8 SGBD com extensões espaciais

Gilberto Ribeiro de Queiroz
Karine Reis Ferreira

8.1 Introdução

Este capítulo apresenta duas extensões espaciais de SGBD convencionais, uma da comunidade de software aberto (PostGIS) e outra comercial (Oracle Spatial).

O PostGIS (Ramsey, 2002) estende o PostgreSQL, seguindo as especificações da SFSSQL. O PostgreSQL é um sistema de gerência de banco de dados objeto-relacional, gratuito e de código fonte aberto (Stonebraker et al, 1990). Foi desenvolvido a partir do projeto Postgres, iniciado em 1986, na Universidade da Califórnia em Berkeley.

O Spatial (Ravada e Sharma, 1999) (Murray, 2003) estende o modelo objeto-relacional do sistema de gerência de banco de dados Oracle. Esta extensão baseia-se nas especificações do OpenGIS.

8.2 Cenários

Esta seção apresenta alguns cenários que serão empregados ao longo do capítulo para ilustrar os principais recursos das extensões espaciais consideradas. Os exemplos que iremos apresentar, representam as informações de distritos, bairros e rede de drenagem da cidade de São Paulo; os municípios que formam a grande São Paulo (São Caetano do Sul, Suzano, Guarulhos entre outras). A Figura 8.1 mostra a parte geométrica dos distritos de São Paulo. Para cada distrito associaremos um polígono e os atributos código do distrito (COD), sigla de abreviação (SIGLA) e o nome do distrito (DENOMINACAO). Aos pontos que representam bairros da cidade de São Paulo (Figura 8.2) iremos associar

os atributos nome do bairro (BAIRRO) e o nome do distrito (DISTR). Às linhas da Figura 8.3 (mapa de drenagem) associaremos um único atributo, a classe de drenagem (CLASSE). E, aos polígonos dos municípios da grande São Paulo, os atributos código do município (CODMUNIC), nome do município (NOMEMUNICP) e população (POPULACAO).

Figura 8.1 – Mapa de Distritos da Cidade de São Paulo.

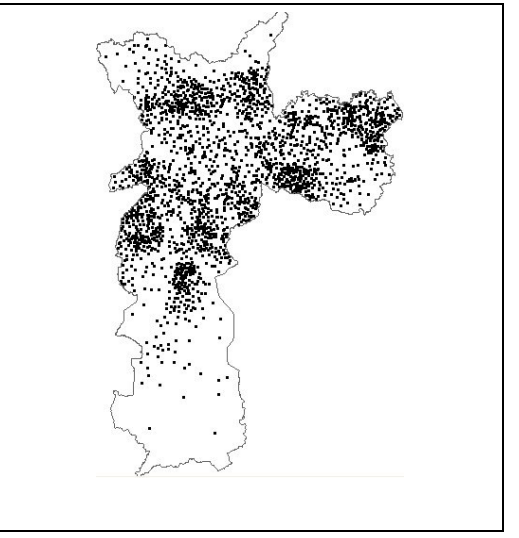

Figura 8.2 – Mapa de Bairros da Cidade de São Paulo.

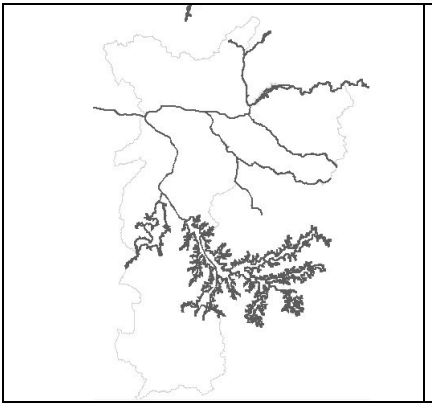

Figura 8.3 – Mapa de drenagem.

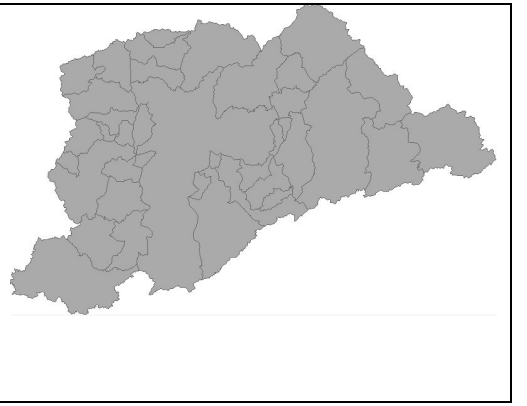

Figura 8.4 – Mapa dos Municípios da Grande São Paulo.

A partir dessas informações, iremos construir alguns cenários envolvendo consultas espaciais que serão utilizados durante a explicação dos recursos de cada uma das extensões.

Cenário 1: "Recuperar o nome de todos os municípios da grande São Paulo que são vizinhos ao município de São Paulo". A Figura 8.5 ilustra esse cenário, com o município de São Paulo destacado em preto e os municípios vizinhos destacados em cinza claro.

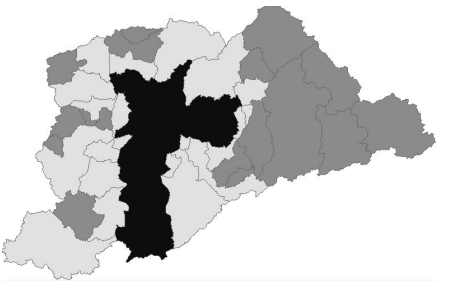

Figura 8.5 – Municípios vizinhos à cidade de São Paulo.

Cenário 2: "Recuperar o nome de todos os municípios da grande São Paulo que são vizinhos ao distrito Anhanguera da cidade de São Paulo". A Figura 8.6 ilustra este cenário, com o distrito Anhanguera de São Paulo destacado em preto e os municípios vizinhos a este distrito, destacados em cinza claro.

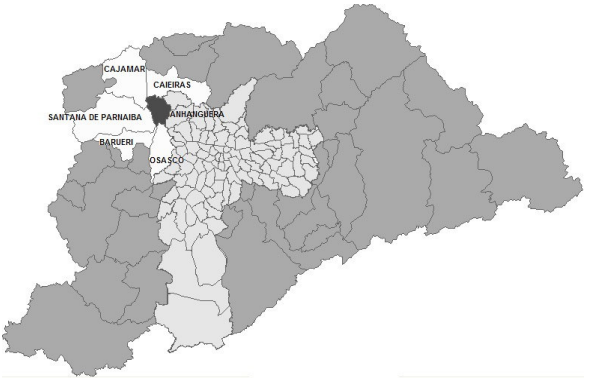

Figura 8.6 – Municípios vizinhos ao distrito Anhanguera.

Cenário 3: "Recuperar o número de bairros contidos no distrito Grajaú". A Figura 8.7 ilustra o cenário, com o distrito Grajaú destacado em cinza escuro e os municípios contidos neste distrito de cinza claro.

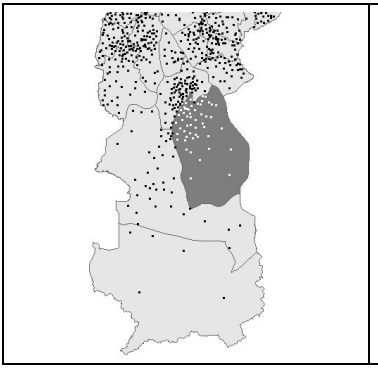

Figura 8.7 – Bairros contidos no distrito de Grajaú.

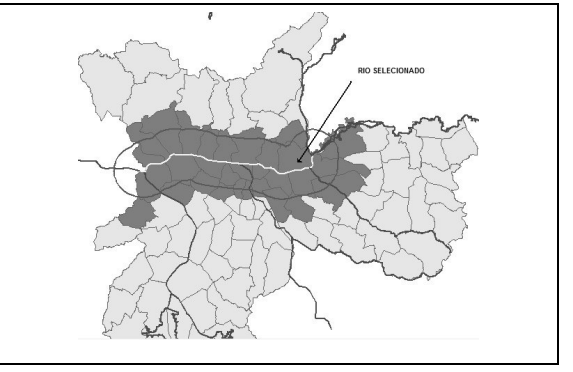

Figura 8.8 – Buffer de 3Km ao redor de um rio e distritos nesta região de influência.

Cenário 4: "Recuperar todos os distritos que estão num raio de 3km de um determinado rio". A Figura 8.8 ilustra o resultado desta consulta, com o rio selecionado destacado em branco e os distritos em cinza escuro contidos no raio de 3 Km do rio. A linha em cinza escuro ao redor do rio selecionado representa um buffer gerado com o qual os distritos foram testados (relacionamento intercepta).

Cenário 5: "Recuperar todos os bairros que estejam a menos de 3 Km do bairro Boacava".

