));

```

INSERT 0 1

No entanto, se tentarmos inserir um novo lote que viole a restrição de integridade espacial definida pelo *trigger*, o SGBD irá cancelar a operação de inserção:

```

INSERT INTO lotes
(geom)
VALUES (
    ST_GeomFromText(
        'POLYGON((-5 0, 7 0, 7 30, -5 30, -5 0))',
        0));

```

ERROR: Lote viola restrição de integridade espacial!

No caso de alterarmos a geometria do primeiro lote, o *trigger* também irá ajudar a manter a consistência do banco de dados, conforme podemos observar com o exemplo abaixo:

```

UPDATE lotes
SET geom = ST_GeomFromText(
    'POLYGON((11 0, 23 0, 23 30, 11 30, 11 0))',
    0)
WHERE gid = 3;

```

Observação: Para remover o objeto *trigger* use um comando como:

```
DROP TRIGGER trigger_verifica_overlap_lote ON lotes;
```

2.3 Extensões

Esta seção encontra-se em desenvolvimento e, portanto, ainda está incompleta!

Uma das características mais interessantes do SGBD PostgreSQL é sua extensibilidade. Podemos estender sua linguagem SQL de diversas formas, entre elas¹⁴:

- Através da criação de novas *funções*, seja através de programas escritos em PL/pgSQL, como fizemos na Seção 2.1, ou através de funções escritas em Linguagem C (nossa objetivo nesta seção).
- Criação de novas funções de *agregação* (ou *aggregates*);
- Criação de novos *tipos de dados* (*data types*);
- Definição e sobrecarga de *operadores* (*operators*);
- Definição de *classes de operadores para índices* (*operator classes for indexes*);
- Criação de extensões na forma de *pacotes*.

Se analisarmos o projeto do PostgreSQL, sua extensibilidade se dá, entre outros motivos, pela forma como o SGBD lida com o registro de todos os possíveis tipos de objetos existentes em um banco de dados. Suas tabelas de metadados incluem não só informações sobre as tabelas, colunas e índices existentes em um banco de dados, mas também informações sobre os tipos de dados, funções, operadores e métodos de acesso (ou mecanismos de indexação). Essa capacidade de trabalhar com tipos, funções e operações definidas no próprio catálogo, conhecida no manual do PostgreSQL por *operações dirigidas pelo catálogo* (*catalog-driven*), é fundamental para a extensibilidade do SGBD.

Outra característica interessante do PostgreSQL é sua capacidade de carregar as extensões através da carga dinâmica de módulos (bibliotecas), sem a necessidade de interromper ou reiniciar o servidor.

¹⁴A melhor documentação sobre extensibilidade do SGBD PostgreSQL é seu manual [7].

2.4 Considerações Finais

Uma boa leitura sobre o assunto de Sistemas de Bancos de Dados Objeto-Relacionais e extensibilidade pode ser encontrada nos livros de Stonebraker [8] e [9].

Existem vários motivos para se usar *procedimentos armazenados*. Um deles tem haver com a diminuição da quantidade de comunicação exigida entre os processos das aplicações clientes e o processo do servidor de bancos de dados. Cada consulta SQL submetida por uma aplicação envolve uma comunicação entre processos, que possui um custo. Além disso, as aplicações cliente e servidor podem estar em diferentes máquinas, exigindo uma comunicação pela rede. Os procedimentos armazenados permitem que agrupemos vários comandos SQL, o que pode ser útil para evitar o tráfego de informações intermediárias que não sejam úteis à aplicação cliente.

Outro uso importante de procedimentos armazenados é para tornar as aplicações clientes o mais imune possível à mudanças da parte lógica do negócio (ou das regras de negócio). Também podemos criar funções que verifiquem a integridade dos dados manipulados por uma aplicação.

Uma boa discussão, diz respeito a quanto de computação (ou lógica) deva ser realizada do lado cliente, e quanto deva ser movida para o lado servidor. As aplicações modernas são construídas em várias camadas, podendo delegar as regras de negócio nos chamados servidores de aplicação. No entanto, muitas vezes é mais interessante mover a computação para onde o dado se encontra, a fim de diminuir a quantidade de dado trafegado para o lado cliente. Outro fato a ser considerado é quando temos várias aplicações, cada uma possivelmente escrita em uma linguagem diferente, que realizam um mesmo tipo de computação. Muitas vezes criar um procedimento armazenado no SGBD simplifica a manutenção dessa computação uma vez que em todas as aplicações apenas a chamada a esta função será utilizada.

Procedimentos armazenados PL/pgSQL podem aceitar um número variável de argumentos (**VARIADIC**). Também podem aceitar e retornar os tipos polimórficos **ANY**, **ANYELEMENT**, **ANYARRAY**, **ANYNONARRAY**, **ANYENUM** ou **ANYRANGE**, de maneira que os tipos reais podem ser mudados a cada chamada da função. Outra capacidade interessante dessas funções é a possibilidade de declará-las com o tipo de retorno de um conjunto de valores de um certo tipo escalar (**RETURNS SETOF** ou **RETURNS TABLE**). Neste caso, a saída é gerada

executando o comando `RETURN NEXT` ou `RETURN QUERY`. Funções podem ser sobre carregadas. Podemos declarar parâmetros de retorno (`OUT`). Podemos associar um custo às funções de modo a ajudar o *query planner* a realizar seu trabalho, ele terá uma forma de estimar quanto irá custar a chamada de sua função.

Triggers são importantes ferramentas para aplicações que necessitam de uma auditoria das modificações realizadas em tabelas especiais ou para gerenciamento de dados relacionados em várias tabelas. Um *trigger* pode ser criado em qualquer linguagem procedural do PostgreSQL, bem como na Linguagem C. Além da associação com tabelas, um *trigger* pode ser associado a visões (*views*) e tabelas externas (*foreign tables*). Como comandos `SELECT` não modificam linhas, não podemos criar *triggers* associadas a este tipo de comando. Neste caso, o mais adequado é o uso de regras (*rules*) e visões (*views*). Para maiores detalhes sobre *triggers*, consulte [10].

As cláusulas `BEGIN` e `END` em funções PL/pgSQL possuem o único propósito de agrupar os comandos, não tendo nenhuma relação com os comandos de controle de transações. Tanto funções quanto *triggers* são sempre executadas dentro da transação iniciada pela consulta que as invocaram.

2.5 Exercícios

Os exercícios desta seção ilustram como criar *triggers* e como criar regras para manter o relacionamento espacial entre objetos geográficos.

1. Crie uma tabela chamada `quadras` com o seguinte esquema:

