Programming Tutorial

TerraLib 3.1.3

TerraLib Time

Contents

1. Introduction

2. Conceptual Model

3. Geographic Databases

   3.1 TeDatabase and Drivers

   3.2 TerraLib database model

4. Layer

5. Cartographic Projection

6. Geometric Representation

   6.1 Geometries Model

   6.2 Geometry Storage Model

7. Descriptive Attributes

   7.1 Static Tables

   7.2 External Tables

8. Accessing a TerraLib database

8.1 Access using the Layer

8.2 Access using the TeDatabase class

   8.2.1 The class TeDatabasePortal

9. Theme

   9.1 View

   9.2 Visual

   9.3 Grouping of objects

10. Raster data

   10.1. Storage in a TerraLib database

11. Spatial operations

   11.1. Generic API to execute spatial operations

12. Dealing with spatio-temporal data

12.1. Data structures to represent spatio-temporal data

12.2. A querier mechanism to retrieve spatio-temporal data: TeQuerier

13. Analysis and spatial statistics

13.1. Proximity Matrix

13.2. Spatial Statistics

14. Bibliography

1. Introduction

This tutorial describes the TerraLib library in its more relevant aspects, including its conceptual model of a geographical database, its physical r of storing geometries and descriptive attributes and the mechanisms to manipulate it in different abstraction levels.

TerraLib is an open source GIS software library, written in C++, which allows a collaborative environment and its use for the development of mu GIS tools. TerraLib source code is available at the website www.terralib.org.

TerraLib provides functions to decode geographical data, spatial analysis algorithms and a conceptual model for a geographical database (Câ et al. 2000). The architecture of the library is shown in Figure 1.1. There is a central module called kernel, that contains the spatio-tempora structures, support to cartographical projections, spatial operators and the interface to stores and retrieving of spatio-temporal data in o relational databases. It also contains some mechanisms of visualization control. In a module composed by drivers, the generic interface to st and retrieving is implemented. This module also contains the routines to decode geographical data to and from a set of open and propr formats. The spatial analysis functions are implemented using the data structures of the Kernel. Finally, on top of the base modules is possi build different interfaces to the components of TerraLib using different programming environments (Java, COM, C++). These interfaces can be to implement the OpenGIS services such as the WMS - Web Map Server (OGIS, 2005).

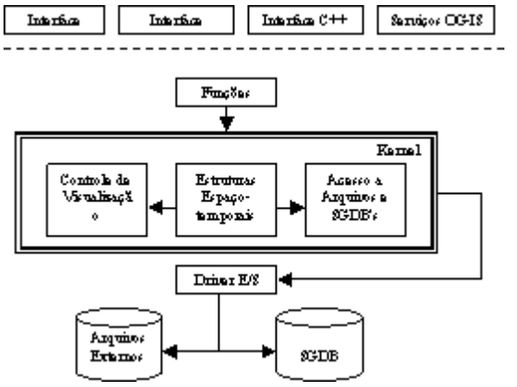


Figure 1.1 – TerraLib architecture.

One of the main features of TerraLib is its capacity to use different object-relational Database Management Systems (OR-DBMS) to store sp temporal data. This feature allows the sharing of large databases by different users´ applications. TerraLib follows a layered model of archite (Davis e Câmara, 2001), where it plays the role of the middleware between the database and the final application.

The GIS TerraView is an example of application built using TerraLib (INPE/DPI, 2005). Figure 1.2 illustrates how TerraView uses TerraLib to a a geographical database built using an OR-DBMS such as MySQL (MYSQL, 2005).

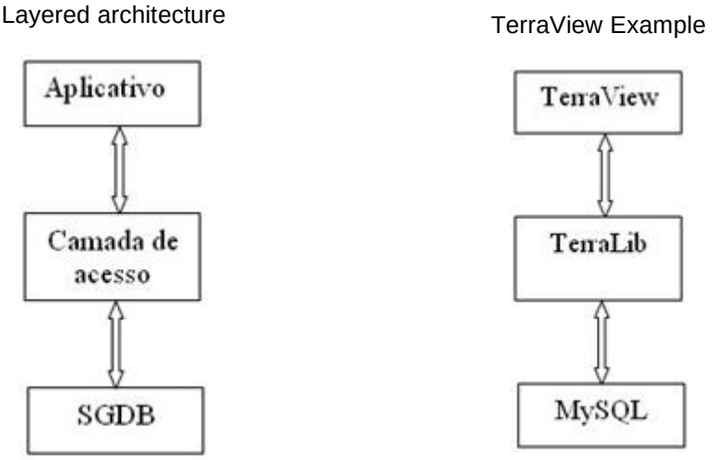


Figure 1.2 – Using TerraLib to access a geographical database.

As a software library, TerraLib is multi-platform, it can be compiled in Linux and Windows operational systems using different C++ compile

code it is standard C++ compliant. It follows a is multi-paradigm design, using the STL and templates mechanisms extensively (Stroustroup, 19
also use design patterns whenever it is useful.

This document presents the main concepts of the TerraLib library and shows some examples on how to use it. The examples are present
snippets of C++ code that can be inserted in complete programs. The examples manipulate a TerraLib database built in a particular D
(Database Management System) driver. To better understand the concepts of TerraLib and its data model, a front end program to the DBMS s
be used (for example MySQL Query Browser for MySQL). It is also useful to use TerraView to visualize the database, including its geometrical

Back to top.

2. Conceptual model

TerraLib proposes not only a model for storing geographical data in a OR-DBMS, but also a conceptual model of a geographical database. On
this model the geographical algorithms are written. The main entities of the conceptual model are:

- **Database:** represents a repository of information that contains the geographical data as well as its organization model. A TerraLib data
can be materialized in different DBMS's, commercial or open source, as long as they are either able to store binary long fields or pr
spatial extensions accessible by software.
- **Layer:** is a structure that aggregates a set of spatial information that are located over a geographical region and that shares a s
attributes. Examples of layers are thematic maps (soil or vegetation maps), cadastral maps of geographical objects (map of districts of a
or raster data such as satellite imagery. A layer knows the geographical projection of its spatial component.
Layers are inserted in the database by routines that import geographical data files in interchange formats such as shapefiles, MID-M
GeoTiff. Layers also can be generated by processing other layers already stored in database.
- **Representation:**deals with the representation model for the spatial component [1] of the data in of a layer. It can be vector or raster type
the vector type TerraLib deals with points, lines or areas geometries. It also support other representations that are more complex, such a
spaces and networks.For the raster type, TerraLib support multi-dimensional regular grids.
TerraLib allows that a geographical object of a layer to be associated to different vector representations (e.g. a city can be represented b
polygon that describes its political boundaries or by a point that represents its center). The entity representation in TerraLib store
information about the minimum boundary rectangle or the vertical and horizontal resolutions of a representations of raster type.
- **Cartographical Projection**: represents the geographical reference of the spatial component of the geographical data. The geograp
projections allow the mapping of the Earth surface to a Cartesian plane(Snyder, 1987).
- **Theme**: is used to define a selection over the data of a layer. This selection can be based on some criteria that has to be valid for the s
or descriptive component of the data. A theme also defines the visual presentation, or the graphical presentation, of the spatial compon
the objects selected by the theme. A Theme can also define a way to group objects, creating legends that characterize the groups.
- **View**: define a particular user view of the database. It defines which themes should be visualizes or processed conjointly. Besides th
each layer can be derived from a theme with a different geographical projection, the view defines the common projection to be us
visualize or process the themes that it aggregates.
- **Visual:** represents a set of presentation attributes for the geometrical representation of the data. For instance, filling and contour colo
polygons, thickness and colors for lines or symbols for points. Transparency and line styles, etc.
- **Legend:** a legend characterizes a group of data, in a theme, that should be presented with the same Visual, when they are grouped in
way.

Back to top.

3. Geographic Databases

One of the main features of TerraLib is its capacity to use different object-relational Database Management Systems (OR-DBMS) to store s
temporal data, which allows the sharing of large databases, by different users´ applications. TerraLib provides an abstract class called `TeData`
[2] that represents a TerraLib database independently of DBMS in which it is physically stored.

3.1. `TeDatabase` and Drivers

The TerraLib conceptual data model does not depend on a specific DBMS and is provided by the library through an abstract class
TeDatabase. This class contains the methods necessary to create, populate and query the database. It is derived in concrete classes called dr
that solve for the different DBMS's, commercial or freeware, its own peculiarities, so that a TerraLib application can use different DBMS's. Te
provides some drivers in its standard distribution (Figure 3.1). Drivers to other DBMS's should be created by implementing the virtual me
specified in the `TeDatabase` abstract class.

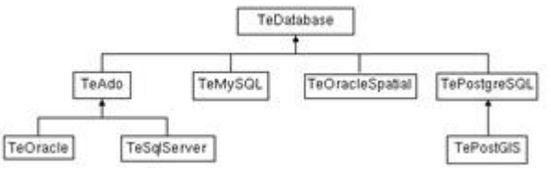


Figure 3.1 – Some concrete DBMS drivers provided by TerraLib.

Typically, TerraLib applications process pointers to the class `TeDatabase` initialized with an instance of a concrete driver, as is shown i
example bellow.

Example 3.1 - Instantiating a concrete TerraLib database.

```
int main()
{
        TeDatabase* db;
        int op;
        cout << "Select a DBMS: 1) ACCESS 2) MySQL \n";
        cin >> op;
        if (op == 1)
                db = new TeAdo();     // Uses ACCESS through ADO
        else
                db = new TeMySQL();   // Uses MySQL
// ... uses only db
        return 0;
}
```

3.2. TerraLib database model

Physically, a TerraLib database contains a set of tables, that store data and metadata. The data table store the geographical data, and the met tables store its organization:

- **Metadata Tables**: used to store TerraLib concepts. They have a pre-defined format reflecting the TerraLib classes. The set of met tables is called TerraLib Conceptual Data Model
- **Data Tables**: used to store geographic data itself - spatial and alphanumeric components. The tables that store the alphanumeric attri do not have a pre-defined format. See below an example of a table containing the attributes of the Districts of a city:

DistrictSP

ID	Name	Population	Area
3550089	Tatuapé	200000	25000
3550070	Moema	145000	60000

The tables that store a spatial or geometric component have a pre-defined format so that the TerraLib classes are able to decode the s geometries, creating instances in memory of the objects. For instance, if each district has a polygonal representation, its boundaries are stored ***polygons table***, or table that stores only polygons:

Polygons1

geom_id	object_id	num_coords	num_holes	spatial_data	...
1	3550070	...	...	...	...
2	3550089	...	...	...	...

