MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-12837-PRE/8127

CAPÍTULO 12

DESCRIÇÃO DA TERRALIB

Lúbia Vinhas
Karine Reis Ferreira

Bancos de dados geográficos

INPE
São José dos Campos
2005

12 Descrição da TerraLib

Lúbia Vinhas
Karine Reis Ferreira

12.1 Introdução

Esse capítulo descreve a biblioteca TerraLib em seus aspectos mais relevantes em termos de bancos de dados geográficos, incluindo o modelo conceitual do banco de dados, o modelo de armazenamento de geometrias e dados descritivos, e os mecanismos de manipulação do banco em diferentes níveis de abstração.

A TerraLib é um projeto de software livre que permite o trabalho colaborativo entre a comunidade de desenvolvimento de aplicações geográficas, servindo desde à prototipação rápida de novas técnicas até o desenvolvimento de aplicações colaborativas. Sua distribuição é feita através da Web no site www.terralib.org.

TerraLib é uma biblioteca de classes escritas em C++ para a construção de aplicativos geográficos, com código fonte aberto e distribuída como um software livre. Destina-se a servir como base para o desenvolvimento cooperativo na comunidade de usuários ou desenvolvedores de SIG's – Sistemas de Informação Geográfica.

TerraLib fornece funções para a decodificação de dados geográficos, estruturas de dados espaço-temporais, algoritmos de análise espacial além de propor um modelo para um banco de dados geográficos (Câmara et al. 2002). A arquitetura da biblioteca é mostrada na Figura 12.1. Existe um módulo central, chamado *kernel*, composto de estruturas de dados espaço-temporais, suporte a projeções cartográficas, operadores espaciais e uma interface para o armazenamento e recuperação de dados espaço-temporais em bancos de dados objeto-relacionais, além de mecanismos de controle de vizualização. Em um módulo composto de *drivers* a interface de

recuperação e armazenamento é implementada. Esse módulo também contém rotinas de decodificação de dados geográficos em formatos abertos e proprietários. Funções de análise espacial são implementadas utilizando as estruturas do *kernel*. Finalmente, sobre esses módulos podem ser construídas diferentes interfaces aos componentes da TerraLib em diferentes ambientes de programação (Java, COM, C++) inclusive para a implementação de serviços OpenGIS como o WMS – Web Map Server (OGIS, 2005).

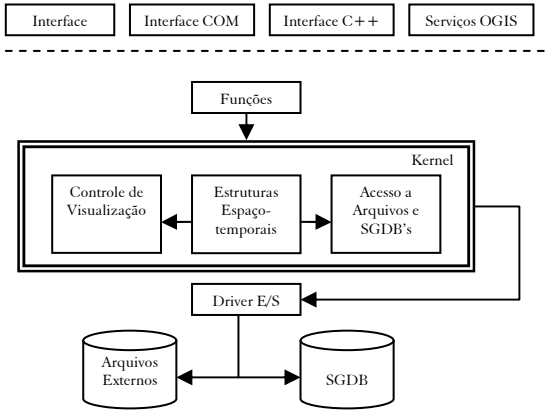


Figura 12.1 – Arquitetura da TerraLib.

Uma das características mais importantes da TerraLib é a sua capacidade de integração a sistemas de gerenciamento de bancos de dados objeto-relacionais (SGBD-OR) para armazenar os dados geográficos, tanto sua componente descritiva quanto sua componente espacial. Essa integração é o que permite o compartilhamento de grandes bases de dados, em ambientes coorporativos, por aplicações customizadas para diferentes tipos de usuários. A TerraLib trabalha em um modelo de arquitetura em camadas (Davis e Câmara, 2001), funcionando como a camada de acesso entre o banco e a aplicação final.

Como exemplo de um aplicativos geográfico construído sobre a TerraLib, podemos citar o TerraView (INPE/DPI, 2005). Na Figura 12.2 ilustramos como o TerraView utiliza a TerraLib como camada de acesso a

um banco de dados sob o controle de um SGDB-OR como o MySQL (MYSQL, 2005), por exemplo.

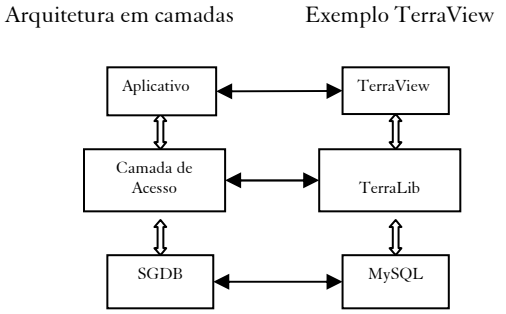


Figura 12.2 – Exemplo de uso da TerraLib como camada de acesso ao banco de dados.

Enquanto biblioteca de software, a TerraLib é compilada em um ambiente multiplataforma, Windows e Linux, e em diferentes compiladores C++. A TerraLib usa extensivamente os mecanismos mais atuais da linguagem C++, como a STL – Standard Template Libray, classes parametrizadas e programação multi-paradigma (Stroustroup, 1997).

Esse capítulo é ilustrado com trechos de código C++ que utilizam a TerraLib. Vale lembrar, que esses exemplos, por questão de clareza, contêm apenas os trechos mais relevantes para ilustrar a informação sendo destacada. Maiores detalhes sobre as classes, estruturas de dados e funções da biblioteca, usadas nos exemplos, podem ser obtidos na documentação de código da TerraLib disponível em www.terralib.org.

12.2 Modelo conceitual

A TerraLib propõe não somente um modelo de armazenamento de dados geográficos em um SGBD-OR, mas também um modelo conceitual de banco de dados geográfico, sobre o qual são escritos seus algoritmos de processamento. As entidades que formam o modelo conceitual são:

Banco de Dados – representa um repositório de informações contendo tanto os dados geográficos quanto o seu modelo de organização. Um banco de dados pode ser materializado em diferentes SGDB's – Sistemas Gerenciadores de Bancos de Dados, comerciais ou de domínio público. O único requisito da TerraLib é que o SGDB possua a capacidade de armazenar campos binários longos, ou uma extensão própria capaz de criar tipos abstratos espaciais, e que possa ser acessado por alguma camada de *software*.

Layer – um *layer* representa uma estrutura de agregação de um conjunto de informações espaciais que são localizadas sobre uma região geográfica e compartilham um conjunto de atributos, ou seja, um *layer* agrega coisas semelhantes. Como exemplos de *layers* podem ser citados os mapas temáticos (mapa de solos, mapa de vegetação), os mapas cadastrais de objetos geográficos (mapa de municípios do Distrito Federal) ou ainda dados matriciais como cenas de imagens de satélites. Independentemente da representação computacional adotada para tratar o dado geográfico, matricial ou vetorial, um *layer* conhece qual a projeção cartográfica da sua componente espacial.

Layers são inseridos no banco de dados através da importação de arquivos de dados geográficos em formatos de intercâmbio como shapefiles, ASCII-SPRING, MID/MIF, GeoTiff, JPEG ou dbf. A biblioteca fornece as rotinas de importação desses arquivos. *Layers* também podem ser gerados a partir de processamentos executados sobre outros *layers* já armazenados no banco.

Representação – trata do modelo de representação da componente espacial dos dados de um *layer* e pode ser do tipo vetorial ou matricial. Na representação vetorial, a TerraLib distingue entre representações formadas por pontos, linhas ou áreas e também outras representações mais complexas formadas a partir dessas como células e redes.

Para representações matriciais, a TerraLib suporta a representação de grades regulares multi-dimensionais.

A TerraLib permite que um mesmo geo-objeto de um *layer* possua diferentes representações vetoriais (ex. um município pode ser representado pelo polígono que define os seus limites, bem como pelo ponto onde está localizado em sua sede). A entidade representação da

TerraLib guarda informações como o seu menor retângulo envolvente ou a resolução horizontal e vertical de uma representação matricial.

O termo representação espacial, no contexto da TerraLib, é muitas vezes usado de maneira análoga ao termo geometria.

Projeção Cartográfica – serve para representar a referência geográfica da componente espacial dos dados geográficos. As projeções cartográficas permitem projetar a superfície terrestre em uma superfície plana. Diferentes projeções são usadas para minimizar as diferentes deformações inerentes ao processo de projeção de um elipsóide em um plano. Cada projeção é definida a partir de certo número de parâmetros como o Datum planimétrico de referência, paralelos padrão e deslocamentos (Snyder, 1987).

Tema – serve principalmente para definir uma seleção sobre os dados de um *layer*. Essa seleção pode ser baseada em critérios a serem atendidos pelos atributos descritivos do dado e/ou sobre a sua componente espacial.

Um tema também define o visual, ou a forma de apresentação gráfica da componente espacial dos objetos do tema. Para o caso de dados com uma representação vetorial a componente espacial é composta de elementos geométricos como pontos, linhas ou polígonos. Para os dados com uma representação matricial, sua componente espacial está implícita na estrutura de grade que a define, regular e com um espaçamento nas direções X e Y do plano cartesiano.

Os temas podem definir também formas de agrupamento dos dados de um *layer*, gerando grupos, os quais possuem legendas que os caracterizam.

Vista – serve para definir uma visão particular de um usuário sobre o banco de dados. Uma vista define quais temas serão processados ou visualizados conjuntamente. Além disso, como cada tema é construído sobre um *layer* com sua própria projeção geográfica, a vista define qual será a projeção comum para visualizar ou processar os temas que agrega.