8.3 PostGIS para PostgreSQL

8.3.1 Características principais do PostgreSQL

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional, gratuito e de código fonte aberto, desenvolvido a partir do projeto Postgres, iniciado em 1986, na Universidade da Califórnia em Berkeley, sob a liderança do professor Michael Stonebraker. Em 1995, quando o suporte a SQL foi incorporado, o código fonte foi disponibilizado na Web (http://www.postgresql.org). Desde então, um grupo de desenvolvedores vem mantendo e aperfeiçoando o código fonte sob o nome de PostgreSQL.

Em sua versão de distribuição oficial, ele apresenta tipos de dados geométricos (Figura 8.9), operadores espaciais simples e indexação espacial através de uma R-Tree nativa ou através de R-Tree implementada no topo do mecanismo de indexação GiST (Hellerstein et al, 1995).

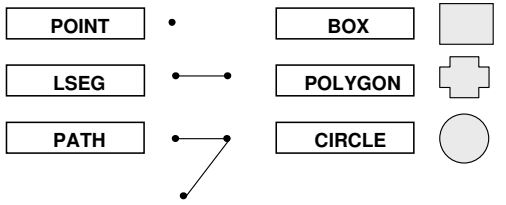

Figura 8.9 - Tipos Geométricos do PostgreSQL.

A implementação da R-Tree nativa possui uma severa limitação em seu uso, uma coluna do tipo polígono não pode exceder 8Kbytes. Na prática, é muito comum um SIG manipular dados representados, por exemplo, por polígonos maiores do 8Kbytes cada, o que torna o uso desse índice inviável. Uma alternativa é o uso da segunda R-Tree.

O método de indexação chamado GiST foi introduzido por Hellerstein et al (1995) e implementado no PostgreSQL. Atualmente, ele é mantido por Signaev e Bartunov (http://www.sai.msu.su/~megera/postgres/gist) e não possui restrições de tamanho do dado a ser indexado. GiST é uma abreviação de *Generalized Search Trees*, consistindo em um índice (árvore balanceada) que pode fornecer tanto as funcionalidades de uma B-Tree quanto de uma R-Tree e suas variantes. A principal vantagem desse índice é a possibilidade de definição do seu "comportamento".

Quanto aos poucos operadores espaciais existentes, estes realizam a computação apenas sobre o retângulo envolvente das geometrias e não diretamente nelas. Já os tipos de dados simples, como polígonos, não permitem a representação de buracos, não existindo também geometrias que permitam representar objetos mais complexos como os formados por conjuntos de polígonos.

Portanto, a integração de um SIG através desses tipos geométricos requer muito esforço – implementação de novas operações espaciais como união, interseção, testes topológicos sobre a geometria exata, definição de um modelo de suporte a polígonos com buracos, entre outras.

Mas, como dito anteriormente, um dos pontos fortes desse SGBD é seu potencial de extensibilidade, o que possibilitou o desenvolvimento de uma extensão geográfica mais completa, chamada PostGIS. Antes de entrar em maiores detalhes dessa extensão, apresentaremos um pouco do mecanismo de extensibilidade para que o leitor possa entender melhor o núcleo da extensão PostGIS.

8.3.2 Extensibilidade do PostgreSQL

O mecanismo de extensibilidade do PostgreSQL permite incorporar capacidades adicionais ao sistema de forma a torná-lo mais flexível para o

gerenciamento de dados para cada classe de aplicação. No caso dos SIG, isso significa a possibilidade do desenvolvimento de uma extensão geográfica capaz de armazenar, recuperar e analisar dados espaciais. A seguir serão apresentados alguns passos envolvidos na criação de uma extensão do PostgreSQL. Os exemplos foram adaptados do tipo de dados POLYGON existente na distribuição oficial.

8.3.2.1 Tipos de dados definidos pelo usuário

A extensão de tipos de dados no PostgreSQL pode ser feita em linguagem C (Kernighan e Ritchie, 1990) através da definição de uma estrutura de dados, que é responsável pela representação do tipo em memória. O trecho de código abaixo ilustra a definição de um tipo chamado `TeLinearRing`, que representa uma linha fechada composta por um conjunto de coordenadas (do tipo `TeCoord2D`). Os tipos definidos pelo usuário podem ser de tamanho fixo ou variável, como no caso do exemplo abaixo (variável):

```
typedef struct
{ int32 size;             //struct length
  int32 ncoords_;         //number of coords
  TeCoord2D coords_[1];   //variable length array
} TeLinearRing;
```

Além da estrutura de dados, é necessário definir outras duas rotinas[1] que fazem a conversão do tipo de acordo com a sua representação em memória para ASCII e vice-versa. Estas rotinas são conhecidas como funções de entrada e saída, e elas associam uma representação textual externa para o dado. Essas funções são utilizadas internamente para permitir que o tipo seja usado diretamente em comandos SQL. Por exemplo, para o tipo TeLinearRing, poderíamos optar pelo seguinte formato de representação: *[(X1, Y1), (X2, Y2), ..., (Xn, Yn)]*. O trecho de código abaixo ilustra a implementação de uma rotina de entrada para o tipo `TeLinearRing`[2]:

```
00   PG_FUNCTION_INFO_V1(TeLinearRing_in);
01   Datum TeLinearRing_in(PG_FUNCTION_ARGS)
```

[1] A partir da versão 7.4, pode-se definir das funções de I/O no formato binário.
[2] As funções `number_of_coords` e `decode` serão omitidas por questões de espaço.

```
02  { char* str = PG_GETARG_CSTRING(0);
03    TeLinearRing* ring;      int ncoords;
04    int size;                char* s;
05
06    if((ncoords = number_of_coords(str, ')')) <= 0)
07      elog(ERROR, "Bad TeLinearRing external
08                   representation '%s'", str);
09    size = offsetof(TeLinearRing, coords_[0]) +
10           sizeof(ring->coords_[0]) * ncoords;
11    ring = (TeLinearRing*)palloc(size);
12    MemSet((char *) ring, 0, size);
13    ring->size = size;       ring->ncoords_ = ncoords;
14    if((!decode(ncoords, str, &s,
15                &(ring->coords_[0]), '[', ']')) ||
16        (*s != '\0'))
17        elog(ERROR, "Bad TeLinearRing external
18             representation '%s'", str);
19    if(!is_TeLinearRing(ring))
20    { pfree(ring);
21      ring = NULL;
22      elog(ERROR, "In a TeLinearRing the first point
23           must be the same as the last point '%s'",
24           str);
25    }
26    PG_RETURN_TeLine2D_P(ring);
27  }
```

Os passos envolvidos na definição desta rotina são:

- Na linha 02, podemos observar que a rotina recebe um polígono (`TeLinearRing`) na forma textual (`str`).
- Na linha 06, a função `number_of_coords` determina quantas coordenadas formam a fronteira do polígono.
- Na linha 11, é alocada memória suficiente para um polígono com o número de coordenadas determinado na linha 06.
- Finalmente, na linha 14, a *string* é decodificada e as coordenadas são armazenadas no vetor de coordenadas da fronteira do polígono.

A função de saída é mais simples, recebe o dado na representação interna, devendo convertê-lo para a representação externa. O trecho de código abaixo ilustra a função de saída para o tipo `TeLinearRing`:

```
PG_FUNCTION_INFO_V1(TeLinearRing_out);
Datum TeLinearRing_out(PG_FUNCTION_ARGS)
{TeLinearRing *ring = (TeLinearRing*)
                        PG_GETARG_TeLinearRing_P(0);
 char* str;
 str = palloc(ring->ncoords_ * (P_MAXLEN + 3) + 2);

 if(!encode(ring->ncoords_, ring->coords_, str, '[',
            ']', "Unable to format TeLinearRing"))
   elog(ERROR, "Unable to format TeLinearRing");
 PG_RETURN_CSTRING(str); }
```

Depois de criado o tipo, uma biblioteca compartilhada deve ser gerada para ser integrada dinamicamente ao servidor de banco de dados. É necessário registrar o tipo através do comando SQL: `CREATE TYPE`, informando as rotinas de entrada e saída:

```
CREATE FUNCTION telinearring_in(opaque)
    RETURNS telinearring
    AS '/opt/tepgdatatypes.so'
    LANGUAGE 'c';

CREATE FUNCTION telinearring_out(opaque)
    RETURNS opaque
    AS '/opt/tepgdatatypes.so'
    LANGUAGE 'c';

CREATE TYPE telinearring
( alignment = double,
  internallength = VARIABLE,
  input   = telinearring_in,
  output  = telinearring_out,
  storage = main );
```

Agora podemos utilizar o tipo `TeLinearRing` dentro de comandos SQL, como:

- Criar uma tabela com uma coluna do tipo TeLinearRing:

```
CREATE TABLE tab_poligonos_simples
(id INTEGER, geom TeLinearRing);
```

- Inserir um quadrado (4 x 4):

```
INSERT INTO tab_poligonos_simples VALUES(1, '[(0,
0), (4, 0), (4, 4), (0, 4), (0, 0)]');
```

- Recuperar todos os polígonos:

```
SELECT * FROM tab_poligonos_simples;
Resultado:
id | geom
-----------------------------------------
1  | '[(0, 0), (4, 0), (4, 4), (0, 4), (0, 0)]'
```

8.3.2.2 Funções e operadores definidos pelo usuário

Além da flexibilidade do sistema de tipos, o PostgreSQL permite a criação de métodos que operem sobre as instâncias dos tipos definidos. Essas funções podem ser integradas ao servidor informando o nome da função, a lista de argumentos e o tipo de retorno. Essas informações são registradas no catálogo do sistema.