All polygon tables have the same schema as shown above. The tables that store other kinds of geometry (points, lines or raster) have differen defined schemas.

A geographic object is then stored on these two tables shown above (alphanumeric and spatial attributes). The link between the spatial and att attributes is done through the relationship between the field **object_id** of the geometric table and a selected field of the attribute table. O example above, the geographic object *3550070* (*Moema*) is represented by polygon *1* and the object *3550089* (*Tatuapé*) by polygon *2*.

When a new database is created, the TerraLib conceptual data model is automatically created. A connection to it is opened throug `TeDatabase` class as is shown in Figure 3.2. To access existing databases the applications should open a connection to them. An applicatio keep several connections to different databases at the same time. To create a new database or to open a connection to an existing one applications should inform the necessary parameters. At the end of the execution all opened connections should be closed.

Example 3.2 - Creating a TerraLib database called "*TerraTeste*", using a MySQL DBMS in a local host "*localhost*", supposing there is a user "*root*" without an access password.

```
#include <TeMySQL.h>
int main()
{
        // DBMS server parameters
        string host = "localhost";
        string dbname = "TerraTeste";
        string user = "root";
        string password = "";
```

```
        // Create a new database
        TeDatabase* db = new TeMySQL();
        if (!db->newDatabase(dbname,user, password, host))
        {
                cout << "Error: " << db->errorMessage() << endl;
                return 1;
        }

        cout << "The database \"" << dbname;
        cout <<"\" was successfully created in the \""  << host <<  "\"";
        cout << " to the user \"" << user << "\"!"  << endl;
        // Close the database
        db->close();
        delete db;
        return 0;
}
```

Example 3.3 - Open the database created in the previous Example using a front end and check the tables that were generated. All the table the "**te**_" prefix are tables that form the conceptual data model.

The metadata tables (Figure 3.2) are used to store the concepts described in Section 2. Each layer inserted in the database generates a rec the table called **te_layer**. The field **layer_id** contains the unique identification of each layer in the database. The other fields of this table sto name and the minimum boundary rectangle of the geometries associated to the objects of the layer. It also stores a reference to its asso cartographical projection definition. Refer to the data model description available in the Data Model section of the TerraLib web site, for a con description of the metadata tables.

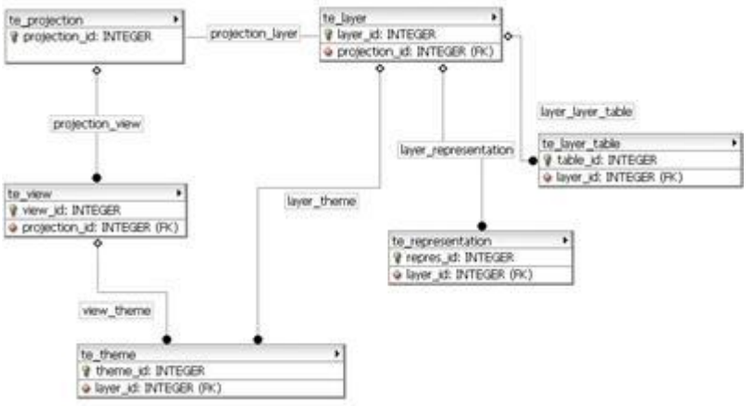


Figure 3.2 – Main tables of the conceptual model.

Each representation associated to a layer generates a record in the table **te_representation**. Each attribute table associated to a layer genera record in the table **te_layer_table**.

Each view created in the database generates a record in a table called **te_view** and each instance of cartographical projection generates a rec the table **te_projection**. Views and layers contain references to the projection table. Each theme created generates a record in the table **te_th** Each theme contains a reference to the view in which it is inserted.

In order to understand the conceptual model of TerraLib database, the contents of its metadata tables and their relationship is interesting to ob it after the execution of a typical sequence of operations:

- Creation of a database;
- Importing of a geographical data from a file;
- Creation of a view;
- Creation of a theme using the layer and inserting it in the view.

The data tables will be discussed later on, after the presentation of the geometry model of TerraLib.

Example 3.4 - In order to use a previously created database it is necessary to open a connection to it, that should be closed when the appli finishes.

```
#include<TeMySQL.h>
int main()
{
        // Open a connection to the database server
        TeDatabase* db = new TeMySQL();
```

```
        // Server parameters
        string host ="localhost";
        string bname = "TerraTeste";
        string user = "root";
        string password = "";
        if (!db->connect(host,user, password,dbname,0))
        {
                cout<< "Error: " << db->errorMessage() << endl;
                return 0;
        }
        cout << “Connection to the database: “ << dbname  << “ is opened.“;
        db->close();
        delete db;
}
```

Next, the main concepts will be presented in mode detail, including their representation in classes of the library and in the tables of the conce model.

Back to top.

4. Layer

The ***Layer*** concept in TerraLib, refers to a set of information located over the same geographic region (area), sharing the same set of attributes the layer aggregates similar "things". Some examples of layers are: thematic maps (soils maps), cadastre maps of geographic objects (districts of a city) or raster data (satellite imagery).
A layer is represented in memory as an object of the `TeLayer`[3] class and in a TerraLib database as a record of the conceptual model table **te_layer**.
The layers are created importing well-known formats of geographic data, coming from specific GIS such as: Shapefile format, from Environn Systems Research Institute, Inc. (ESRI) products, or MapInfo Interchange File (MID/MIF) from MapInfo. GeoTIFF and JPEG are general form raster data. TerraLib also has functions to generate new layers from pre-existing ones.

Example 4.1 - Creating a new layer importing a MID/MID file.

```
// Connects to a database
// Import some file
string filename = "../data/Distritos.mif";
TeLayer* newLayer = TeImportMIF(filename,db_);
if (newLayer)
        cout << "File MID/MIF was imported to the database\n";
else
{
        cout <<"Error importing the data\n";
        db_->close();
}
```

Observe that a record was created on the **te_layer** table and also two new tables were also created: one to store the polygons that represe data spatial component and another for store its attributes.

Back to top.

5. Cartographic Projection

A map projection system establishes a relation between points located on the earth surface and the corresponding points on the map proj plane. Although there are a large number of map projections, they are basically comprised of two main groups: the conformal projections, preserve angles, and the equal-area projections, which preserve areas.
Each map projection depends on different parameters such as a standard parallel, a central meridian, and/or a planimetric datum. The planir datum is defined by positioning a certain reference ellipsoid with respect to the earth surface. A projection is represented in memory as an obj the `TeProjection`[4] class and in a TerraLib database as a record of the conceptual model table called **te_projection**.

Example 5.1 - Creating two different projections and converting coordinates between them.

```
TeDatum dSAD69 = TeDatumFactory::make("SAD69");               // SAD69 Spheroid
TeDatum dWGS84 = TeDatumFactory::make("WGS84");                       // WGS84 Spheroid

// UTM: Origin Latitude -45.0
// TeCDR: "Converting from Decimal Degrees to Radians"
TeUtm* pUTM = new TeUtm(dSAD69,-45.0*TeCDR);
```

```
// Polyconic:  Origin Latitude -45.0
TePolyconic* pPolyconic = new TePolyconic(dWGS84,-45.0*TeCDR);
// Original coordinate in  UTM

TeCoord2D pt1(340033.47,7391306.21);
// Converting from UTM to polyconic
pUTM->setDestinationProjection(pPolyconic);
// Converting to Lat Long
TeCoord2D ll = pUTM->PC2LL(pt1);
// Converting to the output projection
TeCoord2D pt2 = pPolyconic->LL2PC(ll);
printf("UTM ->Polyconic \n");
printf("(%.4f, %.4f)-> ",pt1.x(), pt1.y());
printf("(%.4f, %.4f)\n",pt2.x(), pt2.y());

// Converting from Polyconic to UTM
pPolyconic->setDestinationProjection(pUTM);
ll = pPolyconic->PC2LL(pt2);
pt1 = pUTM->LL2PC(ll);
printf("\nPolyconic-> UTM \n");
printf("(%.4f,%.4f) -> ",pt2.x(), pt2.y());
printf("(%.4f, %.4f)\n",pt1.x(), pt1.y());
```

Some interchange formats of geographic data can come along with information about their cartographic projection (ex. MID/MIF, GeoTIFF). ' the cartographic projection is not known, TerraLib provides a non-geographic default coordinate system called `TeNoProjection`.

Example 5.2 - Importing a geographic data without specifying the cartographic projection, setting the correct one later.

```
TeLayer* layer = new TeLayer(layerName, db);
string filename = "../data/EstadosBrasil.shp";
string tablename = "BrasilIBGE";
string linkcolumn = "SPRROTULO";

if (TeImportShape(layer, filename, tablename, linkcolumn))
        cout >> "The shapefile was imported successfully into the
             TerraLib database!\n" << endl;
else
        cout >> "Error importing the shapefile!\n" << endl;

TeDatum sad69 = TeDatumFactory::make("SAD69");
TePolyconic* proj = new TePolyconic(sad69, -54.0*TeCDR);
layer->setProjection(proj);  //Set the correct projection
```

Check on the **te_projection** table the distinction between the projections of the imported layers on the examples 5.2 and 4.1.

On the example 5.2 the user explicitly specified the attributes table name - "*BrasilIBGE*". In addition, knowing the .dbf file that puts togeth shapefile format, the user defines the "*SPRROTULO*" field as the link between geometries and attributes of the objects or elements of the *layer*.

Back to top.

6. Geometric Representation

The geographic objects can be associated to different geometries to represent their location in space. For example, depending on the type analysis or operation being executed, a city can be represented by a point or by a polygon. Depending of the scale, a river can be represente line or by a polygon. This section describes how the geometric representations are manipulated in TerraLib.

6.1. Geometry model

As described in Section 2, in a TerraLib database, the geographical data are aggregate in layers. Layers contains set of objects, where each has a unique identification, a set of descriptive attributes and a geometrical representation. This section describes the model of geometry class TerraLib. The base class of all geometries in TerraLib is called `TeGeometry`[5].
Each geometry contains a unique identification, a reference to its MBR and the identification of the geographical object that it represents. The geometries of TerraLib are build from a bi-dimensional coordinates represented by the class `TeCoord2D`[6].
These geometries are:

- ***Points***: represented by the class `TePoint`, implemented as a single instance of a `TeCoord2D`;
- ***Lines***: composed by one or more segments are represented by the class `TeLine2D`, implemented as a vector of two or more `TeCoord2`
- ***Rings***: represented by the class TeLinearRing, are closed lines where the last coordinate is equal to the first one. The class `TeLinear` is implemented as a single instance of a `TeLine2D` that satisfies the closeness restriction;
- ***Polygons***: represented by the class `TePolygon`, are delimitation of areas that can have zero or more holes. Are implemented as a vec `TeLinearRing`. The first ring of the vector is always the external ring while the other rings, if they exist, are the holes or child of the ex ring.