Visual – um visual representa um conjunto de características de apresentação de primitivas geométricas. Por exemplo, cores de preenchimento e contorno de polígonos, espessuras de contornos e linhas, cores de pontos, símbolos de pontos, tipos e transparência de preenchimento de polígonos, estilos de linhas, estilos de pontos, etc.

Legenda – uma legenda caracteriza um grupo de dados, dentro de um tema, apresentados com o mesmo visual, quando os dados do tema são agrupados de alguma forma.

As entidades que formam o modelo conceitual estão representadas tanto nas classes que compõe a biblioteca quanto em um conjunto de tabelas no banco de dados. A seção seguinte mostra como as entidades do modelo conceitual são representadas nas classes da TerraLib.

12.3 Classes do modelo conceitual

O conceito de um banco de dados TerraLib é independente do SGDB onde será fisicamente armazenado e é implementado em uma classe abstrata chamada `TeDatabase`. Essa classe abstrata contém os métodos necessários para criar, popular e consultar um banco de dados. A classe `TeDatabase` é derivada em classes concretas, chamadas *drivers*, que resolvem para diferentes SGDB's comerciais e de domínio público, as particularidades de cada um de forma que as aplicações possam criar bancos de dados em diferentes gerenciadores. A TerraLib fornece alguns *drivers* em sua distribuição padrão, como mostrado na Figura 12.3. *Drivers* para outros gerenciadores podem ser criados através da implementação dos métodos virtuais definidos na classe.

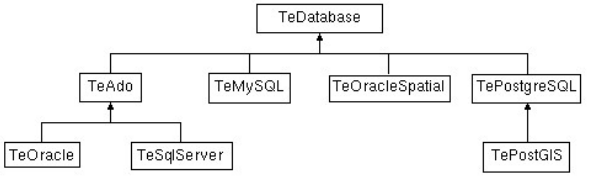


Figura 12.3 – *Drivers* para bancos de dados fornecidos pela TerraLib.

Tipicamente, as aplicações que usam a TerraLib processam ponteiros para a classe `TeDatabase`, inicializados com instancias concretas de algum *driver* como mostrado no Exemplo 12.1.

```
TeDatabase* myDb = 0;
if (op == “ado”)
    myDb = new TeAdo(); // Usa ACCESS através da biblioteca ADO
else
    myDb = new TeMySQL();   // Usa MySQL
```

Exemplo 12.1 – A classe `TeDatabase`.

Um ***Layer*** existe em memória como uma instância da classe `TeLayer`. Um exemplo de criação de um *layer* através da importação de um arquivo de dados em formato MID/MIF é mostrado no Exemplo 12.2. O arquivo contém os dados de um cadastro de municípios de um estado e a rotina de importação vai criar um *layer* no banco de dados com as geometrias e atributos dos municípios descritos no arquivo.

```
TeLayer* lmun =
TeImportMIF("../data/Municipios.mid",myDb,"Municipios");
```

Exemplo 12.2 – Importação de um arquivo de dados.

Uma **Representação** existe em memória como uma instância da estrutura `TeRepresentation`. Cada *layer* possui a informação de quais as representações geométricas (ou geometrias) possui. Assim, após a execução do código acima podemos recuperar algumas informações sobre a representação geométrica dos municípios como o seu menor retângulo envolvente (ver Exemplo 12.3).

```
TeRepresentation* munPol = lmun->getRepresentation(TePOLYGONS);
// recupera o menor retângulo envolvente das geometrias do tipo
// polígono do layer de municípios
TeBox = mumPol->box_;
```

Exemplo 12.3 – A classe `TeRepresentation`.

Uma **Projeção** existe em memória como uma instância da classe `TeProjection`. A TerraLib também fornece uma classe para descrever um datum planimétrico chamada de `TeDatum`. A `TeProjection` é uma classe abstrata que define métodos para converter coordenadas de uma projeção para coordenadas geográficas (latitude e longitude) e vice-versa.

A TerraLib fornece em sua distribuição um conjunto de especializações da classe `TeProjection` representando as projeções mais comuns como UTM, Mercator ou Policônica. O Exemplo 12.4 mostra como converter coordenadas entre uma projeção UTM, Datum SAD69 e uma projeção Policônica, Datum WGS84.

```
TeDatum dSAD69 = TeDatumFactory::make("SAD69");
TeDatum dWGS84 = TeDatumFactory::make("WGS84");
TeUtm* pUTM = new TeUtm(dSAD69,-45.0*TeCDR);
TePolyconic* pPolyconic = new TePolyconic(dWGS84,-45.0*TeCDR);

TeCoord2D pt1(340033.47, 7391306.21);   // em projeção UTM
// Conversão de UTM para policônica
pUTM->setDestinationProjection(pPolyconic);
// Converte uma coordenada de UTM para coordenada geográfica
TeCoord2D ll = pUTM->PC2LL(pt1);
// Converte de geográfica para policônica
TeCoord2D pt2 = pPolyconic->LL2PC(ll);
// Converte uma coordenada de policônica para coordenada UTM
pPolyconic->setDestinationProjection(pUTM);
ll = pPolyconic->PC2LL(pt2);
pt1 = pUTM->LL2PC(ll);
```

Exemplo 12.4 – Conversão de coordenadas para diferentes projeções.

A classe `TeLayer` possui a indicação de qual é a sua projeção cartográfica como mostrado no Exemplo 12.5.

```
TeProjection* munProj = lmun->projection();
```

Exemplo 12.5 – Recuperação da projeção de um *layer*.

Temas existem em memória como instâncias da classe `TeTheme` e **Vistas** como instâncias da classe `TeView`. O Exemplo 12.6 mostra como criar uma nova vista e um novo tema a partir do *layer* criado no Exemplo 12.2.

```
// Cria uma vista com a mesma projeção do layer
TeView* view = new TeView("Municipios ", user);
view->projection(munProj);
// salva a vista no banco de dados
myDb->insertView(view))
// cria um tema com todos os objetos do layer
TeTheme* theme = new TeTheme("t_municipios ", lmun);
view->add(theme);
```

Exemplo 12.6 – Criação de um Tema.

Uma ilustração que resume o relacionamento entre as principais classes que formam o modelo conceitual da TerraLib é mostrado na Figura 12.4. Um banco TerraLib é formado por um conjunto de *layers*, cada *layer* possui uma projeção cartográfica. Em um banco podem ser criadas *n* vistas e cada vista possui sua própria projeção cartográfica. Cada vista pode conter *n* temas sendo que cada tema é criado a partir de um *layer*.

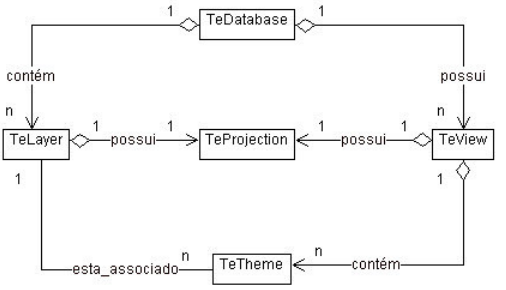


Figura 12.4 – Relacionamento entre as classes do modelo conceitual de TerraLib.

12.4 Modelo do banco de dados

Fisicamente, um banco de dados TerraLib é formado por um conjunto de tabelas em um SGBD-OR, onde são armazenados tanto os dados geográficos (suas geometrias e seus atributos) quanto um conjunto de informações sobre a organização desses dados no banco, ou seja, o modelo conceitual da biblioteca. Essas tabelas podem ser divididas em dois tipos:

1. *Tabelas de Metadados*: possuem nome e formato pré-definido e são usadas para guardar o modelo conceitual da TerraLib;
2. *Tabelas de Dados*: são usadas para guardar os dados em si, tanto em sua componente espacial quanto descritiva.

As tabelas de metadados são automaticamente criadas quando se cria um novo banco TerraLib. Para acessar bancos de dados já existentes, as aplicações abrem conexões a eles, uma aplicação pode manter conexões a mais que um banco de dados ao mesmo tempo. Ao criar ou abrir uma conexão a um banco de dados as aplicações devem informar os parâmetros exigidos pelo SGDBOR onde está armazenado o banco, como mostra o Exemplo 12.7. Ao final da execução da aplicação todas as conexões devem ser fechadas.

```
// Criando um novo banco TerraLib
TeDatabase* db = new TeMySQL();
db->newDatabase(dbname, user, password, host);
```

```
// Acessando um banco de dados já criado
db->connect(host,user,pass,dbname,0);
//... processamento

// Fecha a conexão
db->close();
```

Exemplo 12.7 – Criação e conexão a um banco TerraLib.

As tabelas de metadados servem para armazenar os conceitos descritos na Seção 12.2. Cada *layer* criado no banco gera um registro na tabela chamada `TE_LAYER`, o campo `layer_id` contém a identificação única de cada *layer* no banco de dados. Os outros campos dessa tabela armazenam o nome e o mínimo retângulo envolvente do *layer*, ou seja, o mínimo retângulo envolvente de todas as geometrias associadas ao *layer*.

Cada representação geométrica associada a um *layer* gera um registro na tabela `TE_REPRESENTATION`. Cada tabela de atributos associada a um *layer* gera um registro na tabela `TE_LAYER_TABLE`.