O trecho de código abaixo ilustra a definição de uma função para calcular a área de um polígono simples representado por um `TeLinearRing`.

```
PG_FUNCTION_INFO_V1(TeLinearRingArea);
Datum TeLinearRingArea(PG_FUNCTION_ARGS)
{TeLinearRing *r = (TeLinearRing *)
                   PG_DETOAST_DATUM(PG_GETARG_DATUM(0));
 double area = 0.0; int npoints = 0;
 int    loop;       npoints = r->ncoords_;

 for(loop = 0; loop < (npoints - 1); ++loop)
 {  area += ((r->coords_[loop].x_ *
              r->coords_[loop + 1].y_) -
             (r->coords_[loop + 1].x_ *
              r->coords_[loop].y_)); }
 area *= 0.5;
 PG_RETURN_FLOAT8(area); }
```

Assim como os tipos, as funções também devem ser colocadas em uma biblioteca compartilhada para poderem ser integradas ao servidor de banco de dados. É necessário registrar a função através do comando SQL: `CREATE FUNCTION`, como mostrado abaixo:

```
CREATE FUNCTION area(telinearring)
RETURNS float4
AS '/opt/tepgfunctions.so', 'telinearringarea'
LANGUAGE 'c' WITH (isstrict);
```

Outra funcionalidade importante oferecida pelo PostgreSQL é que os nomes das funções podem ser sobrecarregadas desde que os parâmetros sejam de tipos diferentes. Depois de criada uma função, é possível definir um operador para ela através do comando SQL: `CREATE OPERATOR`. A importância da definição do operador está ligada ao otimizador de consultas que pode usar as informações contidas na definição do operador para decidir a melhor estratégia para a consulta (ou simplificação desta). Outra funcionalidade oferecida pelo PostgreSQL é a definição de funções agregadas, que podem ser construídas de forma análoga às funções sobre tipos, porém são registradas com o comando SQL: `CREATE AGGREGATE`.

8.3.2.3 Extensão do mecanismo de indexação

O PostgreSQL possui quatro mecanismos de indexação (B-Tree, R-Tree, GiST e HASH). Esses índices podem ser usados para os tipos de dados e funções definidos pelo usuário, bastando para isso registrar no catálogo do sistema as informações das operações necessárias a cada índice. Por exemplo, para um tipo que deseje ser indexado pela B-Tree, é necessário indicar ao sistema as operações de comparação "<", "<=", "=", ">=" e ">" para que o índice saiba como realizar buscas.

O exemplo abaixo ilustra a implementação do "operador <" para o tipo `TeCoord2D` (coordenada com os campos x_ e y_), sendo que os demais operadores diferem apenas à forma da chamada:

```
static int
tecoord2d_cmp_internal(TeCoord2D* a, TeCoord2D* b)
{  if(a->x_ < b->x_)
     return -1;
   if(a->x_ > b->x_)
     return 1;
   if(a->y_ < b->y_)
     return -1;
   if(a->y_ > b->y_)
     return 1;
   return 0;
```

```
}

PG_FUNCTION_INFO_V1(tecoord2d_lt);
Datum
tecoord2d_lt(PG_FUNCTION_ARGS)
{   TeCoord2D *a = (TeCoord2D*) PG_GETARG_POINTER(0);
    TeCoord2D *b = (TeCoord2D*) PG_GETARG_POINTER(1);
    PG_RETURN_BOOL(tecoord2d_cmp_internal(a, b) < 0); }

Datum
tecoord2d_cmp(PG_FUNCTION_ARGS)
{ TeCoord2D    *a = (TeCoord2D*) PG_GETARG_POINTER(0);
  TeCoord2D    *b = (TeCoord2D*) PG_GETARG_POINTER(1);
  PG_RETURN_INT32(tecoord2d_cmp_internal(a, b)); }
```

Agora, podemos registrar a função e definir o operador através da seguinte seqüência de comandos:

```
CREATE FUNCTION tecoord2d_lt(TeCoord2D, TeCoord2D) RETURNS
bool
    AS 'filename', 'tecoord2d_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION tecoord2d_cmp(TeCoord2D, TeCoord2D)
    RETURNS integer
    AS 'filename', 'tecoord2d_cmp'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = TeCoord2D, rightarg = TeCoord2D,
   procedure = tecoord2d_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

E, por último, criamos a classe de operação requerida pelo índice *B-tree:*

```
CREATE OPERATOR CLASS tecoord2d_ops
    DEFAULT FOR TYPE TeCoord2D USING btree AS
        OPERATOR   1       < ,
        OPERATOR   2       <= ,
        OPERATOR   3       = ,
        OPERATOR   4       >= ,
        OPERATOR   5       > ,
        FUNCTION 1 tecoord2d_cmp(TeCoord2D, TeCoord2D);
```

8.3.3 A extensão espacial PostGIS do PostgreSQL

A comunidade de software livre vêm desenvolvendo a extensão espacial PostGIS, construída sobre o PostgreSQL. Atualmente, a empresa Refractions Research Inc (http://postgis.refractions.net) mantém a equipe de desenvolvimento dessa extensão, que segue as especificações da SFSSQL.

8.3.3.1 Tipos de dados espaciais

A Figura 8.10 ilustra os tipos espaciais suportados pelo PostGIS e embutidos na SQL do PostgreSQL.

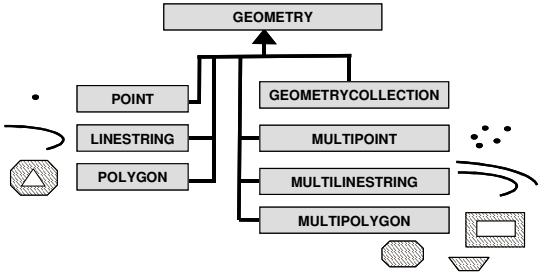

Figura 8.10 - Tipos de dados espaciais do PostGIS.

Esses tipos possuem a seguinte representação textual:

- `Point: (0 0 0)`
- `LineString: (0 0, 1 1, 2 2)`
- `Polygon: ((0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0), ( 1 0 0, ...), ...)`
- `MultiPoint: (0 0 0, 4 4 0)`
- `MultiLineString: ((0 0 0, 1 1 0, 2 2 0), (4 4 0, 5 5 0, 6 6 0))`
- `MultiPolygon: (((0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0), (...), ...), ...)`

- `GeometryCollection: (POINT(2 2 0), LINESTRING((4 4 0, 9 9 0))`

Como exemplo, mostraremos os comandos em SQL para gerar tabelas para armazenar os atributos e as geometrias[3]:

- Dos distritos de São Paulo, mostrados na Figura 8.1.

```
CREATE TABLE distritossp
( cod               SERIAL,
  sigla           VARCHAR(10),
  denominacao     VARCHAR(50),
  PRIMARY KEY  (cod)
);
SELECT AddGeometryColumn('terralibdb', 'distritossp',
'spatial_data', -1, 'POLYGON', 2);
```

- Dos bairros de São Paulo, mostrados na Figura 8.2.