To optimize the maintenance of geometries in memory the geometry classes of TerraLib are implemented using the *handle/body* design p where the implementation is separated of the interface. Besides that, the implementations are counted references. Each instance of a geo class stores the number of references to it, initializing the counter with zero when the geometry is created. Each time that one copy of the instar done, only the number of references is incremented and, each time an instance is destroyed, the number of references to it is decremented instance is really destroyed when the number of references to each reaches zero.

The geometry classes of TerraLib are derived either of the class `TeGeomSingle` or of the class `TeGeomComposite`, that represent respec that they are geometries with a single smaller element or that they are composed by a set of other smaller elements. This pattern of comp elements (Gamma et al. 1995) is also applied to classes that form sets of points, lines and polygons: classes `TePointSet`, `TeLineSet` `TePolygonSet`. The design patterns are implemented as parametrized classes as shown in Figure 6.1 what improves reusability in TerraLib c

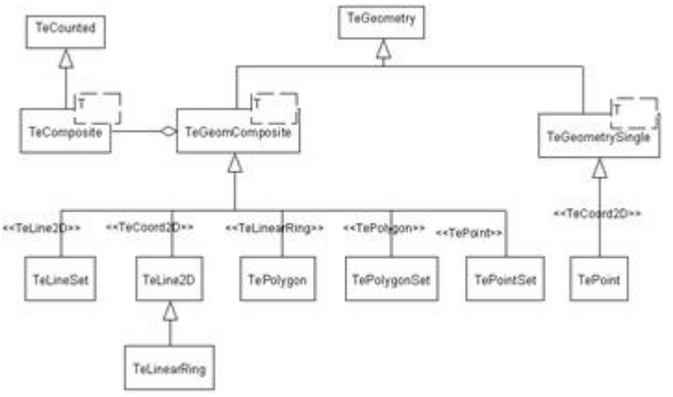


Figure 6.1 – Main geometry classes of TerraLib (adapted of Queiroz, 2003)

The raster geometries are represented by the class TeRaster that will be described in details later on.

Example 6.1 -Creation of geometries in memory.

```
// Creates a set of lines
 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);

 // Create a set of polygons
 // A simple polygon

 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");

// A polygon with a hole
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);

// Creates a set of points.
TePoint p1(40,40);
p1.objectId("ponto1");
TePointSet pos;
pos.add(p1);
```

TerraLib implements a data structure to support cell spaces, that with the support to temporal predicates, attend the requisites to implement dy models based on cell spaces. Cell spaces can be seen as a generalized raster structure where each cell stores a more than one attribute value a set of polygon that do not intercept each other. This structure has as an advantage the possibility of store conjointly, in the same structur entire set of information needed to describe a complex spatial phenomenon, which brings benefits to visualization and user interface aspec attend to this necessity, TerraLib proposes one more geometry called `TeCell`, that represents one cell in a cell space which is materializes class `TeCellSet`. Example 6.2 shows how to create a cell space from a layer stored in the database.

Example 6.2 - Creating a cell space from a layer.

```
// Retrieves the layer of states
TeLayer* estados = new TeLayer("Brasil");
db->loadLayer(estados);
// Creates a cell space over the layer extension
// where each cell has a resolution of a  1/100 of the horizontal extension
TeLayer* espaco_cel = TeCreateCells("CellsEstados", estados,
                          estados->box().width() / 100,
                          estados->box().width() / 100,
                          estados->box(), true);
```

Each cell has a unique identification and a unique reference to its position inside the cell space. To it a set of different attributes can be asso accordingly to the dynamic model being built.

6.2. Geometries storage model

As shown in the Figure 1.1, TerraLib proposes to work with geographical databases that contains the descriptive attributes as well as geometrical (or geographical) attributes (represented by points, lines, polygons, cells or raster structures). These databases can be built on DB that provide spatial extensions, that is, provide efficient mechanism of indexing and querying (Shekkar e Chawla, 2002). They also can be bu DBMS's that can store binary long fields. In TerraLib these two types of DBMS's are treated in the same way by means of the class `TeDatabas`

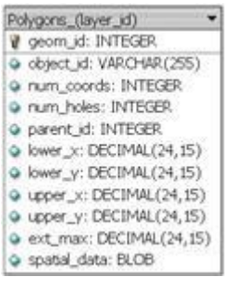
The storage model for geometries in a TerraLib database considers efficiency issues in terms of storaging as well as retrieving. It also conside existence or non-existence of the spatial extensions. Every geometry table has the following fields:

- `geom_id`: integer, the unique identification of the geometry;
- `object_id`: string, stores the unique identification of the object associated to the geometry;
- `spatial_data`: stores the geometrical data. The type of this field depends on the DBMS where the data is being stored. In DBMS's spatial extension, the type is the type provided by the extension, otherwise it is a BLOB.

In DBMS's without spatial extension, the geometry tables to store lines and polygons contains other fields used to store their MBR in four real f `lower_x, lower_y, upper_x, upper_y`. These fields are indexed using the conventional mechanism provided by the DBMS and are us retrieving operations as the spatial indexes. In DBMS's with spatial extensions the column that contains the spatial data is indexed using the s indexing mechanism provide by the spatial extension.

The Figure 6.2 shows the difference between a geometry table to store polygon in two DBMS's: with and without spatial extension. In later cas type "GEOMETRY" represents the spatial type provided by the extension. For the first case, the BLOB model, each ring of the polygon is store record of the table. The external ring contains the information about the number of holes it has and the internal rings the reference to their pare the external ring where they are inserted). This storage model allows the partial retrieving of polygons with a large number of holes (for exa during a zooming operation over a map that contains data over a large geographical area, such as the Amazon region, in small cartograp scale). As the MBR of each ring is explicitly stored in the tables it is possible to retrieve only the records that contains rings that intercept the de zooming area. This is represents an optimization in the data processing.

Polygons Table - without spatial extension



Polygons Table - with spatial extension (PostGIS)

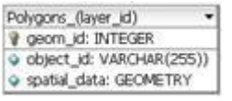
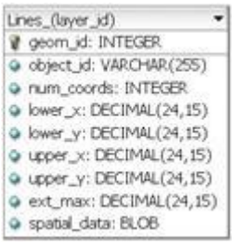


Figure 6.2 – Table model to store polygons.

The Figure 6.3 shows the model to store lines and 6.4 to store points.

Lines Table - without spatial extension



Lines Table - with spatial extension (PostGIS)

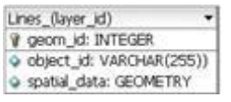
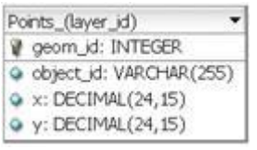


Figure 6.3 – Table model to store lines.

Points Table - without spatial extension



Points Table - with spatial extension (PostGIS)

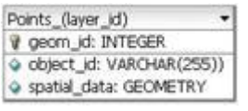


Figure 6.4 – Table model to store points.

In a TerraLib database, for each layer, each geometric representation is stored in a different table with a specific format for that representati geometric representation of a layer is described in memory as an instance of the `TeRepresentation` structure and in a TerraLib database record of the conceptual model table called **te_representation**. The geometric representations of the layers imported from the previous exa can be observed on the te_representation table. In the next example, a TerraLib function will be used to create a second geometric represen for a layer of districts represented by polygons.

Example 6.3 - Creating a new geometric representation of points associated with each district. The districts will be either represented by poly

or by points (in this case the centroid of the polygon).

```
TeLayer* distritos = new TeLayer("Distritos");
if (!db->loadLayer(distritos))
{
        cout << "Error loading layer \"Distritos\": "
          << db->errorMessage() << endl << endl;
        db->close();
        return 1;
}

// Check if the table already exists in the database
string pointsTableName = distritos->tableName(TePOINTS);
if (pointsTableName.empty() == false)
{
        if (db->tableExist(pointsTableName))
        {
                cout << "The table \"" << pointsTableName
               << "\" already exists in the database!\n\n";
                db->close();
                return 1;
        }
}
TeRepresentation* repPol = distritos->getRepresentation(TePOLYGONS);
TePointSet centroids;      // generate the centroids
if (db->Centroid(repPol->tableName_, TePOLYGONS, centroids) == false)
{
        cout << "Error creating centroids: " << db->errorMessage();
        db->close();
        return 1;
}
//  Adds a geometric representation tot he layer
distritos->addPoints(centroids);
cout << "Centroides criados!\n\n";
cout <<"\nNumber of polygons "<< distritos->nGeometries(TePOLYGONS);
cout <<"\nNumero of lines: "   << distritos->nGeometries(TeLINES);
cout <<"\nNumero of points " << distritos ->nGeometries(TePOINTS);
```

Notice on the **te_representation** table the number of records associated with the layer "Distritos", their geometric representations and the where the geometries are stored.

Back to top.

7. Descriptive Attributes

The descriptive attributes of the objects of a layer are represented in relational tables where each field represents an attribute of an object. C the fields should contain the identification of the object and its values are repeated in the geometry tables which permits the link betwee attributes and the geometry of an object.

Each layer can have one or more attribute table in the database. Each one of them id registered in a metadata table called **te_layer_table**. I table is also registered the name of the field that is its primary key and the object identifier. This field is used to link geometries and attrib Themes can choose which attribute tables of the layer they will use.

When the attribute table doesn't have any temporal information e each one of its records represents the attributes of a different object, the ta called static table. This semantic classification of the attribute tables is a characteristic of TerraLib and aims to define functions and processin are dependent of these differences among the geographical data.

Some attribute tables, called external tables, do not represent explicitly the objects contained in a layer, but they can have a field with coinc values with a field of a static table. By means of a junction operation an external table can be add information to the objects of the layer. Becau that the external table can be incorporated to the database and be registered as it becoming available to be used by all the themes of the datab

An attributes table of a layer is described in memory as an instance of the `TeTable`[7] class and in a TerraLib database as a record c conceptual model table called **te_layer_table**. In TerraLib, the attributes tables are classified according to the characteristics of the data they For example, there are attributes tables storing information that change along time (`TeAttrEvent`, `TeFixedGeomDynA` `TeDynGeomDynAttr`, `TeGeomAttrLinkTime`), media information (`TeAttrMedia`) or static information (`TeAttrStatic`).
Depending on the type, some fields of the **te_layer_table** table must be filled in. The most common attributes tables are the ones storing information. To register these tables it is necessary to know what layer they belong to and which field links with the geometric table - object ider

Example 7.1 - Creating an attribute table in memory.

```
// Create the list of attributes
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;          // a string attribute
at.rep_.numChar_ = 255;
at.rep_.name_ = "nome";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

at.rep_.type_ = TeREAL;            // a real attribute
at.rep_.name_ = "val1";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

// Creates an instance of attribute table in memory
TeTable attTable("teste2Attr",attList,"IdObjeto","IdObjeto");
TeTableRow row;
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();
```

Example 7.2 - Check on the databases previously created the data tables and metadata tables.