Cada vista criada no banco de dados gera um registro em uma tabela chamada de `TE_VIEW`. Cada instância de projeção cartográfica é armazenada no banco na tabela `TE_PROJECTION`. *Layers* e vistas possuem referência a um registro da tabela de projeções. Cada tema criado gera um registro na tabela `TE_THEME`. Cada tema possui uma referência para vista no qual está definido.

Para compreender melhor as tabelas do modelo de metadados e os relacionamentos entre elas é interessante observar o conteúdo dessas tabelas após a execução de uma seqüência típica de operações:

- Criar um banco de dados;
- Importar um dado geográfico de um arquivo para um *layer* do banco de dados;
- Criar uma *vista*;
- Criar um *tema* usando o *layer* criado e inseri-lo na *vista*.

A Figura 12.5 mostra as tabelas do modelo conceitual e seus relacionamentos após a execução da seqüência de operações. Por questões de simplicidade apenas alguns campos das tabelas são mostrados.As tabelas de dados serão descritas mais adiante após falarmos sobre o modelo de geometrias e de atributos de TerraLib.

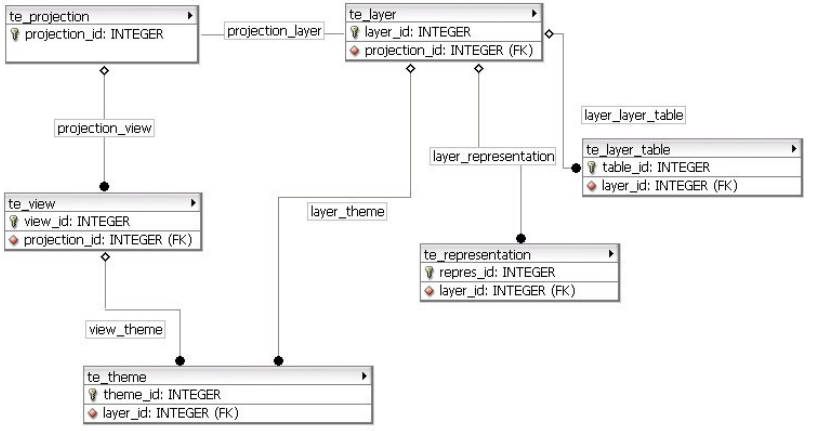


Figura 12.5 – Principais tabelas do modelo conceitual da TerraLib.

12.5 Modelo de geometrias

Conforme descrito na Seção 12.2, na TerraLib os dados geográficos são agregados em *layers*. *Layers* são formados por conjuntos de objetos, onde cada objeto possui uma identificação única, um conjunto de atributos descritivos e uma representação geométrica. Essa seção descreve o modelo de classes de geometria da TerraLib.

A classe base da qual derivam todas as geometrias de TerraLib é chamada de `TeGeometry`. Cada geometria possui uma identificação única, a referência ao seu menor retângulo envolvente e à sua projeção e a identificação do objeto geográfico que representa. As geometrias vetoriais de TerraLib são construídas a partir de coordenadas bi-dimensionais representadas na classe chamada de `TeCoord2D`. Essas geometrias são:

- *Pontos*: representados na classe `TePoint`, implementada como uma instância única de uma `TeCoord2D`;
- *Linhas*: composta de um ou mais segmentos são representadas na classe `TeLine2D`, implementada como um vetor de duas ou mais `TeCoord2D`;
- *Anéis*: representados pela classe `TeLinearRing`, são linhas fechadas, ou seja, a última coordenada é igual a primeira. A classe `TeLinearRing` é implementada como uma instância única de uma `TeLine2D` que satisfaz a restrição de que a primeira coordenada seja igual a última;
- *Polígonos*: representados pela classe `TePolygon`, são delimitações de áreas que podem conter nenhum, um ou mais buracos, ou filhos. São implementados como um vetor de `TeLinearRing`. O primeiro anel do vetor é sempre o anel externo enquanto que os outros anéis, se existirem, são buracos ou filhos do anel externo.

A fim de otimizar a manutenção das geometrias em memória as classes de geometrias de TerraLib são implementadas segundo o padrão de projeto *handle/body* onde a implementação é separada da interface (Gamma et al., 2000). Além disso, as implementações são referências contadas, ou seja, cada instância de uma classe de geometria guarda o número de referências feitas a ela, inicializado com zero quando a instância é criada. Cada vez que uma cópia dessa instância é solicitada apenas o seu número de referências é incrementado e, cada vez que uma instância é destruída, o número de referências a ela é decrementada. A instância é efetivamente destruída apenas quando esse número chega ao valor zero.

Outro aspecto importante das classes de geometria da TerraLib é que elas são derivadas ou da classe `TeGeomSingle` ou da classe `TeGeomComposite`, representando, respectivamente, que são geometrias com um único elemento menor (como um `TePoint`) ou que podem ser compostas de outros elementos menores (como é o caso de `TePolygon`, `TeLine2D` e `TeLinearRing`). Esse padrão de compostos de elementos menores (Gamma et al. 2000) é aplicado também em classes que formam conjuntos de pontos, linhas, polígonos e são representados nas classes

`TePointSet`, `TeLineSet`, e `TePolygonSet`. Os padrões de projeto são implementados com o recurso de classes parametrizadas como mostrado na Figura 12.6, o que torna o código mais reutilizável.

As geometrias matriciais são representadas na classe `TeRaster`, que será descrita por completo no Capítulo 13.

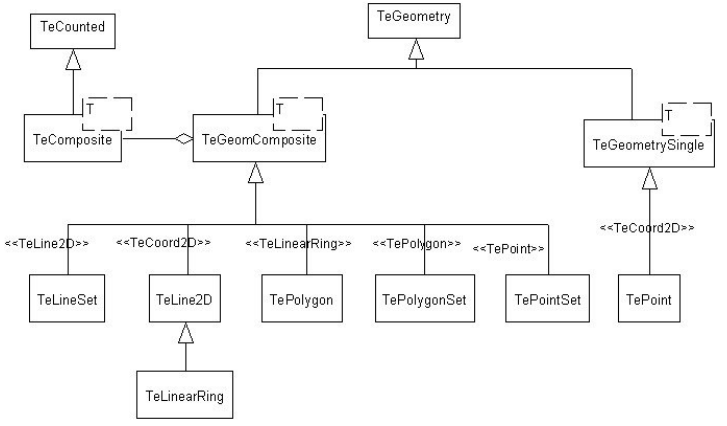


Figura 12.6 – Diagrama das principais classes de geometria da TerraLib (adaptado de Queiroz, 2003)

Um exemplo de criação de geometrias em memória é mostrado no Exemplo 12.8.

```
// Cria um conjunto de linhas
TeLine2D reta;
reta.add(TeCoord2D(500,500));
reta.add(TeCoord2D(600,500));
reta.add(TeCoord2D(700,500));
reta.objectId("reta");

TeLine2D ele;
ele.add(TeCoord2D(700,700));
ele.add(TeCoord2D(800,600));
ele.add(TeCoord2D(900,600));
```

```
ele.objectId("ele");

TeLineSet ls;
ls.add(reta);
ls.add(ele);

// Cria um conjunto de polígonos
// Um polígono simples
TeLine2D line;
line.add(TeCoord2D(900,900));
line.add(TeCoord2D(900,1000));
line.add(TeCoord2D(1000,1000));
line.add(TeCoord2D(1000,900));
line.add(TeCoord2D(900,900));

TeLinearRing r1(line);
TePolygon poly1;
poly1.add(r1);
poly1.objectId("spoli");

// Um polígono com um filho
TeLine2D line2;
line2.add(TeCoord2D(200,200));
line2.add(TeCoord2D(200,400));
line2.add(TeCoord2D(400,400));
line2.add(TeCoord2D(400,200));
line2.add(TeCoord2D(200,200));
TeLinearRing r2(line2);

TeLine2D line3;
line3.add(TeCoord2D(250,250));
line3.add(TeCoord2D(250,300));
line3.add(TeCoord2D(300,300));
line3.add(TeCoord2D(300,250));
```

```
line3.add(TeCoord2D(250,250));
TeLinearRing r3(line3);
TePolygon poly2;
poly2.add(r2);
poly2.add(r3);
poly2.objectId("cpoli");
TePolygonSet ps;
ps.add(poly1);
ps.add(poly2);

// Cria um conjunto de pontos
TePoint p1(40,40);
p1.objectId("ponto1");
TePointSet pos;
pos.add(p1);
```

Exemplo 12.9 – Criação de geometrias vetoriais em memória.

A TerraLib implementa uma estrutura de dados de espaços celulares, que juntamente com o suporte para predicados temporais atende às necessidades de implementação de modelos dinâmicos baseados em espaços celulares. Espaços celulares podem ser vistos ou como uma estrutura matricial generalizada onde cada célula armazena mais que um valor de atributo ou como um conjunto de polígonos que não se interceptam. Essa estrutura traz como uma vantagem a possibilidade de armazenar conjuntamente, numa única estrutura, todo o conjunto de informações necessárias para descrever um fenômeno espacial complexo, o que beneficia aspectos de visualização e interface. Todas as informações podem ser apresentadas da mesma forma que objetos geográficos com representação vetorial. Para atender a essa necessidade, a TerraLib propõe mais uma geometria chamada `TeCell`, que representa uma célula em um espaço celular materializado na classe `TeCellSet`.