```
CREATE TABLE bairrossp
( cod               SERIAL,
  bairro            VARCHAR(40),
  distr             VARCHAR(40),
  PRIMARY KEY       (cod)
);
SELECT AddGeometryColumn('terralibdb',
      'bairrossp', 'spatial_data',  -1,       'POINT',
   2);
```

- Do mapa de drenagem, mostrado na Figura 8.3:

```
CREATE TABLE drenagemsp
( cod            SERIAL,
  classe        VARCHAR(255) NULL,
  PRIMARY KEY   (cod)
);
SELECT AddGeometryColumn('terralibdb', 'drenagemsp',
'spatial_data', -1, 'LINESTRING', 2);
```

- Do mapa de municípios da grande São Paulo (Figura 8.4)

```
CREATE TABLE grande_sp
( cod            SERIAL,
  nomemunicp    VARCHAR(255) NULL,
  populacao     INTEGER,
  PRIMARY KEY  (cod)
```

[3] Aqui consideramos a existência de um banco de dados chamado terralibdb

```
);
SELECT AddGeometryColumn('terralibdb', 'grande_sp',
'spatial_data', -1, 'POLYGON', 2);
```

Observe que a criação de uma tabela com tipo espacial é construída em duas etapas. Na primeira, definimos os atributos básicos (alfanuméricos) e na segunda, usamos a função `AddGeometryColumn` para adicionar a coluna com o tipo espacial. Essa função implementada no PostGIS e especificada no OpenGIS, realiza todo o trabalho de preenchimento da tabela de metadado "`geometry_columns`". Os parâmetros dessa função são:

- nome do banco de dados;
- nome da tabela que irá conter a coluna espacial;
- nome da coluna espacial;
- sistema de coordenadas em que se encontram as geometrias da tabela;
- tipo da coluna espacial, que serve para criar uma restrição que verifica o tipo do objeto sendo inserido na tabela;
- dimensão em que se encontram as coordenadas dos dados.

As tabelas de metadado do PostGIS seguem as especificações da SFSSQL e são representadas pelas seguintes tabelas:

Tabela 8.1 – Tabela de metadado do sistema de coordenadas

spatial_ref_sys		
Attribute	Type	Modifier
`srid`	`INTEGER`	`PK`
`auth_name`	`VARCHAR(256)`	
`auth_srid`	`INTEGER`	
`srtext`	`VARCHAR(2048)`	
`proj4text`	`VARCHAR(2048)`	

Tabela 8.2 – Tabela de metadado das tabelas com colunas espaciais

geometry_columns		
Attribute	Type	**Modifier**
f_table_catalog	VARCHAR(256)	PK
f_table_schema	VARCHAR(256)	PK
f_table_name	VARCHAR(256)	PK
f_geometry_column	VARCHAR(256)	PK
coord_dimension	INTEGER	
srid	INTEGER	FK
type	VARCHAR(30)	

Após criar as tabelas de dados, podemos inserir as informações usando o comando SQL INSERT. Para isso, podemos usar a representação textual das geometrias em conjunto com a função `GeometryFromText` que recebe a representação textual e mais o sistema de coordenadas em que se encontra a geometria:

```
INSERT INTO bairrossp (bairro, spatial_data)
VALUES('JARDIM DOS EUCALIPTOS',
GeometryFromText('POINT(321588.628426 7351166.969244)',
-1));
INSERT INTO drenagemsp (classe, spatial_data)VALUES('RIOS',
GeometryFromText('LINESTRING(344467.895137 7401824.476217,
344481.584686 7401824.518728, 344492.194756 7401825.716359,
…)', -1));

INSERT INTO distritossp (denominacao, sigla, spatial_data)
VALUES('MARSILAC', 'MAR',
GeometryFromText('POLYGON((335589.530575 7356020.721956,
335773.784959 7355873.470174, …))', -1));
```

Podemos também recuperar as informações em cada tabela. Por exemplo, o comando abaixo seleciona a linha do bairro "Vila Mariana":

```
SELECT bairro, AsText(spatial_data) geom FROM bairrossp WHERE
bairro = 'VILA MARIANA';

Resultado:
    bairro    |    geom
--------------+-----------------------------------
 VILA MARIANA | POINT(334667.138663 7388890.076491)
(1 row)       |
```

Aqui, empregamos a função `AsText` para obter a representação textual, pois a partir das versões mais recentes o PostGIS utiliza o formato binário do OpenGIS (WKB) como o padrão nas consultas.

8.3.3.2 Indexação espacial

As colunas com tipos espaciais podem ser indexadas através de uma R-Tree construída no topo do GiST. A sintaxe básica para criação de um índice é a seguinte:

```
CREATE INDEX sp_idx_name ON nome_tabela
  USING GIST (coluna_geometrica GIST_GEOMETRY_OPS);
```

Para as tabelas do nosso exemplo, poderíamos construir os seguintes índices espaciais:

```
CREATE INDEX sp_idx_bairros ON bairrossp USING GIST
(SPATIAL_DATA GIST_GEOMETRY_OPS)

CREATE INDEX sp_idx_bairros ON distritossp USING GIST
(SPATIAL_DATA GIST_GEOMETRY_OPS)

CREATE INDEX sp_idx_bairros ON drenagemsp USING GIST
(SPATIAL_DATA GIST_GEOMETRY_OPS)

CREATE INDEX sp_idx_bairros ON grande_sp USING GIST
(SPATIAL_DATA GIST_GEOMETRY_OPS)
```

Os índices espaciais são usados em consultas que envolvam predicados espaciais, como no caso de consultas por janela, onde um retângulo envolvente é informado e as geometrias que interagem com ele devem ser recuperadas rapidamente.

O operador `&&` pode ser usado para explorar o índice espacial. Por exemplo, para consultarmos os municípios da grande São Paulo que interagem com o retângulo envolvente de coordenadas: 438164.882699,

7435582.150681 e 275421.967006, 7337341.000355, podemos construir a seguinte consulta:

```
SELECT * FROM grande_sp
 WHERE 'BOX3D(438164.882699 7435582.150681,
             275421.967006 7337341.000355)'::box3d
        && spatial_data);
```

Com o uso do operador `&&`, apenas alguns registros precisarão ser pesquisados para responder à pergunta acima.

8.3.3.3 Consultas espaciais

Outro grande destaque desta extensão é o grande número de operadores espaciais disponíveis, entre alguns deles podemos citar:

- Operadores topológicos conforme a Matriz de 9-Interseções DE:

  `equals(geometry, geometry)`

  `disjoint(geometry, geometry)`

  `intersects(geometry, geometry)`

  `touches(geometry, geometry)`

  `crosses(geometry, geometry)`

  `within(geometry, geometry)`

  `overlaps(geometry, geometry)`

  `contains(geometry, geometry)`

  `relate(geometry, geometry)`: retorna a matriz de intersecção.

- Operador de construção de mapas de distância:

  `buffer(geometry, double, [integer])`

- Operador para construção do Fecho Convexo:

  `convexhull(geometry)`

- Operadores de conjunto:

  `intersection(geometry, geometry)`

  `geomUnion(geometry, geometry)`

  `symdifference(geometry, geometry)`

```
difference(geometry, geometry)
```

- Operadores Métricos:

```
distance(geometry,geometry)

area(geometry)
```

- Centróide de geometrias:

```
Centroid(geometry)
```

- Validação (verifica se a geometria possui auto-interseções):

```
isSimple(geometry)
```

O suporte aos operadores espaciais é fornecido através da integração do PostGIS com a biblioteca GEOS (Geometry Engine Open Source) (Refractions, 2003). Essa biblioteca é uma tradução da API Java JTS (Java Topology Suite) (Vivid Solutions, 2003) para a linguagem C++. A JTS é uma implementação de operadores espaciais que seguem as especificações da SFSSQL. Para exemplificar o uso desses operadores e funções, mostraremos os comandos em SQL para executar as consultas dos cenários de 1 a 5, apresentados no início desse capítulo.

Cenário 1 – Usando o operador *touches*, uma possível consulta seria:

```
SELECT d2.nomemunicp
FROM grande_sp d1, grande_sp d2
WHERE touches(d1.spatial_data, d2.spatial_data)
  AND (d2.nomemunicp <> 'SAO PAULO')
  AND (d1.nomemunicp = 'SAO PAULO');
```

Resultado:

nomemunicp	nomemunicp	nomemunicp
COTIA	JUQUITIBA	CAIEIRAS
ITAPECERICA DA SERRA	ITAQUAQUECETUBA	SAO BERNARDO DO CAMPO
EMBU-GUACU	EMBU	DIADEMA
SANTANA DE PARNAIBA	TABOAO DA SERRA	SAO CAETANO DO SUL
SANTO ANDRE	BARUERI	MAUA
GUARULHOS	CAJAMAR	FERRAZ DE VASCONCELOS
MAIRIPORA	OSASCO	POA

(21 rows)

Na consulta acima, o operador *touches* retorna verdadeiro caso as geometrias de *d2* toquem na geometria de São Paulo. Esse é um exemplo

de junção espacial entre duas relações (no nosso caso a mesma relação foi empregada duas vezes). Todas as geometrias da relação *d1*, com exceção da geometria São Paulo, foram avaliadas no teste topológico *touches*, pois o índice espacial não foi empregado. Em tabelas com grandes números de objetos, é importante a utilização desse índice. Ele pode ser explorado empregando-se o operador && em conjunto com os predicados da consulta anterior. Nossa consulta pode ser reescrita como:

```
SELECT d2.nomemunicp
FROM distritossp d1, distritossp d2
WHERE touches(d1.spatial_data, d2.spatial_data)
  AND (d2.nomemunicp <> 'SAO PAULO')
  AND (d1.spatial_data && d2.spatial_data)
  AND (d1.nomemunicp = 'SAO PAULO');
```

Cenário 2 – Novamente iremos empregar o operador *touches*:

```
SELECT grande_sp.nomemunicp
  FROM distritossp, grande_sp
 WHERE touches(distritossp.spatial_data,
               grande_sp.spatial_data)
   AND (distritossp.spatial_data &&
        grande_sp.spatial_data)
   AND (distritossp.denominacao = 'ANHANGUERA');
```

```
Resultado:
nomemunicp          | nomemunicp
--------------------+-----------
BARUERI             | OSASCO
SANTANA DE PARNAIBA | CAIEIRAS
CAJAMAR             |
(5 rows)
```

Cenário 3 – Para este cenário, podemos utilizar o operador *contains*:

```
SELECT COUNT(*)
  FROM bairrossp pt, distritossp pol
 WHERE contains(pol.spatial_data, pt.spatial_data)
   AND (pol.spatial_data && pt.spatial_data)
   AND pol.denominacao = 'GRAJAU';
```

```
Resultado:
count
-----
52
(1 row)
```

Cenário 4 – Aqui empregaremos os operadores *buffer* e *intersects*:

```
SELECT grande_sp.nomemunicp
  FROM grande_sp, drenagemsp
 WHERE intersects(buffer(drenagemsp.spatial_data, 3000),
                         grande_sp.spatial_data)
   AND drenagemsp.cod = 59;
```

Resultado:

Denominação	denominacao	denominacao
AGUA RASA	JAGUARA	SANTANA
ALTO DE PINHEIROS	JAGUARE	SAO DOMINGOS
BARRA FUNDA	LAPA	SE
BELEM	LIMAO	TATUAPE
BOM RETIRO	MOOCA	VILA FORMOSA
BRAS	PARI	VILA GUILHERME
CANGAIBA	PENHA	VILA LEOPOLDINA
CARRAO	PERDIZES	VILA MARIA
CASA VERDE	PIRITUBA	VILA MATILDE
CONSOLACAO	REPUBLICA	VILA MEDEIROS
FREGUESIA DO O	RIO PEQUENO	
JACANA	SANTA CECILIA	

(34 rows)

Cenário 5 – A resposta a essa pergunta poderia ser traduzida, de início, na seguinte consulta:

```
SELECT b1.bairro
  FROM bairrossp b1, bairrossp b2
 WHERE (distance(b1.spatial_data, b2.spatial_data)
                 < 3000)
   AND b1.bairro <> 'BOACAVA'
   AND b2.bairro = 'BOACAVA' order by b1.bairro;
```

Resultado:

Bairro	bairro	bairro
ALTO DA LAPA	PINHEIROS	VILA INDIANA
ALTO DE PINHEIROS	SICILIANO	VILA IPOJUCA
BELA ALIANCA	SUMAREZINHO	VILA LEOPOLDINA
BUTANTA	VILA ANGLO BRASILEIRA	VILA MADALENA
JAGUARE	VILA HAMBURGUESA	VILA RIBEIRO DE BARROS
LAPA	VILA IDA	VILA ROMANA

(18 rows)

No entanto, podemos reescrever essa mesma consulta de forma mais eficiente, utilizando o índice espacial. A estratégia básica é montar um retângulo de 6Km por 6Km centrado no bairro BOACAVA, de forma

que somente seja necessário calcular a distância para os pontos que estejam dentro do retângulo. Reescrevendo a consulta temos:

```
SELECT bl.nome, astext(bl.spatial_data)
  FROM bairros bl, bairros b2
 WHERE (distance(bl.spatial_data, b2.spatial_data)
                 < 3000)
   AND (expand(b2.spatial_data, 3000) &&
        bl.spatial_data)
   AND bl.nome <> 'BOACAVA'
   AND b2.nome = 'BOACAVA' ORDER BY bl.nome;
```

8.4 Oracle Spatial

Oracle Spatial (Murray, 2003) é uma extensão espacial desenvolvida sobre o modelo objeto-relacional do SGDB Oracle. Este modelo permite definir novos tipos de dados através da linguagem de definição de dados SQL DDL, e implementar operações sobre esses novos tipos, através da linguagem PL/SQL (Urman, 2002), uma extensão da SQL (Lassen et al, 1998). Esta extensão é baseada nas especificações do OpenGIS e contém um conjunto de funcionalidades e procedimentos que permitem armazenar, acessar, modificar e consultar dados espaciais de representação vetorial.

O Oracle Spatial é formado pelos seguintes componentes:

- Um modelo próprio de dados chamado MDSYS que define a forma de armazenamento, a sintaxe e semântica dos tipos espaciais suportados.
- Mecanismo de indexação espacial.
- Um conjunto de operadores e funções para representar consultas, junção espacial e outras operações de análise espacial.
- Aplicativos administrativos.

8.4.1 Tipos de dados espaciais

O modelo de dados do Spatial consiste em uma estrutura hierárquica de elementos, geometrias e planos de informação (*layers*). Cada plano é formado por um conjunto de geometrias, que por sua vez são formadas por um conjunto de elementos.

Cada elemento é associado a um tipo espacial primitivo, como ponto, linha ou polígono (com ou sem ilhas), os quais são mostrados na Figura 8.11.

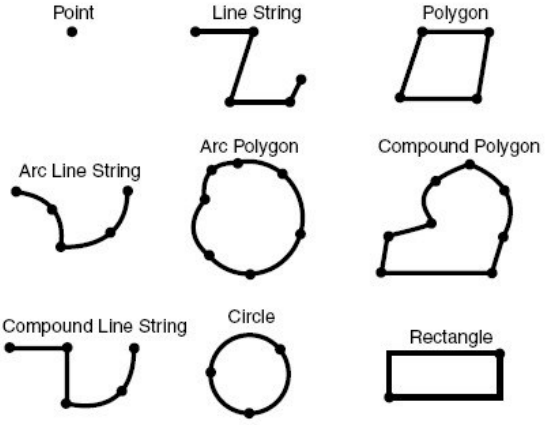

Figura 8.11 – Tipos espaciais primitivos do Oracle Spatial.

Os tipos espaciais bidimensionais são compostos por pontos formados por duas ordenadas X e Y, freqüentemente correspondentes à longitude e latitude. A extensão também suporta o armazenamento e indexação de tipos tridimensionais e tetradimensionais, mas as funções e operadores só funcionam para os tipos bidimensionais.

Uma geometria pode ser formada por um único elemento, ou por um conjunto homogêneo (multipontos, multilinhas ou multipolígonos) ou heterogêneo (coleção) de elementos. Um plano de informação é formado por uma coleção de geometrias que possuem um mesmo conjunto de atributos.

Baseado no modelo objeto-relacional, o Spatial define um tipo de objeto, para representar dados espaciais, chamado `SDO_GEOMETRY`, como mostrado a seguir.

```
CREATE TYPE sdo_geometry AS OBJECT (
SDO_GTYPE           NUMBER,
SDO_SRID            NUMBER,
```

```
SDO_POINT           SDO_POINT_TYPE,
SDO_ELEM_INFO       SDO_ELEM_INFO_ARRAY,
SDO_ORDINATES       SDO_ORDINATE_ARRAY);
```

Este objeto contém a geometria em si, suas coordenadas, e informações sobre seu tipo e projeção. Em uma tabela espacial, os atributos alfanuméricos da geometria são definidos como colunas de tipos básicos (VARCHAR2, NUMBER, DATE, dentre outros), e a geometria como uma coluna do tipo SDO_GEOMETRY. Em uma tabela espacial, cada instância do dado espacial é armazenada em uma linha, e o conjunto de todas as instâncias dessa tabela forma um plano de informação.

O objeto SDO_GEOMETRY é composto pelos seguintes atributos:

- SDO_GTYPE: formado por quatro números, onde os dois primeiros indicam a dimensão da geometria e os outros dois o seu tipo. Os tipos podem ser: 00 (não conhecido), 01 (ponto), 02 (linha ou curva), 03 (polígono), 04 (coleção), 05 (multipontos), 06 (multilinhas) e 07 (multipolígonos);
- SDO_SRID: utilizado para identificar o sistema de coordenadas, ou sistema de referência espacial, associado à geometria;
- SDO_POINT: é definido utilizando um objeto do tipo SDO_POINT_TYPE, que contém os atributos X, Y e Z para representar as coordenadas de um ponto. Somente é preenchido se a geometria for do tipo ponto, ou seja, se os dois últimos números do SDO_GTYPE forem iguais a "01";
- SDO_ELEM_INFO: é um vetor de tamanho variável que armazena as características dos elementos que compõem a geometria. As coordenadas de cada elemento são armazenadas em um vetor variável chamado SDO_ORDINATES e são interpretadas através de três números armazenados no SDO_ELEM_INFO:
    - SDO_STARTING_OFFSET: indica qual a posição da primeira coordenada do elemento no SDO_ORDINATES;
    - SDO_ETYPE: indica o tipo do elemento;
    - SDO_INTERPRETATION: indica como o elemento deve ser interpretado juntamente com o SDO_ETYPE.

- SDO_ORDINATES: é um vetor de tamanho variável que armazena os valores das coordenadas da geometria.

Como exemplo, mostraremos os comandos em SQL para gerar tabelas para armazenar os atributos e as geometrias:

- Dos distritos de São Paulo, mostrados na Figura 8.1