7.1 Static Tables

There are tables that contain attributes that do not change along the time. Besides that, they should have a field with unique values (or a pr key) that is used to link one or more geometry table. A certain layer can have one or more static attribute tables. The set of attributes, or prope of the objects contained in the layer is composed by the union of the attributes described in all of its attribute tables. The geographical data f MID/MIF, Shapefile and SPRING Geo/Tab are composed by two files: one that contains the geometry of the objects and one that contai attributes. For Shapefiles and MID/MIF files the link between the geometries and the attributes is given by its order of appearance within the that is, the first geometry and the first row of attributes in each of the files form the first object, the second geometry and the second row of attri form the second object and so on. Spring. For GEO/TAB files each geometry in the geometry file has explicitly the identification of the objec related to (Table 7.1).

Table 7.1 Geographical data file formats

Format	Geometries	Attributes	Geometry/Attributes linking
MID/MIF	*.mif	*.mid	Sequential
Shapefile	*.shp	*.dbf	Sequential
Spring Geo/Tab	*.geo	*.tab	Explicitly

The importing routines for the formats Shapefile and MID/MIF, by default, create an extra attribute column that contains values that represe order of the objects in the file, and this column is the column used to link geometries and attributes in the TerraLib database. However, whe user knows its data set and is sure that there is a column that has unique values and can be used as the object identifier. The same is va attribute tables in CSV (Comma Separated Values) and DBF formats.

Example 7.3 - Importing a MID/MIF file to a layer with the neighborhoods of Recife. Importing a second attribute table to the same layer from a CSV format.

```
string filename = "../data/BairrosRecife.MIF";
if (db->layerExist("BairrosRecife"))
{
        cout << "Error: the layer\"BairrosRecife\" already exists in the database!\n" << endl;
        db->close();
        return 1;
}
// Imports the MID/MIF file
TeLayer* newLayer = new TeLayer("BairrosRecife", db, 0);
// Select the attribute ID_ to be the object identification
if (TeImportMIF(newLayer, filename,"BairrosRecife", "ID_"))
        cout << "The file was successfully imported!\n" << endl;
else
{
        cout << "Error importing the MID/MIF file!\n";
        db->close();
        return 1;
}
// Imports the CSV file
filename = "../data/BairrosRecife2.csv";
if (db->tableExist("BairrosRecife2"))
{
        cout << "There is already a  \"BairrosRecife2\" table in the database!\n\n";
                return 1;
}
cout << "Importing a second attribute table from a CSV file...\n";
// Creates the definition of the columns represented in the CSV
TeAttributeList attList;
TeAttribute column1;
column1.rep_.name_  = "BAIRRO_ID";
column1.rep_.type_ = TeSTRING;
column1.rep_.isPrimaryKey_ = true;
column1.rep_.numChar_ = 32;
attList.push_back(column1);

TeAttribute column2;
column2.rep_.name_  = "ORDEM";
column2.rep_.type_ = TeINT;
attList.push_back(column2);

TeAttribute column3;
column3.rep_.name_  = "NOME";
column3.rep_.type_ = TeSTRING;
column3.rep_.numChar_ = 32;
attList.push_back(column3);
```

```
TeAttribute column4;
column4.rep_.name_  = "NOME_PADRAO";
column4.rep_.type_ = TeSTRING;
column4.rep_.numChar_ = 32;
attList.push_back(column4);

TeTable attTable2("BairrosRecife2",attList, "BAIRRO_ID",
                      "BAIRRO_ID", TeAttrStatic);
if (!newLayer->createAttributeTable(attTable2))
{
        cout << "Error creating the table \"BairrosRecife2\" in the database!\n\n";
        return 1;
}
TeAsciiFile csvFile(filename);
while (csvFile.isNotAtEOF())
{
        TeTableRow trow;
        csvFile.readNStringCSV (trow, 4, ';');
        if (trow.empty())
                break;
        csvFile.findNewLine();
        attTable2.add(trow);
}
if (attTable2.size() > 0)
{
        if (!newLayer->saveAttributeTable(attTable2))
        {
                cout << "Error inserting the  \"BairrosRecife2\" table in the database  !\n\n";
                return 1;
        }
        attTable2.clear();
}
cout << "Table was successfully imported...\n";
db->close();
```

Check on the database the records generated in the table **te_layer_table**.

7.2 External Tables

External tables are attribute tables that can add information to the objects of one or more layers, but they do not have an attribute that is direct to the geometries. These tables can be considered as non spatial attribute tables. The external tables can be part of a theme by linking one fields to one of a field of a static table. The external tables are dynamic set of attributes that is not explicitly attached to a particular layer, and can be attached to different themes built from different layers.

Example 7.4 - Importing an external table.

```
// Importing a dbf file
string filename = "../data/SOCEC.dbf";
if (db->tableExist("SOCEC"))
{
        cout << "There is already a table called \"SOCEC\" in the database!\n\n";
        return 1;
}
if (TeImportDBFTable(filename,db))
        cout << "The dbf file \"SOCEC.dbf\" was successfully imported to the database!\n\n";
else
{
        db->close();
        cout << "Fail to import the dbf file!\n";
        return 1;
}
TeAttrTableVector tableVec;
if (db->getAttrTables(tableVec, TeAttrExternal))
```

```
{       cout << "External tables in the database: "
           << tableVec.size() << endl;
        TeAttrTableVector::iterator it = tableVec.begin();
        while (it != tableVec.end())
        {  cout << (*it).name() << " Primary key column: "
              << (*it).uniqueName() << endl << endl;
                 ++it;
        }
}
db->close();
```

Back to top.

8. Accessing a TerraLib Database

8.1 Access using the Layer

The importing routines are written using the methods of the class `TeLayer`. This class provides the methods for the inserting of geometrie attributes. Through these methods the records in the conceptual model tables are filled correctly with the correct reference to the layers identifi when necessary.

The Example 8.1 should be followed as a continuation of the Example 6.2 and Example 6.3 e shows how to create a layer and how to u methods to insert the geometries and attributes created in memory to the database.

Example 8.1 - of geometries and attributes in a database using the class `TeLayer`.

```
// Create a projection
TeDatum mDatum = TeDatumFactory::make("SAD69");
TeProjection* pUTM = new TeUtm(mDatum,0.0);

// Create a new layer called "TesteLayer"
string layerName = "TesteLayer";
if (db->layerExist(layerName))
{
        cout << "There is already a layer called  \"TesteLayer\" in the database!\n\n";
        db->close();
        return 1;
}

TeLayer* layer = new TeLayer(layerName, db, pUTM);
if (layer->id() <= 0)      // The layer wasn't created correctly
{
        cout << "Error: " << db->errorMessage() << endl;
        db->close();
        return 1;
}

// Adds the line representation in a table called "TesteLayerLines"
if (!layer->addGeometry(TeLINES, "TesteLayerLines"))
{
        cout << "Error: " << db->errorMessage() << endl;
        db->close();
        return 1;
}

// Creates a set of lines
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(500,600));
ele.add(TeCoord2D(600,600));
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);

// Adds the set of lines to the database
if (!layer->addLines(ls))
{
         cout << "Error: " << db->errorMessage() << endl;
         db->close();
         return 1;
}
else
cout << "Lines inserted in the new layer\n";

//Created a set of polygons
// A simple polygon
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");

// A polygon with a hole
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);
```

```
// Adds the set of polygons to the layer.
// As the method addGeometry wasn't called before, the table will have a standard name
if (!layer->addPolygons(ps))
{
        cout << "Error: " << db->errorMessage() << endl;
        db->close();
        return 1;
}
else
        cout << "Polygons inserted in the new layer!\n";

// Creates a set o points
TePoint p1(40,40);
p1.objectId("ponto1");
TePoint p2(65,65);
p2.objectId("ponto2");

TePointSet pos;
pos.add(p1);
pos.add(p2);
// Adds the set of points to the layer.
// As the method addGeometry wasn't called before, the table will have a standard name
if (!layer->addPoints(pos))
{
        cout << "Error: " << db->errorMessage() << endl;
        db->close();
        return 1;
}
else
        cout << "Points inserted in the new layer!\n";

// Creates an attribute tables
// Defines the list of attributes
TeAttributeList attList;
TeAttribute at;
at.rep_.type_ = TeSTRING;
at.rep_.numChar_ = 16;
at.rep_.name_ = "object_id";
at.rep_.isPrimaryKey_ = true;
attList.push_back(at);

at.rep_.type_ = TeSTRING;       // an string attribute
at.rep_.numChar_ = 255;
at.rep_.name_ = "nome";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

at.rep_.type_ = TeREAL; // a float attribute
at.rep_.name_ = "val1";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

at.rep_.type_ = TeINT;          // an integer attribute
at.rep_.name_ = "val2";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

// Creates an attribute table called "TesteLayerAttr"
TeTable attTable("TesteLayerAttr", attList, "object_id", "object_id");          // creates the table in memory
if (!layer->createAttributeTable(attTable))
```

```
{
        cout << "Error: " << db->errorMessage() << endl << endl;
        db->close();
        return 1;
}


// each line is related to the object by the field object_id
TeTableRow row;
row.push_back("reta");
row.push_back("An straight line");
row.push_back("11.1");
row.push_back("11");
attTable.add(row);
row.clear();

row.push_back("ele");
row.push_back("A mountain shaped line");
row.push_back("22.2");
row.push_back("22");
attTable.add(row);
row.clear();

row.push_back("spoli");
row.push_back("A simple polygon");
row.push_back("33.3");
row.push_back("33");
attTable.add(row);

row.clear();
row.push_back("ponto1");
row.push_back("A point");
row.push_back("55.5");
row.push_back("55");
attTable.add(row);
row.clear();

row.push_back("ponto2");
row.push_back("Another point");
row.push_back("66.6");
row.push_back("66");
attTable.add(row);
row.clear();

row.push_back("cpoli");
row.push_back("A polygon with hole");
row.push_back("44.4");
row.push_back("44");
attTable.add(row);
row.clear();

// Saves the table in the database
if (!layer->saveAttributeTable( attTable ))
{
        cout << "Error: " << db->errorMessage() << endl << endl;
        db->close();
        return 1;
}
else
cout << "Objects inserted in the new layer!\n\n";
```

The example 8.1 shows how to control the insertions of each kind of geometry (points, lines or polygons), possibly specifying the name fo geometry tables (as shown in the case of lines geometries). It is useful to check the database after the running of this example. It should be che the contents of the metadata tables specially the references to the names of the geometry tables, the names of the attribute tables, to the identifier and the registering of the column names used to relate geometry and attributes. Figure 8.1 shows the contents of these tables and relationship. The solid lines represent the physical relationships between the fields and the dashed lines represent the relationships based o contents of the fields. For example, the name of the attribute tables and the name of the field that has the linking with the geometries are regis on the table **te_layer_table**.