O Exemplo 12.10 mostra como criar um espaço celular a partir de um *layer* armazenado em um banco.

```
// Recupera o layer de municípios
TeLayer* municipios = new TeLayer("Municipoios");
db_->loadLayer(municipios);
// Cria um espaço celular sobre a extensão do layer
// de municípios, onde cada célula tem 100 x 100 metros
TeLayer* espaco_cel = TeCreateCells("CellsMunic", municipios,
100, 100);
```

Exemplo 12.10 – Criação de um espaço celular.

Cada célula possui uma identificação única e uma referência a sua posição dentro do espaço celular, a qual podem ser associados diferentes atributos conforme o modelo dinâmico sendo construído.

12.6 Modelo de armazenamento de geometrias

A proposta da TerraLib, conforme mostramos na Figura 12.2, é trabalhar em bancos de dados geográficos que podem armazenar tanto atributos descritivos como atributos geométricos (como pontos, linhas, polígonos e dados matriciais). Esses bancos de dados podem ser construídos em SGDB's que possuem extensões espaciais, ou seja, possuem a capacidade de criar tipos espaciais e manipulá-los como tipos básicos e fornecem mecanismos eficientes de indexação e consulta (Shekkar e Chawla, 2002). Podem também ser construídos em SGDB's que oferecerem somente a capacidade de criar tabelas com campos do tipo binário longo. Na TerraLib esses dois tipos de SGDB's são usados de maneira transparente através da classe abstrata `TeDatabase`.

O modelo de armazenamento de geometrias em um banco leva em conta questões relativas à eficiência no seu armazenamento e na sua recuperação, e também a existência ou não de um tipo espacial no SGDB. Todas as tabelas que armazenam as geometrias possuem os campos:

- `geom_id`: do tipo inteiro, que armazena a identificação única da geometria;
- `object_id`: do tipo texto, que armazena a identificação única do objeto geográfico ao qual a geometria está relacionada;

- `spatial_data`: armazena o dado geométrico. O tipo desse campo depende do SGDB onde está armazenado o banco. Para SGDB's com extensão espacial é o tipo fornecido pela extensão. Para SGDB's sem a extensão espacial é um binário longo.

Para os SGDB´s sem extensão espacial as tabelas de geometrias do tipo linhas e polígonos possuem outros campos para armazenar o mínimo retângulo envolvente da geometria (`lower_x`, `lower_y`, `upper_x`, `upper_y` todos do tipo real). Esses campos são indexados pelos mecanismos fornecidos pelo SGBD e serão usados pelas rotinas de recuperação como os indexadores espaciais do dado. Para os SGDB's com extensão, a coluna com o dado espacial é indexada espacialmente pelo mecanismo oferecido pela extensão.

A Figura 12.7 mostra a diferença entre uma tabela de geometria do tipo polígono criada em um banco sem extensão espacial e em um banco com extensão espacial. No segundo caso o tipo "GEOMETRY" representa o tipo espacial fornecido pela extensão. No caso das geometrias do tipo polígono, o modelo de armazenamento em campos longos, prevê que cada anel do polígono é armazenado em um registro da tabela. O anel externo contém a informação sobre o número de filhos que o polígono possui e os anéis internos guardam a identificação de seu pai, ou seja, do anel externo no qual estão contidos. Essa forma de armazenamento permite a recuperação parcial de polígonos com um grande número de filhos (por exemplo, uma operação de *zoom* em um mapa em uma área grande como a Amazônia legal, em uma escala pequena). Como são armazenados os retângulos envolventes de cada filho é possível recuperar somente o pai e os filhos que estão dentro do retângulo envolvente definido pelo *zoom*. Isso representa uma otimização no processamento desse dado.

Tabela de polígonos em SGDB´s sem extensão espacial

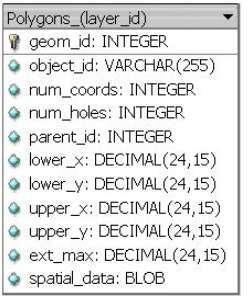


Tabela de polígonos no Oracle Spatial

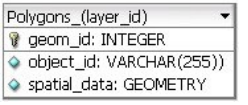


Figura 12.7 – Modelo de armazenamento de geometrias do tipo polígonos (adaptado de Ferreira, 2003).

A Figura 12.8 mostra como são as tabelas que armazenam geometrias do tipo linhas e a Figura 12.9 as tableas para geometrias do tipo pontos.

Tabela de linhas em SGDB´s sem extensão espacial

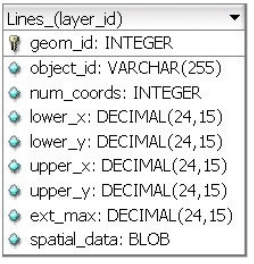


Tabela de linhas no Oracle Spatial

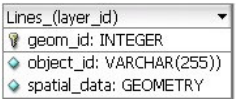


Figura 12.8 – Modelo de armazenamento de geometrias do tipo linhas.

Tabela de pontos em SGDB´s sem extensão espacial

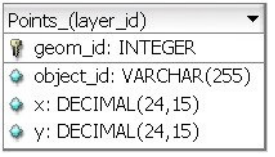


Tabela de pontos no Oracle Spatial

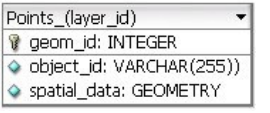


Figura 12.9 – Modelo de armazenamento de geometrias do tipo pontos.

12.7 Atributos descritivos

Os atributos descritivos dos objetos de um *layer* são representados em tabelas relacionais onde cada campo representa um atributo do objeto. Um dos campos deve conter a identificação do objeto e seus valores são repetidos nas tabelas de geometria permitindo assim a ligação entre os atributos descritivos e a geometria do objeto.

Cada *layer* pode ter uma ou mais tabelas de atributos e ao serem inseridos no banco de dados cada tabela de atributo é registrada na tabela de metadados chamada `te_layer_table`. Nessa tabela é registrada também o nome do campo que é a chave primária e identificador do objeto. Esse campo serve de ligação entre atributos e geometrias. Ao serem criados, os temas selecionam quais tabelas do *layer* irão usar.

Quando a tabela de atributos não possui nenhuma informação temporal e cada registro representa os atributos de um objeto diferente, a tabela é chamada de **tabela estática**. Essas classificação semântica das tabelas de atributos é uma característica da TerraLib e tem por objetivo definir funções e processamentos dependentes desses tipos. Isso ficará mais claro adiante quando falarmos do processamento de dados espaço-temporais.

Algumas tabelas de atributos, chamadas **tabelas externas**, não representam nenhum objeto definido em um *layer*, mas podem possuir algum campo com valores coincidentes com os valores de um campo de uma tabela estática de um *layer*. Através de uma operação de junção por esses campos coincidentes uma tabela externa pode acrescentar informações aos objetos de um *layer*. Por isso, tabelas externas podem ser incorporadas ao banco e registradas como tal, ficando disponíveis para serem usadas em todos os temas do banco.

Em memória, as tabelas de atributos são instâncias da classe `TeTable`. Essa classe sabe qual o nome, qual o tipo, quais são os campos e pode conter também o seu conteúdo. No Exemplo 12.11 mostramos como criar uma instância da classe `TeTable` em memória e como adicionar alguns registros a essa tabela.

```
// Cria a lista de atributos da tabela
TeAttributeList attList;
TeAttribute at;
at.rep_.type_ = TeSTRING;
at.rep_.numChar_ = 16;
at.rep_.name_ = "IdObjeto";
at.rep_.isPrimaryKey_ = true;
attList.push_back(at);

at.rep_.type_ = TeSTRING;        // um atributo do tipo string
at.rep_.numChar_ = 255;
at.rep_.name_ = "nome";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

at.rep_.type_ = TeREAL;          // um atributo do tipo real
at.rep_.name_ = "val1";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

// Cria uma instância da tabela em memória
TeTable attTable("teste2Attr",attList,"IdObjeto","IdObjeto");
row.push_back("reta");
row.push_back("L reta");
row.push_back("11.1");
attTable.add(row);
row.clear();
row.push_back("ele");
```

```
row.push_back("L ele");
row.push_back("22.2");
attTable.add(row);
row.clear();
row.push_back("spoli");
row.push_back("Pol simples");
row.push_back("33.3");
attTable.add(row);
row.clear();
row.push_back("ponto1");
row.push_back("Um ponto");
row.push_back("55.5");
attTable.add(row);
row.clear();
row.push_back("cpoli");
row.push_back("Pol buraco");
row.push_back("44.4");
attTable.add(row);
row.clear();
```

Exemplo 12.11 – Criação de uma tabela de atributos em memória.

As seções anteriores tiveram por objetivo explicar o modelo conceitual de um banco de dados geográfico TerraLib e o modelo físico de persistência desse modelo conceitual e dos dados geográficos. As seções seguintes tratarão do acesso ao banco em diferentes níveis de abstração: no nível do *layer* (`TeLayer`), no nível da banco de dados (`TeDatabase`) e no nível de objetos geográfico (`TeQuerier`).

12.8 Acesso ao banco de dados através do *layer*

As rotinas de importação que inserem um arquivo de dados geográfico são escritas usando os métodos da classe `TeLayer` (ver Exemplo 12.2). Essa classe fornece métodos para a inserção de geometrias e atributos. Através desses métodos os registros nas tabelas do modelo conceitual são

preenchidos corretamente, fazendo as referências ao identificador do *layer* quando necessário.