```
quadra(gid : SERIAL PRIMARY KEY, geom : GEOMETRY(POLYGON, 0))
```

2. Crie um *trigger* que não permita a sobreposição de quadras.
3. Crie um *trigger* para garantir que um lote ao ser criado ou alterado esteja contido em uma quadra.

Referências Bibliográficas

- [1] OGC. OpenGIS Implementation Standard for Geographic Information - Simple Feature Access - Part 1: Common Architecture. Technical report, Open Geospatial Consortium, 2011.
- [2] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In *Proceedings of the Third International Symposium on Advances in Spatial Databases*, SSD '93, pages 277–295, Berlin, Heidelberg, 1993. Springer-Verlag.
- [3] Rick Greenwal, Robert Stackowiak, and Jonathan Stern. *Oracle Essentials: Oracle Database 12c*. O'Reilly, CA, USA, 5th edition, 2013.
- [4] Grant Allen. *Beginning DB2: From Novice to Professional*. Apress, CA, USA, 1st edition, 2008.
- [5] Joshua D. Drake and John C. Worsley. *Practical PostgreSQL*. O'Reilly, CA, USA, 1st edition, 2002.
- [6] The PostgreSQL Global Development Group. Postgresql 9.5.3 documentation - chapter 40. plpgsql - sql procedural language.
- [7] The PostgreSQL Global Development Group. Postgresql 9.5.3 documentation - chapter 35. extending sql.
- [8] Michael Stonebraker. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, CA, USA, 2nd edition, 1996.
- [9] Michael Stonebraker, Paul Brown, and Dorothy Moore. *Object-Relational DBMSs*. Morgan Kaufmann, CA, USA, 2nd edition, 1998.

- [10] The PostgreSQL Global Development Group. Postgresql 9.5.3 documentation - chapter 36. triggers.