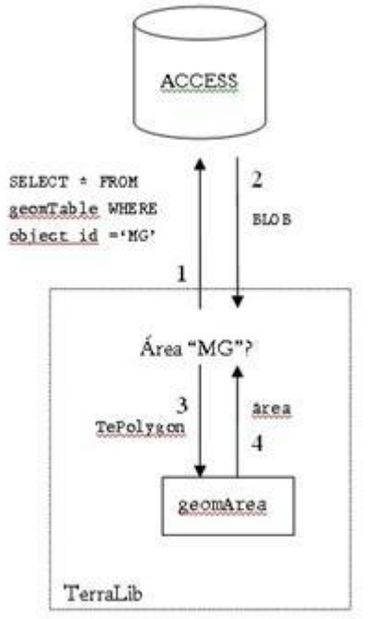


Figure 8.1 – Contents of the database after running the Example 8.1.

The retrieving of the data stored in the database can also be done directly using the class `TeLayer`. This class provides some methods to re the geometries of a certain type as can be seen in Example 8.2.

Example 8.2 - Retrieving the geometries of a *layer*.

```
TePolygonSet ps2;
layer1->getPolygons(ps2);
cout << "Number of object with polygon geometry: " << ps2.size();
ps2.clear();
// retrieves only the objects with the identifier "spoli"
layer1->loadGeometrySet("spoli", ps2);
```

The class `TeLayer` has just a few methods to retrieve the geometries and the attributes, but the class `TeDatabase` is a more complete interfa access the database as will be described bellow.

8.2 Accessing the database using the `TeDatabase` class

The class `TeDatabase` is a complete interface to manipulate the database, which allows the inserting, altering and retrieving the entities c conceptual model as well as the geographical data. It contains the full set of methods to create the metadata tables, the geometry and attri tables as can be seen in Example 8.3. This example shows how to retrieve a layer from the database and row to remove it later. The removin layer implies that all of its data and metadata are physically removed of the database as well as all the instances of the conceptual model that to it. For example, when a layer is deleted from the database, all themes created from it are also removed.

Example 8.3 - Methods for creation and altering of tables available in the class `TeDatabase`.

```
db->createConceptualModel(); // creates the conceptual model
// creates a table to store polygons
db->createPolygonGeometry("TablePoligonos");

// create a table to stores descriptive attributes
TeAttributeList meusAttributos;
TeAttribute coluna;
coluna.rep_.type_ = TeSTRING;
coluna.rep_.name_ = "coluna1";
coluna.rep_.numChar_ = 10;
meusAttributos.push_back(coluna);
```

```
db->createTable("TableAtributos", meusAttributos);

// creates a new column in a existing table
TeAttributeRep novaColuna;
novaColuna.type_ = TeSTRING;
novaColuna.name_ = "novaCol";
novaColuna.numChar_ = 10;
db->addColumn("TableAtributos", novaColuna);
// retrieves a layer
TeLayer *layer = new TeLayer("Distritos");
// loads the information about the layer called Distritos
db->loadLayer(layer);
// removes the layer
db->deleteLayer(layer->id());
```

The class `TeDatabase` also offers methods to insert and retrieve geometries from the database as is shown in Example 8.4.

Example 8.4 - Insertions using the class `TeDatabase`.

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

TeLineSet lineset;
db->insertLineSet ("tabLine", lineset);

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

TeTable tabAttributes;
insertTable(tabAttributes);
```

The class `TeDatabase` is also able to submit queries to the database written in SQL - Structured Query Languages as can be seen in the Exa 8.5. Even though the SQL is considered standard for the relational databases, different DBMS may exhibit different capacities in terms of exe of some SQL commands. The method `TeDatabase::execute` returns true or false whether the command was successfully executed respectively. The class `TeDatabase` stores the error returned by the DBMS when trying to execute the last SQL command submitted. **Examp** - Execution of a SQL command.

```
string sql = "UPDATE TableAtributos SET novaCol = 'xxx' ";
if (db->execute(sql))
        cout << "SQL command executes successfully!";
else
        cout << "Error: " << db->errorMessage();
```

The method `TeDatabase::execute` should not be used to submit commands that are queries to the database, that is, commands that results. The typical commands that should be submitted are those that alter tables and records of tables.

8.2.1. The class `TeDatabasePortal`

There is a mechanism to query the database that is more flexible than those provided by the use of the class `TeDatabase`. This mechani implemented using the class `TeDatabasePortal`. The query portals are created from any instance of the class `TeDatabase` that has an o connection to a TerraLib database. The portals can submit a SQL query and return the resulting record set to the application. A database can as much portals as necessary, but after having their resulting record set processed they should be deleted. The Example 8.6 shows a typical a query portal.

Example 8.6 - Using the `TeDatabasePortal`.

```
// Gets a portal to the database
TeDatabasePortal* portal = db->getPortal();
if (!portal)
         return -1;

// Submit a SQL command
string sql = "SELECT * FROM TableAtributos";
if (!portal->query(sql))
```

```
{
         cout << "Could not execute..." << db->errorMessage();
   delete portal;
   return -1;
}
// Consuming the resulting records
while (portal->fetchRow())
{
         cout << "Attribute 1: " << portal->getData(0) << endl;
   cout << "Attribute 2: " << portal->getData("nome") << endl;
   cout << "Attribute 3: " << portal->getDouble(2) << endl;
}
// Releases the portal for a new query
portal->freeResult();

// Submits a new query
sql = "SELECT SUM(val1) FROM TableAtributos WHERE val1 > 33.5";
if (portal->query(sql) && portal->fetchRow())
         cout << "Sum " << portal->getDouble(0);
delete portal;
```

When analyzing the example above, it can be seen that after calling the method `TeDatabasePortal::query` the resulting record set are ava to be used sequentially. An internal pointer is positioned in a before the first record position. Each time the m `TeDatabasePortal::fetchRow` the internal pointer is incremented and the value True is returned if is now in a valid position or False othe This makes possible to write a loop that passes over all the records of record set.

The fields of a record returned in portal query can be accessed in two ways: using the order defined in the selection clause of the query (start zero), or directly by the name of the field. The value of the field can be obtained as a string using the method `TeDatabasePortal::get` independently of the type of the field or using one of the methods that specify the return type (`getDouble, getInt, getDate or getB` Using the second way the application should know the type of the field being accessed.

The class `TeDatabasePortal` can also execute queries over a geometry table and access the resulting fields as the geometry types of Ter as is shown in Example 8.7.

Example 8.7 - Using a portal to retrieve geometries

```
// Submit a query to a geometry table
string q =" SELECT * FROM Polygons11 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);
         // uses the polygon that was returned
} while (flag);
delete portal;
```

Back to top.

9. Theme

A theme represents a subset of objects or elements of a layer. This subset contains all objects of a layer or some selected objects accordin restriction on the attributes, on geometries or on temporal information. The theme contains information about the visual of the data geometry theme can also define a way of grouping selected objects generating legends (`TeLegend`).

The theme is described in memory as an instance of the `TeTheme` class and in a TerraLib database as a record of the conceptual model table **te_theme**.

The objects identifiers of a layer that belong to a theme are registered in a collection table. The collection tables are part of the TerraLib conc

model. There is a collection table associated to each created theme, which optimizes the handling of the objects of this theme.

9.1 View

The TerraLib database is visualized through a View which defines what themes will be simultaneously presented. Since each theme can come layers with different projections, the view also determines a common projection to present its themes. Any themes that are not in the view proje will be remapped. However, it is important to notice that a non-cartographic coordinates system ("NoProjection") cannot be remapped for any projection. The view is described in memory as an instance of the `TeView` class and in a TerraLib database as a record of the conceptual r table called **te_view**.

9.2 Visual

A visual defines a set of characteristics for the graphical presentation of the data geometry. For polygons it is possible to define: filling color and transparency, and contour color type and width. As to lines, it is possible to define their color, type and width.

The visual is described in memory as an instance of the `TeVisual` class and in a TerraLib database as a record of the conceptual model called **te_visual**.

Example 9.1 - Creating a view and two themes over a layer - one theme with no restrictions and another with attributes restriction.

```
// Connects to a  MySQL server
//Loads the layer "Distritos" that contains the districts of São Paulo city
TeLayer* dist = new TeLayer("Distritos");
db->loadLayer(dist);

TeProjection* proj = dist->projection();
// Creates a view with the same projection as the layer
string viewName = "SaoPaulo";
// Check if the view already exists in the database
if (db->viewExist(viewName))
{
        db->close();
        return 1;
}
TeView* view = new TeView(viewName, user);
view->projection(proj);
if (!db->insertView(view))          // saves the view in the database
{
        db->close();
        return 1;
}
// creates a theme without restriction
TeTheme* theme = new TeTheme("DistritosSaoPaulo", dist);
view->add(theme);

// defines the geometry presentation visual
// Polygons will be blue
TeColor color;
TeVisual polygonVisual(TePOLYGONS);
color.init(0,0,255);
polygonVisual.color(color);
theme->setVisualDefault(polygonVisual, TePOLYGONS);
// Points will be red
TeVisual pointVisual(TePOINTS);
color.init(255,0,0);
pointVisual.color(color);
pointVisual.style(TePtTypeX);
theme->setVisualDefault(pointVisual, TePOINTS);

// Make all the representations visible
int allRep = dist->geomRep();
theme->visibleRep(allRep);

// Theme will use all the attribute tables of the layer
theme->setAttTables(dist->attrTables());
```

```
// Saves the theme in the database
if (!theme->save())
{
        db->close();
        return 1;
}

// Builds the collection table associated to the theme
if (!theme->buildCollection())
{
        db->close();
        return 1;
}
cout << "The theme \"DistritosSaoPaulo\" without restriction was created!\n";
// Creates a theme with attribute restriction:
// Districts with a population over 100,000 people

TeTheme* themeRest = new TeTheme("Pop91GT100000", dist);
themeRest->setAttTables (dist->attrTables());

// Sets the attribute restriction
string restAttr = " Pop91 > 100000 ";
themeRest->attributeRest(restAttr);
//  Make all the representations visible
themeRest->visibleRep(allRep);

// Defines the presentation visual
themeRest->setVisualDefault(polygonVisual, TePOLYGONS);
themeRest->setVisualDefault(pointVisual, TePOINTS);
// Insert theme in the view
view->add(themeRest);
// Save the theme to the database
if (!themeRest->save())
{
        db->close();
        return 1;
}

// Builds the collection table associated to the theme
if (!themeRest->buildCollection())
{
        db->close();
        return 1;
}
cout << "The theme  \"Pop91GT100000\" with an attribute restriction was created!\n\n";
db->close();
```

Observe on the database the collection tables and the records on the te_theme and te_view tables. Use the TerraView software to visualiz
TerraLib database.

A theme can decide which attribute tables of a layer it will use as is shown in the example 9.2. After running this example, check in the databas
records generated in the table **te_theme_table**.

Example 9.2 - Creation of a theme selecting only the second attribute table of a layer.