O Exemplo 12.12 deve ser lido como uma seqüência do Exemplo 12.10 e do Exemplo 12.11 e mostra como criar um *layer* e como usar seus métodos para inserir no banco de dados as geometrias e atributos criados em memória.

```
// Cria a definição da projeção do layer
TeDatum mDatum = TeDatumFactory::make("SAD69");
TeProjection* pUTM = new TeUtm(mDatum,0.0);
// Cria um novo layer chamado "teste"
// myDb_ é um ponteiro para um banco de dados já conectado
TeLayer *layer1 = new TeLayer("teste",myDb_,pUTM);
layer1->addGeometries(TeLINES,"tabelaLinhas");

// Insere as linhas criadas em memória
layer1->addLines(ls);
// Insere os polígonos criados em memória
layer1->addPolygons(ps);
// Insere os pontos criados em memória
layer1->addPoints(pos);
// cria a tabela de atributos criada em memória
layer1->createAttributeTable(attTable);
// salva a tabela de atributos no aco de dados
layer1->saveAttributeTable(attTable))
```

Exemplo 12.12 – Inserção de geometrias e atributos no banco através da classe `TeLayer`.

O Exemplo 12.12 mostra como controlar a inserção de cada tipo de geometria (pontos, linhas ou polígonos) podendo inclusive especificar o nome para as tabelas de geometrias (como no caso da inserção de linhas).

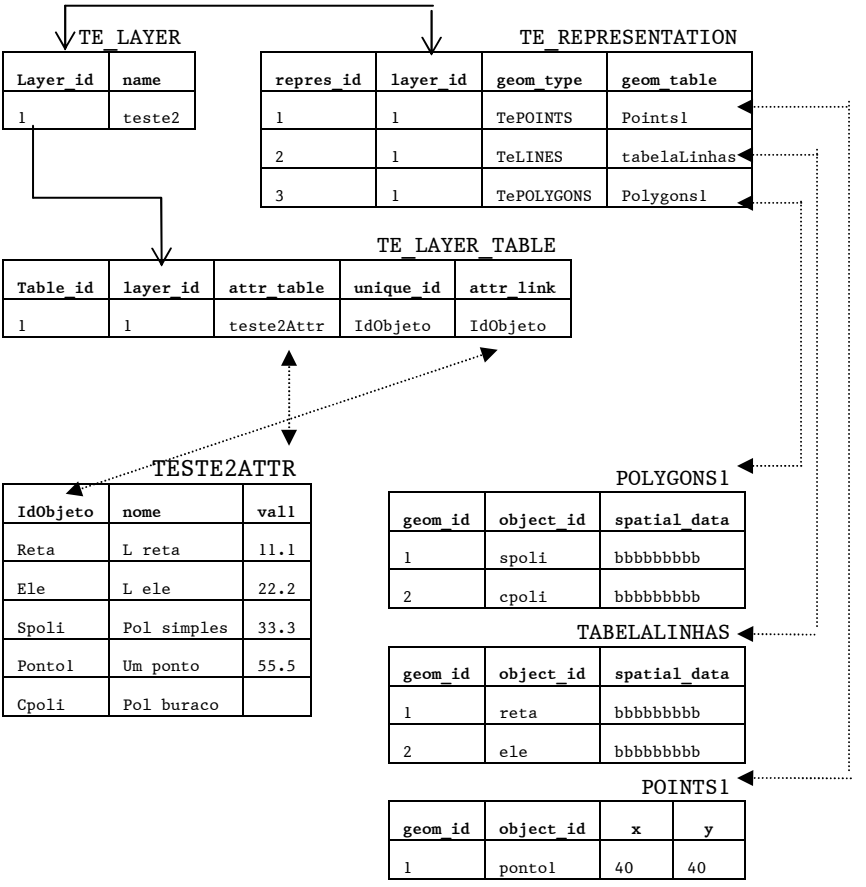


Figura 12.10 – Conteúdo do banco após a execução do Exemplo 10.

É interessante verificar o conteúdo do banco de dados após a execução do Exemplo 12.12. Devem ser notados o conteúdo das tabelas de metadados e as referências feitas aos nomes das tabelas de geometrias, aos nomes das tabelas de atributos, ao identificador do *layer*, e o registro dos nomes de colunas usados para relacionar geometrias e atributos. Na Figura 12.10 são representados os conteúdos essas tabelas e seus relacionamentos. Para efeito de clareza nem todos os campos das tabelas são representados. As linhas contínuas representam relacionamentos físicos entre os campos das tabelas e as linhas pontilhadas representam

relacionamentos em termos de conteúdo. Por exemplo, o nome da tabela de atributos e o nome do campo que faz o seu relacionamento com as tabelas de geometrias são registrados na tabela `TE_LAYER_TABLE`.

A recuperação dos dados armazenados no banco, também pode ser feita diretamente através da classe `TeLayer`. Essa classe fornece alguns métodos para recuperar as geometrias de um determinado tipo como mostrado no Exemplo 12.13.

```
TePolygonSet ps2;
layer1->getPolygons(ps2);
cout << "Numero de objetos com geometria do tipo poligono: "
<< ps2.size();
ps2.clear();
// recupera somente os polígonos associados ao objeto com o
// identificador “spoli”
layer1->loadGeometrySet("spoli");
```

Exemplo 12.13 – Recuperação de geometrias através do *layer*.

Os métodos para recuperação de geometrias e atributos através da classe `TeLayer` são poucos, mas a classe `TeDatabase` é uma interface completa de acesso ao banco de dados, essa interface será descrita a seguir.

12.9 A interface TeDatabase

A classe `TeDatabase` é uma interface completa de manipulação do banco de dados através da qual é possível inserir, alterar e recuperar as entidades do modelo conceitual e os dados geográficos. Ela contém todos os métodos para criar as tabelas de metadados, as tabelas de geometrias bem como uma tabela de atributos como ilustrado no Exemplo 12.14. Esse exemplo mostra também como recuperar um *layer* do banco e posteriormente como removê-lo. A remoção do *layer* implica que todos os seus dados e metadados são fisicamente removidos do banco bem como todas as outras entidades do modelo conceitual que fazem referência a ele. Por exemplo, ao remover um *layer* do banco todos os temas criados sobre esse *layer* devem ser removidos também.

```
db->createConceptualModel(); // cria todo o modelo conceitual
// cria uma tabela que armazena geometrias do tipo polígono
db->createPolygonGeometry(“TabelaPoligonos”);

// cria uma tabela de atributos descritivos
TeAttributeList meusAttributos;
db->createTable(“TabelaAtributos”, meusAttributos);

// cria uma nova coluna em uma tabela já existente
TeAttributeRep novaColuna;
novaColuna.type_ = TeSTRING;
novaColuna.name_ = "novaCol";
novaColuna.numChar_ = 10;
db->addColumn(“TabelaAtributos”, novaColuna);
// recupera um layer do banco
TeLayer *layer = new TeLayer("Distritos");
// carrega todas as informações sobre o layer chamado Distritos
db->loadLayer(layer);
// remove o layer
db->deleteLayer(layer->id());
```

Exemplo 12.14 – Métodos de criação e modificação de tabelas da classe `TeDatabase`.

A classe `TeDatabase` também fornece métodos para a inserção de geometrias e atributos nas tabelas do banco de dados como mostra o Exemplo 12.15.

```
TePolygonSet polyset;
db->insertPolygonSet(“tabPoligono”, polyset);

TeLineSet lineset;
db->insertLineSet   (“tabLine”, lineset);
```

```
TePointSet pointset;
db->insertPointSet("tabPoint", pointset);

TeTable tabAttributes;
insertTable(tabAttributes);
```

Exemplo 12.15 – Inserções no banco através da classe `TeDatabase`.

A classe `TeDatabase` também é capaz de submeter ao banco comandos escritos em SQL – Structured Query Language como mostrado no Exemplo 12.16. Apesar da SQL ser considerada padrão para SGDB's relacionais, podem existir algumas variações em termos da capacidade de execução de um comando SQL. O método `TeDatabase::execute` retorna verdadeiro ou falso caso o comando foi executado com sucesso ou não, respectivamente. A classe `TeDatabase` armazena o erro retornado pelo SGDB resultante do último comando executado sobre o banco.

```
string sql = "UPDATE TabelaAtributos SET novaCol = 'xxx' ";
if (db->execute(sql))
  cout << "Comando executado com sucesso!";
else
  cout << "Erro: " << db->errorMessage();
```

Exemplo 12.16 – Execução de um comando SQL.

Note que o método `TeDatabase::execute` não deve ser usado para comandos que representam consultas ao banco, ou seja, que retornam valores ou registros, mas somente para comandos que alteram as tabelas e registros do banco.