```
CREATE TABLE  DistritosSP (
cod                 NUMBER(32) NOT NULL ,
sigla               VARCHAR2(20),
denominacao         VARCHAR2(200),
spatial_data        MDSYS.SDO_GEOMETRY,
PRIMARY KEY (cod))
```

- Dos bairros de São Paulo, mostrados na Figura 8.2.

```
CREATE TABLE BairrosSP (
geom_id             NUMBER(32) NOT NULL,
bairro              VARCHAR2(200),
distr               VARCHAR2(200),
spatial_data        MDSYS.SDO_GEOMETRY,
PRIMARY KEY (geom_id))
```

- Do mapa de drenagem, mostrado na Figura 8.3

```
CREATE TABLE DrenagemSP (
geom_id          NUMBER(32) NOT NULL,
classe           VARCHAR2(100)  NULL,
spatial_data     MDSYS.SDO_GEOMETRY,
PRIMARY KEY (geom_id))
```

As tabelas criadas anteriormente contêm colunas de tipos básicos como NUMBER e VARCHAR2 para armazenar atributos, e uma coluna do tipo SDO_GEOMETRY para armazenar geometrias. Note que não é preciso especificar, na criação, o tipo de geometria que será armazenado.

Mostramos a seguir alguns exemplos de comandos SQL para inserir dados nessas tabelas criadas:

```
INSERT INTO DistritosSP (cod, sigla, denominacao,
spatial_data)  VALUES (1, 'VMR', 'VILA MARIA'
MDSYS.SDO_GEOMETRY(2003, NULL, NULL,
MDSYS.SDO_ELEM_INFO_ARRAY( 1, 1003, 1 ),
MDSYS.SDO_ORDINATE_ARRAY(6,10, 10,1, 14,10, 10,14, 6,10)))
```

```
INSERT INTO DrenagemSP ( geom_id, classe, spatial_data)
VALUES (1, 'RIO',
MDSYS.SDO_GEOMETRY(2002, NULL, NULL,
MDSYS.SDO_ELEM_INFO_ARRAY( 1, 2, 1 ),
MDSYS.SDO_ORDINATE_ARRAY(10,10, 10,14, 6,10, 14,10)))

INSERT INTO BairrosSP ( geom_id, bairro, distr, spatial_data)
VALUES ( 1, 'JARDIM SHANGRILA', 'GRAJAU'
MDSYS.SDO_GEOMETRY(2001, NULL,
MDSYS.SDO_POINT_TYPE(32.628, 736.944, NULL ), NULL, NULL))
```

Observe que para incluir uma geometria através de um comando SQL é preciso montar um objeto `SDO_GEOMETRY`. Através do atributo `SDO_GTYPE`, podemos identificar qual o tipo da geometria contida no `SDO_GEOMETRY`. A geometria inserida na tabela DistritosSP é um polígono (`SDO_GTYPE=2003`), na DrenagemSP é uma linha (`SDO_GTYPE=2002`) e na BairrosSP é um ponto (`SDO_GTYPE=2001`), todos bidimensionais. Além disso, todas as geometrias são formadas por um único elemento cujo tipo pode ser identificado através dos atributos `SDO_ETYPE` e `SDO_INTERPRETATION`.

A Tabela 8.3 apresenta os tipos de elementos possíveis. Note que a primeira coordenada de um polígono tem que ser igual à última, e seus anéis externos (`SDO_ETYPE=1003`) devem estar no sentido anti-horário, e os internos (`SDO_ETYPE=2003`), no sentido horário. Nesse exemplo não estamos usando o sistema de coordenadas fornecido pela extensão (`SDO_SRID=NULL`).

Através de um comando SQL simples, é possível recuperar o objeto `SDO_GEOMETRY`. A seguir mostraremos uma consulta e o resultado retornado pelo Spatial:

```
SELECT spatial_data FROM DistritosSP
WHERE denominacao = 'PARELHEIROS'
Resultado:
SPATIAL_DATA
-----------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL,
SDO_ELEM_INFO_ARRAY(1, 1003, 1),
SDO_ORDINATE_ARRAY(370139,955, 7408390,5, 369854,277,
7407971,06, 370014,832, 7408419,18, 370139,955, 7408390,5, 369854,277,
7407971,06, 370014,832, 7408419,18, 370139,955, 7408390,5))
```

Tabela 8.3 – Tipos de elementos espaciais

SDO_ ETYPE	SDO_ INTERPRETATION	***Descrição do tipo***
0	Nenhum valor	Tipo não suportado
1	1	Ponto
1	n>1	Conjunto de n pontos
2	1	Linha formada por vértices conectados por segmentos retos
2	2	Linha formada por vértices conectados por arcos circulares
1003 ou 2003	1	Polígono simples composto por vértices conectados por segmentos retos
1003 ou 2003	2	Polígono simples composto por vértices conectados por arcos circulares
1003 ou 2003	3	Retângulo otimizado composto por dois pontos
1003 ou 2003	4	Círculo formado por três pontos da circunferência
4	n>1	Linha composta por alguns vértices conectados por segmentos de reta e outros por arcos
1005 ou 2005	n>1	Polígono composto por alguns vértices conectados por segmentos de reta e outros por arcos

O modelo MDSYS apresenta dois conjuntos de tabelas de metadados que são utilizadas por funcionalidades internas da extensão, como por exemplo, nas consultas espaciais:

- Tabelas de metadados sobre geometrias armazenadas, chamadas `USER_SDO_GEOM_METADATA` e `ALL_SDO_GEOM_METADATA`.
- Tabelas de metadados sobre indexação espacial, chamadas `USER_SDO_INDEX_METADATA` e `ALL_SDO_INDEX_INFO`.

As tabelas de metadados sobre geometrias armazenam, para cada tabela espacial: o seu nome (TABLE_NAME); o nome da coluna de tipo geométrico (COLUMN_NAME); todas as dimensões das geometrias, cada uma com um mínimo retângulo envolvente e uma tolerância (DIMINFO); e o sistema de coordenadas (SRID).

Sua estrutura é mostrada a seguir:

```
TABLE_NAME    VARCHAR2(32),
COLUMN_NAME          VARCHAR2(32),
DIMINFO              SDO_DIM_ARRAY,
SRID                 NUMBER
```

Após criar e inserir os dados em uma tabela espacial, o usuário deve registrar seu metadado. A seguir, mostraremos um exemplo de um comando em SQL para inserir o metadado referente à tabela espacial DistritosSP criada anteriormente.