```
TeLayer* recife = new TeLayer("BairrosRecife",db);
TeTable attTable2;
recife->getAttrTablesByName("BairrosRecife2",attTable2);
TeView* viewrec = new TeView("Recife",user);
viewrec->projection(recife->projection());
if (!db->insertView(viewrec))       // saves the view in the database
{
```

```
        cout << "Fail to insert the view  \"Recife\" no banco: " <<
            db->errorMessage() << endl;
        db->close();
        return 1;
}
TeTheme* themerec = new TeTheme("Recife", recife);
viewrec->add(themerec);
themerec ->addThemeTable(attTable2);
themerec->visibleRep(recife->geomRep());
if (!themerec->save() || !themerec->buildCollection())
{
        cout << "Error trying to create the theme \' Recife \': "
          << db->errorMessage() << endl;
        db->close();
        return 1;
}
cout << "Theme created ..\n";
db->close();
```

9.3 Grouping of the Theme Objects

The objects of a theme can be grouped based on their attributes. The information (type, number of groups and visual) about these groups is s on the themes. There are several grouping functions used in TerraLib, such as unique value, quantis and equal step. Each generated gr associated to a legend. Each legend points to a visual and has information about the group - upper and lower values, number of objects belong it and its label.

The theme grouping is described in memory as an instance of the `TeGrouping` class and in a TerraLib database as a record of the conc model table called te_grouping. Each one of the groups is defined in memory as an instance of the `TeLegendEntry` class, which co information about the group and its presentation visual. In a TerraLib database, the groups are stored on the **te_legend** table and their visuals **te_visual** table.

Example 9.3 - Creating a grouping of an existing theme.

```
// Load the theme DistritosSaoPaulo.
// Created with a layer of the metropolitan districts of São Paulo
TeTheme* trmsp = new TeTheme("DistritosSaoPaulo");
if (!db->loadTheme(trmsp))
{
        db->close();
        return 1;
}
trmsp->resetGrouping();    // Reset the grouping previously created

TeAttributeRep rep;
// The grouping based on the attribute called "area_km2"
// that contains the area of the districts
rep.name_ = " area_km2 ";
rep.type_ = TeREAL;

TeGrouping equalstep3;     // defines a way to group the objects
equalstep3.groupAttribute_ = rep;
equalstep3.groupMode_ = TeEqualSteps;       // equal steps in three groups
equalstep3.groupNumSlices_ = 3;

// generate the grouping
if (!trmsp->buildGrouping(&equalstep3))
{
        db->close();
        return 1;
}

// associates a different visual to each group
TeVisual visual(TePOLYGONS);               // group 1: dark green
```

```
TeColor color(0,200,0);
visual.color(color);
trmsp->setGroupingVisual(1,visual,TePOLYGONS);

color.init(0,150,0);
visual.color(color);                // group 2: green
trmsp->setGroupingVisual(2,visual,TePOLYGONS);

color.init(0,100,0);
visual.color(color);                // group 3: light green
trmsp->setGroupingVisual(3,visual,TePOLYGONS);

if (!trmsp->saveGrouping())                 // saves the  grouping
{
        db->close();
        return 1;
}
db->close();
```

Check the changes on the collection, grouping, visual and legend tables. Use the TerraView software to visualize the generated grouping.

Back to top.

10. Raster data

Dados matriciais, ou raster, na TerraLib são considerados como um tipo de geometria associada a um determinado objeto e são armazenado Tables de geometria raster segundo um determinado padrão. A interface de manipulação de dados raster na Terralib é feita através das classe

- `TeRaster`[9]: generic class for handling raster data independent from format, size of each raster element or storage device. This provides the `setElement` and `getElement` methods used for developing the applications.
- `TeRasterParams`:structure that stores all specific parameters of a particular raster. These parameters include: number of lines, column bands, type of element, number of bytes per element, projection, bounding box etc. Each`TeRaster` has a `TeRasterParams`.
- `TeDecoder`: class that allows the access to specific raster data. The decoder associated to raster data can be explicitly chosen or in from a parameter, for instance, extensions on raster data file names.

It is possible to traverse some elements of raster data using the `TeRaster` class and its parameters:

```
void show (TeRaster* rst)
{
    TeRasterParams params = rst->params();
    cout << "Nro Bands " << params.nBands() << endl;
    cout << "Nro Lines: " << params.nlines_ << endl;
    cout << "Nro Columns: " << params.ncols_ << endl;
    cout << "Projection " << params.projection()->name() << endl;
    double val;
    if (rst->getElement(45,2,val,0))
       cout << "value of the element of the line 2 and column 45: " << val << endl;
    if (rst->getElement(67,2,val,0))
       cout << "value of the element of the line 2 and column 67: " << val << endl;

    if (rst->getElement(300,2,val,0))
       cout << "value of the element of the line 2 and column 300 " << val <<endl;
         else
       cout << "there are no values at line 2 and column 300" << endl;
}
```

Example 10.1 - Creation of same raster data in different formats and devices, and using the routine described above to check some inform about the data.

Part A - Handling a raster data in memory.

```
void main()
{
```

```
        // initializes the decodification mechanism
        TeInitRasterDecoders();
        // defines a projection
        TeProjection* noproj = new TeNoProjection();

        // defines the parameters of a raster data in memory
        TeRasterParams  params;
        params.projection(noproj);                  // projection
        params.setPhotometric(TeRasterParams::TeMultiBand);// photometric interpretation
        params.nBands(1);                                   // number of bands
        params.setDataType(TeFLOAT);                // size of the element
        params.boundingBoxLinesColumns(0,0,99,99,100,100);
        params.decoderIdentifier_ = "MEM";  // righ decoder
        params.mode_ = 'c';                                         // openning for creation

        cout << "Box: [" << params.box().x1_ << ",";
        cout << params.box().y1_ << ",";
        cout << params.box().x2_ << ",";
        cout << params.box().y2_ << "]\n";
        cout << "Bounding Box: [" << params.boundingBox().x1_ << ",";
        cout << params.boundingBox().y1_ << ",";
        cout << params.boundingBox().x2_ << ",";
        cout << params.boundingBox().y2_  <<  "]\n\n";

        TeRaster* rastermem = new TeRaster(params);
        rastermem->init();
        if (!rastermem->status())
        {
                cout << "Error creating raster data in memory." << endl;
                return 1;
        }
        // put value in some elements
        for (int l=0; l<params.nlines_; ++l)
        {
                for (int c=0; c<params.ncols_; ++c)
                        rastermem->setElement(c,l,c+1);
        }
        show(rastermem);
}
```

Part B - Accessing raster data in a non-formatted binary file without a header.

```
void main()
{
        // initializes the decodification mechanism
        TeInitRasterDecoders();
        // defines the parameters of a raster data in a raw file
        TeRasterParams  params;
        params.projection(noproj);
        params.setPhotometric(TeRASTERMULTIBAND);
        params.nBands(1);
        params.setDataType(TeUNSIGNEDCHAR);
        // bounding box parameters
        params.topLeftResolutionSize(7404193,321045,30,30,589,703,true);
        params.decoderIdentifier_ = "MEMMAP";
        params.fileName_ = "../data/sp_589x703.raw";
        params.mode_ = 'r';
        cout << "Box: [" << params.box().x1_ << ",";
        cout << params.box().y1_ << ",";
        cout << params.box().x2_ << ",";
        cout << params.box().y2_  <<  "]\n";
```

```
        cout << "Bounding Box: [" << params.boundingBox().x1_ << ",";
        cout  << params.boundingBox().y1_<< ","  ;
        cout << params.boundingBox().x2_ << "," ;
        cout << params.boundingBox().y2_  <<  "]\n\n";
        TeRaster* rasterraw = new TeRaster(params);
        if (!rasterraw->init())
        {
                cout << "Error reading raster data in a raw file." << endl;
                return;
        }
         show(rasterraw);
}
```

Part C - Accessing raster data from files in different formats. Getting the parameters from the file.

```
void main()
{
        // initializes the decodification mechanism
        TeInitRasterDecoders();
        TeRaster* rastertif = new TeRaster("../data/nat1.tif");
        rastertif->init();
        if (!rastertif->status())
        {
                cout << "Error initializing the data from the file." << endl;
                return;
        }
        cout << " \n --- Raster TIFF --- \n";
        show(rastertif);

        TeRaster* rasterjpg = new TeRaster("../data/ sampa.jpg");
        if (!rasterjpg ->init())
        {
                cout << "Error initializing the data from the file." << endl;
                return;
        }
        cout << " \n --- Raster JPEG ---- \n";
        show(rasterjpg);
}
```

10.1. Storage in a TerraLib database

Likewise vector data, there is also a special TerraLib pattern for storing raster data in relational tables considering storage optimization and re efficiency. Thus, raster data are stored in binary long fields on relational tables or in abstract data types provided by DBMS with spatial exte and coded and decoded through the TerraLib classes.

The TerraLib conceptual model presumes that raster data should be divided into fixed sized blocks before being stored in the database. Each is spatially indexed, has a unique identifier and is stored on a record of a raster table. This storage is done using the `TeImportRaster` function that imports raster data as a geometry associated with a specific object of a layer.

Example 10.2 - Importing raster data to a TerraLib database.