12.10 A classe TeDatabasePortal

A classe `TeDatabase` fornece um mecanismo mais flexível de consulta ao banco de dados do que apenas através dos métodos da classe `TeDatabase`. Esse mecanismo é implementado pela classe `TeDatabasePortal`. Os

portais de consultas são criados a partir de qualquer instância da classe `TeDatabase` que possua uma conexão aberta. Os portais podem submeter consultas SQL ao banco e disponibilizar os registros resultantes da consulta para a aplicação processar. Um banco pode criar quantos portais forem necessários, porém após serem consumidos os resultados todo portal deve ser destruído. O Exemplo 12.17 mostra o uso típico de um portal.

```
// Abre um portal no banco de dados
TeDatabasePortal* portal = db->getPortal();
if (!portal)
  return;

// Submete uma consulta sql a uma tabela
string sql = "SELEC * FROM TabelaAtributos";
if (!portal->query(sql))
{
  cout << "Não foi possível executar..." << db->errorMessage();
   delete portal;
   return;
}
// Consome todos os registros resultantes
while (portal->fetchRow())
{
  cout << "Atributo 1: " << portal->getData(0) << endl;
  cout << "Atributo 2: " << portal->getData("nome") << endl;
  cout << "Atributo 3: " << portal->getDouble(2) << endl;
}
// Libera os portal para fazer uma nova consulta
portal->freeResult();

// Submete uma nova consulta
sql = "SELECT SUM(val1) FROM TabelaAtributos WHERE val1 >
33.5";
if (portal->query(sql) && portal->fetchRow())
```

```
  cout << "Soma: " << portal->getDouble(0);
delete portal;
```

Exemplo 12.17 – Uso do `TeDatabasePortal`.

Analisando o exemplo vemos que após a chamada do método `TeDatabasePortal::query` os registros ficam disponíveis para serem consumidos em seqüência. Um ponteiro interno é posicionado em uma posição *anterior ao primeiro registro*. A cada chamada ao método `TeDatabasePortal::fetchRow` o ponteiro interno é incrementado e o valor verdadeiro é retornado quando o ponteiro está em um registro válido ou falso caso contrário. Dessa forma é possível fazer o *loop* em todos os registros retornados.

Os campos de um registro do portal podem ser acessados de duas formas: pela ordem do campo na seleção expressa na SQL (iniciando em zero), ou diretamente pelo nome do campo. O valor do campo pode ser obtido como uma *string* de caracteres através do método `TeDatabasePortal::getData` independentemente do tipo do campo ou através dos métodos que especificam o tipo de retorno (`getDouble`, `getInt`, `getDate` ou `getBlob`). Na segunda forma é necessário que a aplicação conheça o tipo do campo sendo acessado.

A classe `TeDatabasePortal` também pode executar consultas sobre uma tabela de geometrias e acessar os campos resultantes como os tipos geométricos da TerraLib, como mostrado no Exemplo 12.18.

```
// Submete uma consulta sobre uma tabela de geometrias
string q =" SELECT * FROM Poligons11 WHERE object_id='cpoli';
       q += " ORDER BY parent_id, num_holes DESC, ext_max ASC";

if (!portal->query(q) || !portal->fetchRow())
{
  delete portal;
  return false;
}
```

```
bool flag = true;
do
{
  TePolygon poly;
  flag = portal->fetchGeometry(poly);
  // usa o polígono retornado
} while (flag);
delete portal;
```

Exemplo 12.18 – Uso de um portal para recuperar geometrias.

Essa seção mostrou como utilizar a interface `TeDatabase` para construir e consultar um banco de dados. As consultas expressas até agora foram sempre baseadas em critérios a serem atendidos sobre atributos descritivos porém, parte importante de um banco de dados geográfico é a consulta por critérios espaciais. A seção seguinte mostra como as operações espaciais são tratadas na TerraLib.

12.11 Operações espaciais

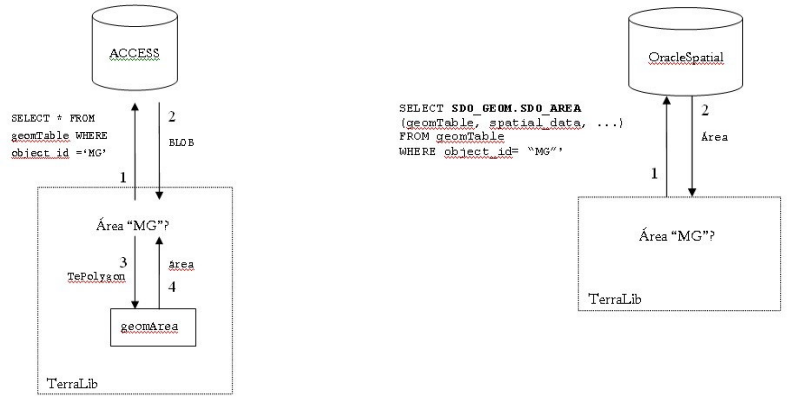
A TerraLib oferece um conjunto de operações espaciais sobre geometrias vetoriais e matriciais (uma revisão da literatura e descrição mais detalhada sobre operações espaciais pode ser encontrada em Ferreira, 2003). As operações espaciais implementadas na TerraLib podem ser divididas em:

- Determinação de relacionamentos topológicos entre geometrias vetoriais: são baseados na matriz de 9-intersecções dimensionalmente estendidas onde são formalizadas relações como toca, intercepta, cobre, disjunto ou é-coberto-por, como mostrado no Capítulo5.
- Funções métricas: como cálculo de áreas e comprimentos e buffers de distância;
- Geração de novas geometrias: como a geração de centróides e geometrias convexas;
- Operações que combinam geometrias: como diferença, união e intersecção e diferença simétrica;

- Operações zonais e focais sobre geometrias matriciais: como a obtenção de medidas estatísticas sobre uma região de um dado matricial e operações de recorde de uma região do dado matricial.

As operações espaciais estão implementados em um conjuntos de métodos da classe `TeDatabase` que funcionam como uma Interface Genérica de Operações Espaciais. Essa interface é genérica porque, através da classe `TeDatabase`, esses métodos podem ser chamados da mesma maneira em *drivers* para SGDB's com extensão espacial ou sem extensão espacial. *Drivers* para SGDB's com extensão espacial implementam a interface de operações especiais usando as funções da extensão espacial. *Drivers* que não possuem a extensão implementam a interface através de funções espaciais básicas escritas sobre as classes de geometria, após a sua recuperação do banco de dados. Na Figura 12.12 mostramos a diferença de execução de uma operação espacial, cálculo de área de um objeto, executada em um *driver* sem extensão espacial (p. ex. ACCESS) e um *driver* com extensão espacial (p. ex. Oracle Spatial).

As operações sobre dados matriciais foram implementadas somente usando as classes da TerraLib, uma vez que as extensões espaciais ainda não fornecem as operação sobre dados matriciais. Uma descrição completa da interface genérica de operações espaciais em um banco de dados geográfico pode ser encontrada em Ferreira (2003).



(a) Banco sem extensão espacial (b) Banco com extensão espacial

Figura 12.11 – Representação da operação de cálculo de área (adaptado de

Ferreira, 2003).

O Exemplo 12.19 mostra a execução de operações espaciais unárias sobre uma tabela de geometrias do tipo polígono, que representam objetos que são municípios do estado de São Paulo, identificados pelo nome do município. Sobre um objeto escolhido (o município de São Paulo) são calculados um valor de área e um *buffer* de distância de 1000 metros do objeto. O resultado da operação é um conjunto com dois polígonos onde o primeiro é o *buffer* externo e o segundo é o *buffer* interno.

```
string polyTable = "Poligons11";  // tabela de polígonos
// Consulta 1: calcular a área do município de São Paulo
// Chama uma função unária sobre a tabela de polígonos
vector<string> spId;              // identificadores dos objetos
spId.push_back("São Paulo");
double area;
db->calculateArea(polyTable, TePOLYGONS, spId, area);
// Consulta 2: calcular um buffer de distância de 1000 metros
do município de São Paulo
// Chama uma função unária sobre a tabela de polígonos
TePolygonSet& bufferSet;
db->Buffer(polyTable, TePOLYGONS, spId, bufferSet, 1000.0);
```

Exemplo 12.19 – Operações espaciais unárias.

O Exemplo 12.20 mostra a execução de operações espaciais envolvendo mais uma tabela de geometrias, que contém geometrias do tipo linhas representando estradas estaduais. Essas operações são baseadas em consultas sobre relações topológicas entre geometrias, no primeiro caso entre geometrias de uma mesma tabela e no segundo entre geometrias de duas tabelas diferentes. É interessante notar que o resultado é retornado em um portal de forma que a aplicação possa consumir cada geometria resultante uma a uma.

```
string lineTable = "Lines12";     // tabela de linhas
```

```
TeDatabasePortal result = db->getPortal();  // portal
// Consulta 3: quais os municípios que tocam São Paulo?
// Chama uma função binária sobre a tabela de polígonos
if (db->spatialRelation(polyTable, TePOLYGONS, spId, result,
TeTOUCHES)) {
  bool flag = true;
  do {
       TePolygon poly;
       flag = result->fetchGeometry(poly); // polígono retornado
  } while (flag);
}
result->freeResult();



// Consulta 4: quais as estradas que cruzam São Paulo?
// Função binária entre a tabela de polígonos e de linhas
if (db->spatialRelation(polyTable, TePOLYGONS, spId,
    lineTable, TeLINES, result, TeINTERSECTS)) {
  bool flag = true;
  do {
       TeLine line;
       flag = result->fetchGeometry(line); // linha retornada
  } while (flag);
}
```

Exemplo 12.20 – Operações Espaciais para a consulta de relacionamentos topológicos.