```
INSERT INTO USER_SDO_GEOM_METADATA
VALUES ( 'DistritosSP' ,'spatial_data' ,
MDSYS.SDO_DIM_ARRAY(
MDSYS.SDO_DIM_ELEMENT('X',275.9670,429.567,0.0005),
MDSYS.SDO_DIM_ELEMENT('Y',833.0355,582.15,0.0005)),
NULL)
```

Note que as geometrias armazenadas na tabela DistritosSP possuem duas dimensões X e Y, portanto, existem duas entradas no vetor SDO_DIM_ARRAY, uma para cada dimensão contendo seu mínimo retângulo envolvente e sua tolerância.

8.4.2 Indexação espacial

Indexação espacial, como qualquer outro tipo de indexação, fornece mecanismos para limitar o conjunto de busca, aumentando assim a performance das consultas e da recuperação dos dados espaciais. O Spatial fornece dois tipos de indexação espacial, R-tree e Quadtree, podendo ser utilizados simultaneamente. Porém, o Oracle recomenda fortemente o uso de R-tree ao invés de Quadtree, por causa do seu desempenho. Mais informações sobre tipos de indexação espacial podem ser encontrados no Capítulo 6.

O usuário pode criar uma R-tree utilizando os parâmetros default do MDSYS ou especificando cada parâmetro como, por exemplo, o tamanho da memória utilizada e o número de dimensões a serem indexadas. A

sintaxe de um comando em SQL pra gerar uma R-tree com parâmetros default do `MDSYS` é:

```
CREATE INDEX index_name ON table_name (spatial_column_name)
INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

Como exemplo, mostraremos os comandos em SQL usados para gerar os índices espaciais das tabelas criadas anteriormente:

```
CREATE INDEX DistritosSP_IDX ON
DistritosSP(SPATIAL_DATA) INDEXTYPE IS
MDSYS.SPATIAL_INDEX

CREATE INDEX DrenagemSP_IDX ON
DrenagemSP (SPATIAL_DATA)
INDEXTYPE IS MDSYS.SPATIAL_INDEX

CREATE INDEX BairrosSP_IDX ON BairrosSP(SPATIAL_DATA)
INDEXTYPE IS MDSYS.SPATIAL_INDEX
```

Ao inserir, remover ou modificar geometrias de uma tabela, a performance do índice espacial gerado inicialmente pode ser degradada. Para isso, a extensão fornece um conjunto de funções para avaliar a performance dos índices, como a função `SDO_TUNE.QUALITY_DEGRADATION`, e para reconstruí-lo, usando o comando `ALTER INDEX REBUILD`.

Após a criação de índices espaciais, a extensão atualiza, automaticamente, as tabelas de metadados sobre indexação citadas na Seção 8.2.1.1. Essas tabelas são mantidas pela extensão e não devem ser alteradas pelos usuários.

8.4.3 Consultas espaciais

O Oracle Spatial utiliza um modelo de consulta baseado em duas etapas, chamadas de primeiro e segundo filtro, como mostrado na Figura 8.12. O primeiro filtro considera as aproximações das geometrias, pelo critério do mínimo retângulo envolvente (MBR), para reduzir a complexidade computacional. Este filtro é de baixo custo computacional e seleciona um subconjunto menor de geometrias candidatas, que será passado para o segundo filtro. O segundo filtro trabalha com as geometrias exatas, por isso é computacionalmente mais caro e só é aplicado ao subconjunto resultante do primeiro filtro. Retorna o resultado exato da consulta.

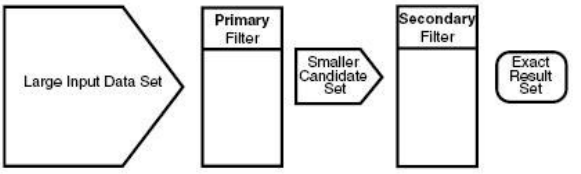

Figura 8.12 – Modelo de Consulta.

Para executar consultas e operações espaciais, o Oracle Spatial fornece um conjunto de operadores e funções que são utilizados juntamente com a linguagem SQL.

Os operadores, alguns mostrados na Tabela 8.4, são usados na cláusula WHERE e utilizam indexação espacial. Portanto, só podem ser executados sobre colunas espaciais já indexadas. As funções, algumas mostradas na Tabela 8.5, são definidas como subprogramas em PL/SQL, e utilizadas na cláusula WHERE ou em subconsultas, podendo ser executadas sobre colunas espaciais não indexadas. Devido ao fato dos operadores sempre explorarem a indexação, é recomendável usá-los, em lugar de funções, quando possível.

Tabela 8.4 – Principais operadores espaciais

Operador	*Descrição*
`SDO_FILTER`	Implementa o primeiro filtro do modelo de consulta, ou seja, verifica se os mínimos retângulos envolventes das geometrias têm alguma interação entre si.
`SDO_RELATE`	Avalia se as geometrias possuem uma determinada relação topológica.
`SDO_WITHIN_DISTANCE`	Verifica se duas geometrias estão dentro de uma determinada distância.
`SDO_NN`	Identifica os *n* vizinhos mais próximos de uma geometria

A sintaxe de cada operador é mostrada a seguir:

```
SDO_FILTER    (geometry1 SDO_GEOMETRY,
               geometry2 SDO_GEOMETRY)

SDO_RELATE (geometry1 SDO_GEOMETRY,
               geometry2 SDO_GEOMETRY,
               param VARCHAR2)

SDO_WITHIN_DISTANCE (geometry1  SDO_GEOMETRY,
                           aGeom   SDO_GEOMETRY,
                           params  VARCHAR2);

SDO_NN (geometry1   SDO_GEOMETRY,
          aGeom    SDO_GEOMETRY, param  VARCHAR2,
       [, number NUMBER]);
```

Os operadores `SDO_FILTER` e `SDO_RELATE` recebem os parâmetros em comum:

- `geometry1`: uma coluna do tipo `SDO_GEOMETRY` de uma tabela que deve estar indexada.
- `geometry2`: um objeto do tipo `SDO_GEOMETRY`. Esse objeto pode ser uma instância em memória ou estar armazenado em alguma tabela espacial, que não precisa ser indexada.

Além dos parâmetros citados anteriormente, o `SDO_RELATE` recebe ainda o tipo da relação topológica a ser computada (`param`), que pode ser `EQUAL`, `DISJOINT`, `TOUCH`, `INSIDE`, `COVERS`, `COVERREDBY`, `OVERLAPBDYINTERSECT`, `ON`, `CONTAINS`, `OVERLAPBDYDISJOINT` e `ANYINTERACT`. Este operador é baseado no Modelo de 9-Interseções mostrado no capitulo 2.

Os operadores `SDO_WITHIN_DISTANCE` e SDO_NN recebem os parâmetros em comum:

- `geometry1`: uma coluna do tipo `SDO_GEOMETRY` de uma tabela que deve estar indexada.
- `aGeom`: uma instância do tipo `SDO_GEOMETRY`.

Além dos parâmetros em comum, os operadores `SDO_WITHIN_DISTANCE` e `SDO_NN` recebem outros, como a distância a ser

considerada e o número de vizinhos que devem ser retornados. A extensão ainda fornece alguns operadores que não serão abordados neste capitulo, como por exemplo, SDO_JOIN, SDO_TOUCH, SDO_INSIDE, SDO_ON, dentre outros.

As funções fornecidas pelo Spatial podem ser agrupadas em:

- Relação (verdadeiro/falso) entre duas geometrias:
  RELATE e WITHIN_DISTANCE.
- Validação:
  VALIDATE_GEOMETRY_WITH_CONTEXT,
  VALIDATE_LAYER_WITH_CONTEXT.
- Operações sobre uma geometria:
  SDO_ARC_DENSIFY, SDO_AREA, SDO_BUFFER, SDO_CENTROID,
  SDO_CONVEXHULL, SDO_LENGTH, SDO_MAX_MBR_ORDINATE,
  SDO_MIN_MBR_ORDINATE, SDO_MBR, SDO_POINTONSURFACE.
- Operações sobre duas geometrias:
  SDO_DISTANCE, SDO_DIFFERENCE, SDO_INTERSECTION,
  SDO_UNION, SDO_XOR.

Tabela 8.5 – Algumas funções espaciais.

Função	Descrição
SDO_BUFFER	Gera uma nova geometria ao redor ou dentro de uma outra, considerando uma distância passada como parâmetro.
SDO_AREA SDO_ LENGTH	Calculam, respectivamente, a área e o perímetro ou comprimento de uma geometria.
SDO_DISTANCE	Calcula a distância entre duas geometrias.
SDO_INTERSECTION SDO_UNION SDO_DIFFERENCE	Geram uma nova geometria resultante da interseção, união e diferença, respectivamente, entre outras duas.

Para exemplificar o uso desses operadores e funções, mostraremos os comandos em SQL para executar as consultas dos cenários de 1 a 5, apresentados no início desse capítulo.