```
#include <TeDatabase.h>
#include <TeMySQL.h>
#include <TeInitRasterDecoders.h>
#include <TeImportRaster.h>
int main()
{
        TeInitRasterDecoders(); // initializes the decodification mechanism
        TeRaster image("../data/sampa.jpg"); // accessing the input image
        if (!image.init())
        {
                cout  << "Can not access the input image!" << endl;
                return 1;
```

```
        }
        // server parameters
        string host = "localhost";
        string dbname = "TerraTeste";
        string user = "root";
        string password = "";
        TeDatabase* db = new TeMySQL();
        if (!db->connect(host, user, password, dbname))
        {
                image.clear();
                cout << "Error: " << db->errorMessage() << endl;
                return 1;
        }
        string layerName = "SampaJPEG";
        if (db->layerExist(layerName))
        {
                cout << "Database already has a layer called \"";
                cout << layerName << "\"!" << endl << endl;
                db->close();
                return 1;
        }
        // creates a layer to receive the raster geometry
        TeLayer* layer = new TeLayer(layerName, db, image.projection());
        if (layer->id() <= 0)
        {
                image.clear();
                db->close();
                cout << "Destination layer couldn't be created!\n"
                        << db->errorMessage() << endl << endl;
                return 1;
        }
        // imports the raster to the layer
        if (!TeImportRaster(layer, &image, 128, 128))
        {
                image.clear();
                db->close();
                cout << "Fail to import image!\n" << endl;
                return 1;
        }
        db->close();
        cout << "JPEG image was imported!\n\n";
        return 0;
}
```

The import function also allows that a set of raster data be aggregated on the same representation, creating a mosaic. Thus, the secon subsequent data should be imported to an already existent geometry of a layer.

Example 10.3 - Building a mosaic with two files of different images (nat1.tiff and nat2.tiff).

```
#include <TeDatabase.h>
#include <TeMySQL.h>
#include <TeInitRasterDecoders.h>
#include <TeImportRaster.h>
int main()
{
        TeInitRasterDecoders();    // initializes the decodification mechanism
        //access the first image
        TeRaster img1("../data/nat1.tif");
        if (!img1.init())
        {
                cout << "Can not access the first image!" << endl << endl;
                return 1;
```

```
}
TeRaster img2("../data/nat2.tif");
if (!img2.init())
{
        cout << "Can not access the second image!" << endl << endl;
        return 1;
}
string host = "localhost";
string dbname = "TerraTeste";
string user = "root";
string password = "";

TeDatabase* db = new TeMySQL();
if (!db->connect(host, user, password, dbname))
{
        cout << "Error: " << db->errorMessage() << endl << endl;
        return 1;
}

// creates a layer to receive the mosaic (same projection as the first image)
string layerName = "NatividadeMosaic";
if (db->layerExist(layerName))
{
        db->close();
        cout << "Database already has a layer called \"";
        cout << layerName << "\"!" << endl << endl;
        return 1;
}
TeLayer* layer = new TeLayer(layerName, db, img1.projection());
if (layer->id() <= 0)
{
        db->close();
        cout << "Destination layer couldn't be created!\n" << db->errorMessage() << endl;
        return 1;
}

// import the first image to the layer
if (!TeImportRaster(layer,&img1,256,256,TeRasterParams::TeNoCompression,"",255,
                    true,TeRasterParams::TeExpansible))
{
        db->close();
        cout << "Fail to import the first image.\n\n!";
        return 1;
}
else
        cout << "The first image was imported to the database!\n\n";

delete layer;
layer = new TeLayer(layerName, db);
        // mosaic the second image to the same layer
if (!TeImportRaster(layer, &img2, 256, 256, TeRasterParams::TeNoCompression, "", 255,
                    true, TeRasterParams::TeExpansible))
{
        db->close();
        cout << "Fail to import the second image.\n\n!";
        return 1;
}
else
        cout << "The second image was imported!"<< endl << endl;
db->close();
return 0;
```

```
}
```

The retrieval of raster data stored in a TerraLib database is directly performed by a layer.

Example 10.4 - Retrieving the raster geometry associated with two different objects of a layer called "Brasilia".

```
TeLayer* brasLayer = new TeLayer("Brasilia", db); // retrieves the layer
TeRaster* bras = brasLayer->raster();       // retrieves the first raster

TeRaster* sul = brasLayer->raster("sul");   // geometry of the object "sul"
TeRaster* norte = brasLayer->raster("norte"); // geometry of the object "norte"
```

Back to top.

11. Spatial Operations

TerraLib provides a set of functions to perform spatial operations over geographic data (a revision of the literature and a detailed description spatial operators can be founded at Ferreira, (2003). These spatial operations can be grouped as:

1. **Determination of topological relationships among vector geometries**: based on the 9-intersection dimensionally extended where relationships are formalizes as touch, contain, within, covered by etc.
2. **Metric operations**: area calculation, length or perimeter and geometries distance.
3. **Operations that generate new geometries**: buffer (generates a new geometry around another geometry considering a certain dista centroid and convex geometry.
4. **Operations that combine geometries**: such as difference, union, intersection or symmetrical difference.
5. **Zonal and focal operations over raster data**: as the obtaining of statistical measures over a region of the raster data and clipping of a area.

The basic functions to perform these operations are defined as functions on the files `TeGeometryAlgorithms`, `TeBufferRegion` `TeOverlay` of the kernel module. They receive as parameters the TerraLib geometric types (see example below).

Example 11.1 - Verifying which points are inside a certain polygon, where the geometries are defined in memory as instances of geometry cl `TePolygon` and `TePoint`.

```
// defines a precision of the operations
TePrecision::instance().setPrecision(0.001);

// creating a polygon
TeLine2D line;
line.add(TeCoord2D(480275.38, 6667799.34));
line.add(TeCoord2D(483337.32, 6666464.65));
line.add(TeCoord2D(486006.70, 6668898.50));
line.add(TeCoord2D(484593.50, 6671410.86));
line.add(TeCoord2D(481688.58, 6672431.51));
line.add(TeCoord2D(480275.38, 6667799.34));
TeLinearRing ring(line);
TePolygon poly;
poly.add(ring);

//calculate the area
double area = TeGeometryArea(poly);
//calculate the perimeter
double len = TeLength(ring);
//generate the centroid
TeCoord2D centroid = TeFindCentroid(poly);

//creating some points
TePointSet points;
TePoint point1(TeCoord2D(480000.38, 6667799.34));
point1.objectId ("point_1");
points.add (point1);

TePoint point2(TeCoord2D(483180.30, 6668191.90));
point2.objectId ("point_2");
```

```
points.add (point2);

TePoint point3(TeCoord2D(483180.30, 6660000.90));
point3.objectId("point_3");
points.add (point3);

TePointSet::iterator it = points.begin();
while(it != points.end())
{
        if (TeWithin((*it), poly))
                cout << The point " << (*it).objectId() << " is inside the polygon." << endl ;
        ++it;
}
```

The example above used the topological functions `TeGeometryArea`, `TeLength`, `TeFindCentroid` and `TeWithin` and a precision de by `TePrecision` class.

Example 11.2 -Calculate a union and the intersection between two polygons.

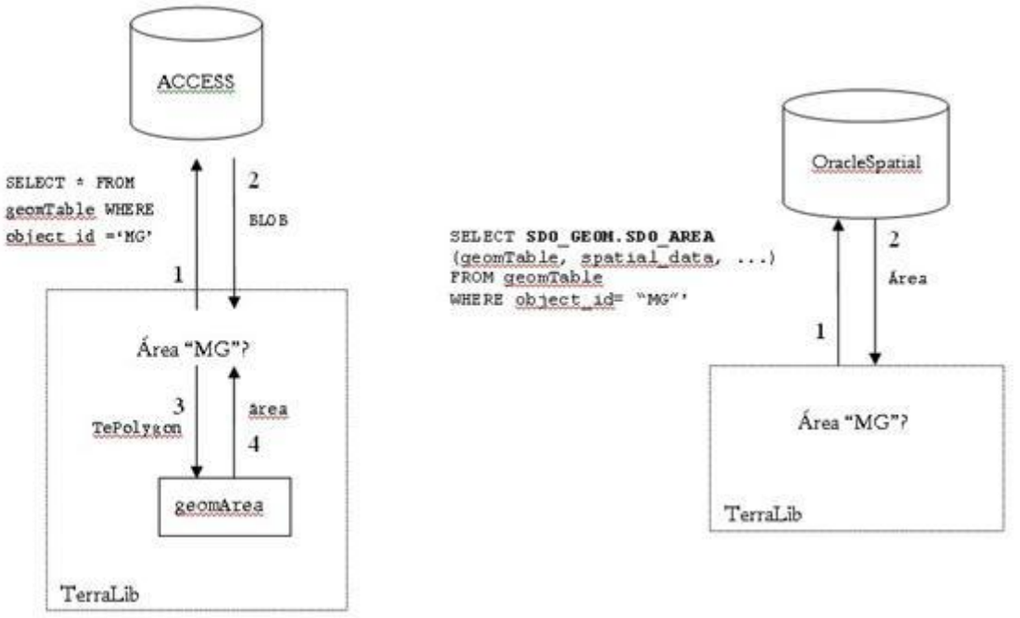
```
// create polygon 1
TeLine2D line1;
line1.add(TeCoord2D(480275.38, 6667799.34));
line1.add(TeCoord2D(483337.32, 6666464.65));
line1.add(TeCoord2D(486006.70, 6668898.50));
line1.add(TeCoord2D(484593.50, 6671410.86));
line1.add(TeCoord2D(481688.58, 6672431.51));
line1.add(TeCoord2D(480275.38, 6667799.34));
TeLinearRing ring(line1);
TePolygon poly1;
poly1.add(ring);

// create polygon 2
TeLine2D line2;
line2.add(TeCoord2D(484000.00, 6668000.00));
line2.add(TeCoord2D(484000.00, 6670000.00));
line2.add(TeCoord2D(485000.00, 6670000.00));
line2.add(TeCoord2D(485000.00, 6668000.00));
line2.add(TeCoord2D(484000.00, 6668000.00));
TeLinearRing ring2(line2);
TePolygon poly2;
poly2.add(ring2);

TePolygonSet polSet1;
polSet1.add(poly1);
TePolygonSet polSet2;
polSet2.add(poly2);
TePolygonSet polSetResult;

// calculate the union of the polygons
bool res = TeOVERLAY::TeUnion(polSet1, polSet2, polSetResult);
polSetResult.clear();
// calculate the intersection between the polygons
res = TeOVERLAY::TeIntersection(polSet1, polSet2, polSetResult);
```

11.1. Generic API to execute spatial operations

The spatial operations were also implemented as a set of methods of the class `TeDatabase` forming a generic application interface for s operations. This API is generic because, being used through the class `TeDatabase`, the methods can be used through with the same signatu drivers that do not have spatial extensions as well for those that do not have. The drivers with spatial extensions the API uses the extens implement the operations. For drivers that do not have spatial extension the API used the basic operations in memory (as shown in the exa above), after the retrieving of the geometries from the database.

Figure 11.1 shows the difference of the execution of a spatial operation, calculus of area, when executed in a DBMS without spatial extensio example ACCESS) or in a DBMS with spatial extension (for example, Oracle Spatial).

A complete description of the TerraLib Generic Database API can be found in Ferreira (2003).



(a) Database without spatial extension (b) Database with spatial extension

Figure 11.1 – Representation of the calculus of area operation (adapted from Ferreira, 2003).

The Spatial Operations API can be used when the geographic data are stored in a TerraLib database. The following examples show how to use

Example 11.3 - Retrieve the adjacent states (`TeTOUCHES`), the rivers that cross (`TeCROSSES`) and the cities that are within (`TeWITHIN`) a state the identifier 23.