O Exemplo 12.21 mostra as operações Zonal e Recorte de um dado matricial. Na operação zonal, um conjunto de estatísticas (por exemplo soma, valor máximo, valor mínimo, contagem, média, variância, etc.) é calculado sobre a região do dado matricial contida dentro de um polígono que representa a zona de interesse. O resultado é fornecido em uma

estrutura própria que armazena as estatísticas calculadas em cada dimensão do dado matricial.

Na operação de recorte o dado matricial é recortado pela zona de interesse, o resultado é um novo *layer* no banco de dados. A implementação dessa função foi feita usando o conceito de *iterador* sobre uma representação matricial, ou seja, um ponteiro que percorre todos os elementos do dado matricial. No caso da operação de recorte, foi usada uma especialização do conceito de *iterador* sobre dados matriciais, de forma que esse percorra somente os elementos que possuam uma certa relação com a região de interesse. As relações possíveis são que a área do elemento esteja toda dentro da região, que a área do elemento esteja toda fora da região, que o centro do elemento esteja dentro da região ou que o centro do elemento esteja fora da área de interesse.

```
string rasterTable = "RasterLayer12";   // tabela de linhas
TePolygon reg;
TeStatisticsDimensionVect result;

// Consulta 5: Calcular as estatísticas do dado matricial
usando como região de interesse
// o polígono defindo em "reg'
db->Zonal(rasterTable, reg, result);

// Consulta 5: Recortar o dado matricial usando como região de
interesse o polígono
// definido em "reg'
TeStrategicIterator st = TeBoxPixelIn;
db->Mask(rasterTable, reg, "Recorte", st);
```

Exemplo 12.21 – Operações sobre dados matriciais.

As rotinas que implementam as operações espaciais sobre as estruturas geométricas também estão disponíveis para serem usadas diretamente pelas aplicações. Essas rotinas são funções parametrizadas de forma a poder serem chamadas com a mesma assinatura, tomando como

parâmetros as diferentes classes de geometrias vetoriais, como mostrado no Exemplo 12.22. Uma descrição completa da das operações espaciais da TerraLib está em Queiroz (2003).

```
TeLine line1;
TeLine line2;
bool res = TeOverlaps(line1, line2);
TePolygon pol1;
TePolygon pol2;
res = TeOverlaps(pol1, pol2);
```

Exemplo 12.22 – Chamada de operações espaciais sobre geometrias.

12.12 Dados espaço-temporais

A TerraLib tem uma proposta de trabalhar com dados geográficos espaciais e temporais, ou seja, de considerar a componente temporal dos dados geográficos, quando houver. Como exemplos de dados espaço-temporais podemos identificar:

- *Eventos*: ocorrências independentes no espaço e no tempo, que possuem um rótulo temporal de quando o evento ocorreu. Esse é o caso de crimes que ocorrem em uma determinada localização de uma cidade em uma determinada data e hora. Cada objeto espacial associado a um associado a um evento terá uma única identificação no banco de dados;
- *Objetos Mutáveis*: os quais possuem uma geometria fixa ao longo do tempo, porém seus atributos descritivos podem variar. Um exemplo desse tipo de objeto são os dados manipulados por modelos dinâmicos baseados em espaços celulares, ou estações fixas de coletas de dados;
- *Objetos em Movimento*: os quais possuem limites e localizações, ou seja, geometrias que podem variar no tempo assim como seus atributos descritivos. Esse é o caso quando se acompanha a evolução de dos lotes de uma cidade, ou dos polígonos de desmatamento de uma região.

No modelo de banco de dados geográfico proposto pela TerraLib os dados são registrados semanticamente em função da sua natureza temporal com o objetivo de possam ser construídas funcionalidades específicas relativas a essa natureza temporal. Entre essas funcionalidades estão incluídos os algoritmos de agrupamento temporal, visualização de animações ou criação de temas com restrições temporais e ferramentas de modelagem dinâmica. Assim, quando uma tabela de atributos, associada a um *layer*, é inserida em um banco de dados, uma das informações registradas na tabela de metadados `TE_LAYER_TABLE` é qual o seu tipo, e dependendo desse tipo outras informações são necessárias. Além das tabelas estáticas e externas descritas anteriormente (ver Seção 12.7) os outros tipos de tabelas são:

- *Eventos:* uma tabela de atributos de eventos. É preciso registrar qual de seus campos possui a identificação única do evento (e que faz a ligação com sua geometria) e quais de seus campos possuem a informação temporal da ocorrência do evento. A informação temporal normalmente é um intervalo (tempo inicial e tempo final), mas para rótulos temporais pontuais o tempo inicial é igual ao tempo final;
- *Atributos Variáveis:* uma tabela de atributos relacionada a dados do tipo objetos mutáveis. É preciso identificar quais o campo possui a identificação única do objeto no banco (e que faz a ligação com sua geometria), qual campo identifica de maneira única cada versão, ou instância de atributos associada ao objeto, e quais campos possuem a informação temporal que determina o intervalo de validade dessa instância.

O modelo de armazenamento dos objetos em movimento ainda está sendo desenvolvido na TerraLib, uma vez que esse é o caso mais complexo. Além de se considerar a variação no tempo associado aos atributos dos objetos é preciso considerar a variação associada às suas geometrias. Cada geometria tem que armazenar um intervalo (ou instante) de validade e é necessário propor um mecanismo de associação, em um determinado instante ou intervalo, entre a geometria e os atributos do objeto.

Nas classes da TerraLib o modelo de tratamento de dados espaço-temporais está distribuído nas classes `TeSTInstance`, `TeSTElement`, `TeSTElementSet` e `TeQuerier`.

A classe `TeSTInstance` da representa uma instância, ou versão, no tempo de um elemento ou objeto geográfico. Uma instância contém o identificador do elemento ou objeto ao qual pertence, um intervalo de tempo de validade, suas geometrias e seus de atributos. Todas as instâncias de um mesmo elemento ou objeto geográfico são representadas pela classe `TeSTElement`. A classe `TeQuerier` implementa um mecanismo flexível de consulta ao banco de dados e recuperação de objetos e suas instâncias, dinâmicos ou estático no tempo. No segundo tipo cada objeto tem apenas uma única instância.

```
// Recupera o layer de estações de coletas, ou armadilhas
TeLayer* armadilhas = new TeLayer("LAYER_ARMADILHAS");
db_->loadLayer(armadilhas);
// Preenche os parâmetros da consulta
// carregar todos os atributos dos objetos
bool loadAllAttributes = true;
// não carregar as geometrias dos objetos
bool loadGeometries = false;
TeQuerierParams                    querierParams(loadGeometries,
    loadAllAttributes);

// carregar as instâncias dos objetos do layer
querierParams.setParams(armadilhas);

// Cria uma consulta a partir dos parâmetros
// Carrega as instâncias de objetos do banco que atendem aos
critérios da consulta
TeQuerier  querier(querierParams);
querier.loadInstances();

// Retorna a lista dos atributos descritivos carregados e
mostra seus nomes
```

```
unsigned int i;
TeAttributeList attrList = querier.getAttrList();
for (i=0; i<attrList.size(); ++i)
  cout << attrList[i].rep_.name_ << endl;
// Percorre as instâncias selecionadas uma a uma
TeSTInstance sti;
while(querier.fetchInstance(sti)) {
  cout << " Object: " << sti.objectId() << endl << endl;
   // Mostra o nome e valor de cada atributo da instância
  TePropertyVector vec = sti.getPropertyVector();
  for (i=0; i<vec.size(); ++i) {
     string  attrName = vec[i].attr_.rep_.name_;
     string  attrValue = vec[i].value_;
     cout << attrName  << " : " << attrValue << endl;
   }
}
```

Exemplo 12.23 – Aplicação de um `TeQuerier` para recuperar as instâncias dos objetos de um *layer*.

A classe `TeQuerier` contém uma série de parâmetros que definem quais instâncias de quais objetos e quais atributos desses objetos devem ser recuperados. Uma vez definidos os parâmetros do `TeQuerier` e aplicada a seleção, as aplicações podem processar as instâncias retornadas uma a uma. O Exemplo 12.23 mostra como criar um `TeQuerier` para recuperar todas os objetos de um *layer* e processar todas as instâncias desses objetos.

O *layer* utilizado no exemplo agrega objetos armadilhas para a coleta de ovos de um determinado inseto, causador de uma doença. A cada semana as armadilhas são visitadas e o número de ovos na armadilha e outros parâmetros da são observados e inseridos no banco. Esse dado é do tipo geometrias fixas e atributos variáveis no tempo. A armadilha não muda, mas seu atributo quantidade de ovos varia a cada semana.

Um `TeQuerier` também pode carregar todos objetos definidos em um tema, lembrando que um tema pode definir um subconjunto dos objetos de um *layer*. A seleção expressa no `TeQuerier` vai levar em conta as restrições definidas no tema sendo recuperado. O Exemplo 12.24 mostra

uma consulta para recuperar todas as instâncias, atributos e geometrias dos objetos pertencentes às armadilhas do tipo A.

```
// Recupera o tema com somente as armadilhas do tipo A
TeTheme* armadilhasTipoA = new TeTheme("Armadilhas_A");
db_->loadTheme(bairros);
// Preenche os parâmetros da consulta
// carregar todos os atributos dos objetos
bool loadAllAtt = true;
// carregar as geometrias dos objetos
bool loadGeom = true;
// carregar as instâncias dos objetos do tema
TeQuerierParams querierParams(loadGeom, loadAllAttr);
querierParams.setParams(armadilhasTipoA);

// Cria uma consulta e carrega as instâncias
// de objetos do banco que atendem aos critérios da consulta
TeQuerier  querier(querierParams);
querier.loadInstances();
// Percorre as instâncias selecionadas uma a uma
TeSTInstance sti;
while (querier.fetchInstance(sti)) {
  if (sti.hasPoints()) {
     TePointSet ponSet;
     sti.getGeometry(ponSet);
     for (unsigned int i=0; i<ponSet.size(); ++i) {
        cout << " Point : " << ponSet[i].location().x() << ", ";
         cout << ponSet[i].location().y() << endl;
    }
  }
}
```

Exemplo 12.24 – Aplicação de um `TeQuerier` para recuperar as instâncias dos objetos de um tema.