Cenário 1 – Essa consulta é realizada sobre as geometrias da tabela espacial MunicipiosSP. Para respondê-la, podemos usar tanto o operador SDO_RELATE quanto o operador SDO_TOUCH na cláusula WHERE da SQL:

```
SELECT t1.nomemunicp
FROM  MunicipiosSP t1, MunicipiosSP t2
WHERE SDO_RELATE   (t1.spatial_data,
t2.spatial_data, 'mask=TOUCH') = 'TRUE'
AND   t2.nomemunicp = 'SAO PAULO'

SELECT t1.nomemunicp
FROM MunicipiosSP t1, MunicipiosSP t2
WHERE
SDO_TOUCH (t1.spatial_data, t2.spatial_data) =
'TRUE' AND   t2.nomemunicp = 'SAO PAULO'
```

Resultado:

nomemunicp	nomemunicp	nomemunicp
COTIA	JUQUITIBA	CAIEIRAS
ITAPECERICA DA SERRA	ITAQUAQUECETUBA	SAO BERNARDO DO CAMPO
EMBU-GUACU	EMBU	DIADEMA
SANTANA DE PARNAIBA	TABOAO DA SERRA	SAO CAETANO DO SUL
SANTO ANDRE	BARUERI	MAUA
GUARULHOS	CAJAMAR	FERRAZ DE VASCONCELOS
MAIRIPORA	OSASCO	POA

(21 rows)

Cenário 2 – Essa consulta é realizada sobre as geometrias de duas tabelas espaciais distintas, MunicipiosSP e DistritosSP. Podemos usar tanto o operador SDO_RELATE quanto o operador SDO_TOUCH na cláusula WHERE:

```
SELECT t1.nomemunicp
FROM  MunicipiosSP t1, DistritosSP t2
WHERE SDO_RELATE (t1.spatial_data, t2.spatial_data,
    'mask= TOUCH') = 'TRUE'
AND   t2.denominacao = 'ANHANGUERA'
SELECT t1.nomemunicp
FROM  MunicipiosSP t1, DistritosSP t2
WHERE SDO_TOUCH (t1.spatial_data, t2.spatial_data) = 'TRUE'
AND   t2.denominacao = 'ANHANGUERA'
```

```
Resultado:
nomemunicp              | nomemunicp
------------------------+-----------
BARUERI                 | OSASCO
SANTANA DE PARNAIBA     | CAIEIRAS
CAJAMAR
(5 rows)
```

Cenário 3 – Essa consulta utiliza a função de agregação COUNT juntamente com o operador SDO_RELATE ou SDO_INSIDE.

```
SELECT COUNT(*)
FROM BairrosSP t1, DistritosSP t2
WHERE SDO_RELATE (t1.spatial_data, t2.spatial_data,
'mask=INSIDE') = 'TRUE'
AND   t2.denominacao = 'GRAJAU'

SELECT COUNT(*)
FROM BairrosSP t1, DistritosSP t2
WHERE      SDO_INSIDE (t1.spatial_data, t2.spatial_data) =
'TRUE'  AND   t2.denominacao = 'GRAJAU'

Resultado:
count
-----
52
(1 row)
```

Cenário 4 – Para realizar essa consulta utilizamos dois operadores espaciais, SDO_BUFFER e SDO_RELATE. Para gerar o buffer, o operador usará a tolerância definida na tabela de metadados. Observe que o operador SDO_RELATE recebe como parâmetro a combinação de três relações topológicas.

```
SELECT t1.denominacao
FROM   DistritosSP t1, DreanagemSP t2,
       user_sdo_geom_metadata m
WHERE SDO_RELATE (t1.spatial_data,
       SDO_GEOM.SDO_BUFFER(t2.spatial_data, m.diminfo, 3000),
       'mask=INSIDE+TOUCH+ OVERLAPBDYINTERSECT')
       = 'TRUE'
       AND    m.table_name = ' DreanagemSP '
       AND    m.column_name = 'spatial_data'
       AND    t2.geom_id = 55
Resultado:
denominacao          | denominacao | denominacao
---------------------+-------------+------------
AGUA RASA            | JAGUARA     | SANTANA
```

ALTO DE PINHEIROS	JAGUARE	SAO DOMINGOS
BARRA FUNDA	LAPA	SE
BELEM	LIMAO	TATUAPE
BOM RETIRO	MOOCA	VILA FORMOSA
BRAS	PARI	VILA GUILHERME
CANGAIBA	PENHA	VILA LEOPOLDINA
CARRAO	PERDIZES	VILA MARIA
CASA VERDE	PIRITUBA	VILA MATILDE
CONSOLACAO	REPUBLICA	VILA MEDEIROS
FREGUESIA DO O	RIO PEQUENO	
JACANA	SANTA CECILIA	

(34 rows)

Cenário 5 – Esta consulta utiliza a função SDO_DISTANCE na cláusula WHERE.

```
SELECT t1.BAIRRO
FROM BairrosSP t1, BairrosSP t2
WHERE SDO_GEOM.SDO_DISTANCE (t1.spatial_data,
t2.spatial_data, 0.00005) < 3000
AND    t2.bairro = 'BOACAVA'
```

Resultado:

bairro	bairro	bairro
ALTO DA LAPA	PINHEIROS	VILA INDIANA
ALTO DE PINHEIROS	SICILIANO	VILA IPOJUCA
BELA ALIANCA	SUMAREZINHO	VILA LEOPOLDINA
BUTANTA	VILA ANGLO BRASILEIRA	VILA MADALENA
JAGUARE	VILA HAMBURGUESA	VILA RIBEIRO DE BARROS
LAPA	VILA IDA	VILA ROMANA

(18 rows)

8.5 Leituras suplementares

Além das extensões apresentadas neste capítulo, ainda temos o IBM DB2 Spatial Extender (IBM, 2002), o Informix Spatial e Geodetic Datablade (IBM, 2003) e a extensão do MySQL (Yarger et al, 1999).

A extensão espacial deste último SGBD encontra-se em construção (MySQL AB, 2003). O seu projeto segue as especificações da SFSSQL, e os únicos recursos disponibilizados até o momento são os tipos de dados espaciais semelhantes aos fornecidos pelo PostGIS (Point, LineString,

Polygon, MultiLineString, MultiPolygon, MultiPoint e GeometryCollection) e o mecanismo de indexação através de R-Tree.

O quadro abaixo apresenta um resumo comparativo entre os SGBDs com suporte espacial:

Tabela 8.6 – Quadro Comparativo entre os SGBDs com Suporte Espacial.

Recurso	*Oracle Spatial*	*PostgreSQL com Tipos Geométricos*	*PostgreSQL com PostGIS*	*MySQL*
Tipos espaciais	SFSSQL	Tipos simples	SFSSQL	SFSSQL
Indexação espacial	R-Tree e QuadTree	R-Tree nativa ou R-Tree sobre GiST	R-Tree sobre GiST	R-Tree
Operadores topológicos	Matriz 9-Interseções	Não	Matriz 9-Interseções DE	Em desenv.
Operadores de conjuntos	Sim	Não	Sim	Em desenv.
Operador de *buffer region*	Sim	Não	Sim	Em desenv.
Transformação entre sistemas de coordenadas	Sim	Não	Sim	Não
Tabelas de metadados das colunas geométricas	Sim (diferente do OGIS)	Não	Sim (conforme OGIS)	Não

Referências

HELLERSTEIN, J. M.; NAUGHTON, J. F.; PFEFFER, A. Generalized search trees for databases systems. In: international Conference in VLDB, 21., set. 1995, Zurich, Switzerland. **Proceeding.** San Francisco: Morgan Kaufman, 1995, 562-573.

IBM. **IBM DB2 Spatial Extender:** user's guide and reference. Disponível em: <http://www.ibm.com.br>. Acesso em: jun. 2002.

IBM. **Working with the geodetic and spatial datablade modules**. Disponível em: <http://www.ibm.com>. Acesso em: out. 2003.

KERNIGHAN, B. W.; RITCHIE, D. M. **C: a linguagem de programação -** padrão ANSI. Brasil: Campus, 1990.

LASSEN, A. R. O., J.; OSTERBYE, K., 1998, Object Relational Modeling, Centre for Object Technology (COT).

MURRAY, C., 2003, Oracle® Spatial User's Guide and Reference 10g Release 1 (10.1), Redwood City, Oracle Corporation, p. 602.

MySQL AB. **MySQL reference manual**. Disponível em: <http://www.mysql.com>. Acesso em: nov. 2003.

RAMSEY, P. **PostGIS manual**. Disponível em: <http://postgis.refractions.net/documentation>. Acesso em: jan. 2002.

RAVADA, S.; SHARMA, J. Oracle8i Spatial: experiences with extensible databases. In: International Symposium on Spatial Databases, 6., jul. 1999, Hong Kong, China. **Proceedings…** Berlin: Springer-Verlag, 1999, p. 355-359.

REFRACTIONS Inc. **GEOS API documentation**. Disponível em: <http://geos.refrcations.net>. Acesso em: nov. 2003.

STONEBRAKER, M.; ROWE, L. A.; HIROHAMA, M. The implementation of Postgres. **IEEE Transactions on Knowledge and Data Engineering**, v. 2, n. 1, p. 125-142, mar. 1990.

URMAN, S. **Oracle 9i Programacao PL/SqL.** Rio de Janeiro: Editora Campos, 2002. 552 p.

VIVID SOLUTIONS. **JTS technical specifications**. Disponível em: <http://www.vividsolutions.com/jts/jtshome.htm>. Acesso em: nov. 2003.

YARGER, R. J.; REESE, G.; KING, T. **MySQL & mSQL**. EUA: O'Reilly, 1999.