O Exemplo 12.25 mostra como incluir uma restrição espacial nos parâmetros da consulta a ser expressa no `TeQuerier`, mostra também como selecionar apenas alguns atributos dos objetos. O exemplo seleciona somente as instâncias das armadilhas que estão dentro do polígono que delimita certo bairro, e para cada instância de armadilha seleciona somente o atributo 'nro_ovos'.

```
TePolygonSet limiteBairro;
// Preenche os parâmetros da consulta
vector<string> attributes;
attributes.push_back ("ARMADILHAS. NRO_OVOS");
bool loadGeom = true;      // carregar as geometrias dos objetos
TeQuerierParams querierParams(loadGeom, attributes);
querierParams.setParams(armadilhas); // carregar as instâncias
dos objetos do tema
querierParams.setSpatialRest(limiteBairro);
```

Exemplo 12.25 – Definição de uma consulta com restrição espacial e com seleção de somente um atributo.

A classe `TeQuerier` também pode expressar agrupamentos de valores de atributos de as instâncias por objeto, por restrição espacial ou ainda por unidades (chronons) de tempo. O Exemplo 12.16 mostra como agrupar por armadilha todas as instâncias do seu atributo 'nro_ovos' através de uma operação de soma.

```
// Cria uma definição de agrupamento do atributo 'nro_ovos'
// através da operação de soma
TeGroupingAttr attributes;
pair<TeAttributeRep,TeStatisticType>
     attrl(TeAttributeRep("ARMADILHAS.NRO_OVOS"), TeSUM);
attributes.insert(attrl);
// Opção 1: Consulta de instâncias agrupadas por objeto
```

```
// Resultado vai ser uma única instância por objeto
TeQuerierParams querierParams(loadGeom, attributes);
querierParams.setParams(armadilhas);
```

Exemplo 12.26 – Consulta de instâncias agrupadas por objetos.

Os Exemplo 12.25 e 12.26 mostram como usar a classe `TeQuerier` para expressar uma seleção de objetos e suas instâncias no tempo. Uma característica importante é que através desse mecanismo não é necessário armazenar em memória todas as instâncias selecionadas, já que essas podem ser processadas e descartadas uma a uma. Cada instância é carregada para a memória somente quando é requisita através do método `TeQuerier::fetchInstance`. Uma alternativa para carregar para a memória todas as instâncias de todos os objetos de uma seleção é através da classe `TeSTElementSet`. Essa classe funciona como uma estrutura de agregação que armazena em memória as instâncias dos objetos com suas geometrias e atributos descritivos, fornecendo mecanismos para o seu percorrimento. A TerraLib fornece algumas funções de preenchimento da estrutura `TeSTElementSet` como mostramos no Exemplo 12.27.

```
// Recupera o layer de estações de coletas
TeLayer* armadilhas = new TeLayer("LAYER_ARMADILHAS");
db_->loadLayer(armadilhas);
// Cria um STElementSet para os objetos do layer de armadilhas
TeSTElementSet steSet(armadilhas);

// Carrega STElementSet carregando cada instância com todos os
seus atributos e sua geometria
TeSTOSetBuildDB(&steSet, true, true);
cout << "Número de elementos no conjunto: " <<
steSet.numElements() << endl;

// Percorre elementos e suas intâncias através de iteradores
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end()) {
```

```
  TeSTInstance st = (*it);
  cout << " Id do objeto :  " << st.objectId() << endl;
  if (sti.hasPoints()) {
     TePointSet ponSet;
     sti.getGeometry(ponSet);
     for (unsigned int i=0; i<ponSet.size(); ++i) {
        cout << " Point : " << ponSet[i].location().x() << ", ";
         cout << ponSet[i].location().y() << endl;
    }
  }
  ++it;
}
```

Exemplo 12.27 – Criação de um `TeSTElementSet`.

Essa seção mostrou os mecanismos para tratar dados espaço-temporais da TerraLib. Esses mecanismos são baseados no registro da característica temporal das tabelas de atributos no banco de dados e expressão dos conceitos de objetos com uma identidade única e persistente ao longo do tempo e de instâncias no tempo de atributos e geometrias desses objetos.

12.13 Funções de processamento de dados geográficos

Essa Seção descreve resumidamente algumas das funções de processamento oferecidas pela TerraLib, implementadas usando as estruturas e mecanismos descritos nas seções anteriores.

Matriz de Proximidade Generalizada: e uma extensão da matriz de proximidade usada em muitos métodos de análise espacial (Souza et. al, 2003). A TerraLib oferece as funções para gerar matriz de proximidade seguindo diferentes critérios, como por exemplo adjacência ou distância mínima, e para ponderar e fatiar a matriz segundo algum atributo. Além disso, uma matriz de proximidade pode ser gerada a partir de um `TeSTElementSet`.

Algoritmos de análise espacial: incluem algoritmos para geração de superfícies de kernel, índices de correlação espacial como LISA, Moran e Bayesiano local e global. Para saber mais sobre análise espacial de dados geográficos consulte Bayley e Gatrell, (1995).

*Operações Geográfica*s: combinam temas para gerar novos *layers* gerando novos atributos e novas geometrias vetoriais. Essas funções incluem a agregação de objetos por atributos, por exemplo, a agregação de municípios pelo atributo estado, gerando um *layer* de estados a partir de um *layer* de municípios. Incluem também funções de associação de dados por localização, por exemplo, atribuindo ao *layer* de municípios valores de atributos calculados a partir de todas as ocorrências de crime que acontecem dentro do município.

Geocodificação de endereços: implementa o processo de associar uma coordenada geográfica a um evento tendo por base um *layer* que contenha os endereços que se deseja encontrar. Deste modo, uma vez associado a uma localização, o endereço pode ser usado para visualização dos eventos sobre um mapa ou utilizado para análises.

Atualmente começam a ser integrados na TerraLib funções de processamento de imagens como segmentação e classificação, e interpolação de superfícies a partir de amostras.

Referências

BAILEY, T.; GATRELL, A. **Interactive Spatial Data Analysis.** London: Longman Scientific and Technical,1995.

CÂMARA, G.; SOUZA, R. C. M.; PEDROSA, B.; VINHAS, L.; MONTEIRO, A. M. V.; PAIVA, J. A.; CARVALHO, M. T.; GATTASS, M. TerraLib: Technology in Support of GIS Innovation. In: II **Simpósio Brasileiro em Geoinformática**, GeoInfo2000, 2000, São Paulo.

DAVIS, C.; CÂMARA, G. Arquitetura de Sistemas de Informação Geográfica. In: CÂMARA, G.; DAVIS, C.; MONTEIRO, A. M. V. (Org.). Introdução à ciência da geoinformação. São José dos Campos: INPE, out. 2001. cap. 3.

FERREIRA, K. R.; QUEIROZ, G. R.; PAIVA, J. A. C.; SOUZA, R. C. M.; CÂMARA, G. **Arquitetura de Software para Construção de Bancos de Dados Geográficos com SGBD Objeto-Relacionais**. p. 57-67, 2002. **XVII Simpósio Brasileiro de Banco de Dados**.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLÍSSIDES, J. **Design Patterns - Elemens of Reusable Object-Oriented Software**. Reading, MA: Addison-Wesley,1995. 395 p.

INPE-DPI. O aplicativo TerraView. Disponível em: <http://www.dpi.inpe.br/terraview>. Acesso em: maio 2005.

MySQL AB. **MySQL reference manual**. Disponível em: <http://www.mysql.org>.Acesso em: abril 2005.

Open GIS Consortium. Web Map Service Version 1.13. Disponível em: <http://www.opengeospatial.org>. Acesso em: maio 2005.

QUEIROZ, G. R. **Algoritmos Geométricos para Banco de Dados geográficos: da teoria à prática na TerraLib.**São José dos Campos, SP: INPE - Instituto Nacional de Pesquisas Espaciais, 2003. Dissertação de Mestrado, Computação Aplicada, 2003.

SHEKHAR, S.; CHAWLA, S. **Spatial Databases: A Tour.** Prentice Hall,2002.

SNYDER, J. P. **Map Projections – A Working Manual**. Washington, DC, USA: United States Government Printing Office,1987.

SOUZA, R. C.; CÂMARA, G.; AGUIAR, A. P. D. Modeling Spatial Relations by Generalized Proximity Matrices. In: V Simpósio Brasileiro de Geoinformática, 2003, Campos do Jordão. Anais do GeoInfo 2003. São José dos Campos : SBC, 2003.

STROUSTROUP, B. **C++ Programming Language - 3rd Edition.** Reading, MA: Addison-Wesley,1997. 911 p.