```
// retrieve the layers from the database
TeLayer* states = new TeLayer("states");     // states: polygons
db->loadLayer(states);
TeLayer* rivers = new TeLayer("rivers");     // rivers: lines
db->loadLayer(rivers);
TeLayer* cities = new TeLayer("cities");     // cities: points
db->loadLayer(cities);

vector<string> objsOut; // a vector to store the identification of the resulting objects
vector<string> objsIn;  // a vector to store the identification of the object being consulted
objsIn.push_back("23");

// defines a default precision according to the projection of the data
TePrecision::instance().setPrecision(TeGetPrecision(states->layer()->projection()));

// retrieves the states that touch the state 23
bool res = db->spatialRelation(states->tableName(TePOLYGONS), TePOLYGONS, objsIn, objsOut, TeTOUCHES);
if (res)
{
        cout << "States that touch the state \"23\": \n";
        unsigned int i;
        for (i=0; i<objsOut.size(); ++i)
                cout << "State " << objsOut[i] << endl;
}

// retrieves the rivers that touch the state 23
res = db->spatialRelation(states->tableName(TePOLYGONS), TePOLYGONS, objsIn,
                rivers->tableName(TeLINES), TeLINES, objsOut, TeCROSSES);
if(res)
{
        cout << "\nRivers that cross the state \"23\": \n";
```

```
        unsigned int i;
        for (i=0; i<objsOut.size(); i++)
                cout << "River " << objsOut[i] << endl;
}

// retrieves the cities that are inside the state 23
res = db->spatialRelation(states->tableName(TePOLYGONS), TePOLYGONS,
                objsIn, cities->tableName(TePOINTS), TePOINTS, objsOut, TeWITHIN);
if (res)
{
        cout << "\nCities that are inside the state \"23\": \n";
        unsigned int i;
        for (i=0; i<objsOut.size(); i++)
                cout << "City " << objsOut[i] << endl;
}

db->close();
return 0;
```

It can be observed that the topological relation is a parameter for the functions. The possible topological relations are defined in th
`TeDataTypes` and are: `TeDISJOINT`, `TeTOUCHES`, `TeCROSSES`, `TeWITHIN`, `TeOVERLAPS`, `TeCONTAINS`, `TeINTERSECTS`, `TeEQU
`TeCOVERS`, `TeCOVEREDBY`.

Example 11.4 - Retrieving the crime events that occurred within a buffer of 500 meters around the frontier of the neighborhood with identifier "6

```
//retrieves the layer with the neighborhoods"
TeLayer* bairros = new TeLayer("BairrosPoA");
db->loadLayer(bairros);
// retrieves the layers with the crime occurrences
TeLayer* events = new TeLayer("OcorrenciasPoA");
db->loadLayer(events);



vector<string> objsOut;    // to store the identifier of the resulting occurrences
vector<string> objsIn;      // the identifier of the object to queried
objsIn.push_back("60");

TePrecision::instance().setPrecision(TeGetPrecision(bairros->projection()));

// generates the distance buffer
TePolygonSet bufferPol;
if (!db->buffer(bairros->tableName(TePOLYGONS), TePOLYGONS, objsIn, bufferPol, 500))
{
        cout << "Error generating the buffer!" << endl;
        return 1;
}
// retrieves the occurrences within the buffer
if(!db->spatialRelation(events->tableName(TePOINTS), TePOINTS, &bufferPol, objsOut, TeWITHIN))
{
        cout << "Error retrieving the events!" << endl;
        return 1;
}
>cout << " Crime occurrences within the buffer:" << endl << endl;

for(unsigned int i=0; i<objsOut.size(); i++)
        cout << "      " << objsOut[i] << endl;

db->close();
return 0;
```

The resulting of buffer generation operation is a set of polygons with two rings, the first is the external buffer and second one the internal buffer.

The Example 11.5 shows the `zonal` and `mask` operations of the API. The zonal operation calculates a set of statistical measures (for exa sum, maximum and minimum values, counting, mean, variance, etc.) over the raster data region that is inside the polygon that represents the r of interest. The result is returned in a data structure provided by TerraLib.

The mask operation clips the raster data creating a new layer in the database. The operation can retrieve the area that is internal or external polygon that represents the region of interest.

Example 11.5 - Clipping a raster data from a mask given by a polygon.

```
//connect to the database
// initializes the raster decoding mechanism
TeInitRasterDecoders();

//loads the layer with the raster data
TeLayer* layerBrasilia =  new TeLayer("Brasilia_RGB",db);
string rasterTable =  layerBrasilia->tableName(TeRASTER);

//retrieves the polygon that will be used as the clipping mask
TeLayer* layerPolBrasilia = new TeLayer("BrasiliaPol", db);
TePolygonSet ps;
layerPolBrasilia->loadGeometrySet("0",ps);

//clips the raster data using the internal side of the mask, creating a new layer.
db->mask (layerBrasilia->tableName(TeRASTER), ps[0], "BrasiliaRecortadoIn", TeBoxPixelIn);

//clips the raster data using the external side of the mask, creating a new layer.
db->mask (rasterTable, poly, "BrasiliaRecortadoOut", TeBoxPixelOut);

//zonal operation: generates a series of measures within the mask
TeStatisticsDimensionVect result;
db->zonal(rasterTable, poly, result);
```

Back to top.

12. Dealing with spatio-temporal data

The object attributes and geometries or geographic element might vary along time, that is, a geographic element can contain one or se instances in time. Some examples of spatio-temporal data are:

- ***Events***: independent occurrences in space and time, that have a temporal label that says when the occurrence happened. For exa crimes that occur in a city on a certain date and location. Each occurrence has a unique identification in the database;
- ***Mutable objects***: objects that have a geometry that do not change along the time, but that has a set of descriptive attributes that change. For example, the data used by dynamic models based on cell spaces, or fixed measurement stations;
- ***Moving objects***: data set where the attributes and the geometries of the objects can change along the time. For example, the data relat the evolution of parcels of a city.

In a TerraLib database the data are registered semantically as belonging to one of the classes shown above, accordingly to their tem characteristics. This allows those functionalities in terms of visualization, retrieving and grouping can be built. So when an attribute table is imp to the database, some metadata information are registered in the table **`te_layer_table`**. Besides the static and external tables (desc previously) the other types of tables currently supported are:

- ***Events***: it has to register the name of attribute that uniquely identifies each object (and that links it to a geometry) and the fields that co the time stamp associated to each event;
- ***Variable Attributes***: it is related to a mutable objects. It has to register the name of attribute that uniquely identifies each object (an links it to a geometry), the name of the attribute that uniquely identifies each instance in time of the object and the fields that contains the stamp associated to each instance.

The model to store moving objects is still under development in TerraLib.

Example 12.1 - Use the TerraView software to import the file with the crime occurrences and observe the records generated in the **`te_layer_table`**.

12.1. Data structures to represent spatio-temporal data

The TerraLib classes to represent spatio-temporal data are: `TeSTInstance, TeSTElement, TeSTElementSet. TeSTInstance` repre an instance of an object, or a version in time of a geographical object. An instance has a unique identifier, the identifier of the object t represents and the validity time interval (class `TeTimeInterval`), its geometries (class `TeMultiGeometry`) and its attributes (`TePropertyVector`). All the instances of the same geographical object are grouped in the class `TeSTElement`.

The TeSTElementSet class is an instances memory container (TeSTInstance) of a Layer of Theme stored in a TerraLib database. Ther the TeSTElementSet can be associated to a Layer or to a Theme. If it is associated to a Layer, it will have all instances of all objects or eler that belong to this Layer. If it is associated to a Theme, it will just contain the instances of elements of a Layer that fulfill the attributes restric spatial or temporal, defined on this Layer. Then, this container must be used as input for spatial analysis algorithms. The container pro mechanisms for traversing and retrieval of attributes and geometries of an instance on time of an object.

Besides, the TeSTElementSet can be fulfilled from a shapefile file, as shown below.

Example 12.2 - Retrieving all elements of a shapefile file using the STElementSet.

```
// shape file name with its path
string filename = "../data/EstadosBrasil.shp";

TeSTElementSet  steSet;
// fills the element set from shape file
if (!TeSTOSetBuildSHP(steSet,filename))
{
        cout << "Erro\n";
        return;
}
// prints the number of elements of the element set
cout << "Number of elements: " << steSet.numElements() << endl;
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end())
{
        TeSTInstance st = (*it);
        // returns the attributes
        TePropertyVector vectp = st.getPropertyVector();
        cout << "Id: " << st.objectId() << " ------------ " << endl;
        for (unsigned int i=0; i<vectp.size(); i++)
        {
                cout << vectp[i].attr_.rep_.name_ << "  =  ";
                cout << vectp[i].value_ << endl;
        }
        cout << endl;
        ++it;
}
```

Example 12.3 - Retrieving all elements or objects of a Layer, stored in a TerraLib database, using the TeSTElementSet. In this case TeSTElementSet is fulfilled with some attributes of the Layer.

```
// Connects to database ...
// Loads a layer named BairrosPoA
TeLayer* estados =  new TeLayer("EstadosBrasil");
db_->loadLayer(estados);

//Inits querier strategies
TeInitQuerierStrategies();

// Creates a elementSet from the layer "estados"
TeSTElementSet steSet (estados);

// Defines what attributes will be kept in the elementSet.
// The attribute name must be in the format
// "table_name.attribute_name"
vector<string> attrs;
attrs.push_back("EstadosBrasil.NOME_UF");
attrs.push_back("EstadosBrasil.CAPITAL");

// Fills the elementSet without geometries and with the
// attributes NOME_UF and CAPITAL
bool loadGeometries = false;
```

```
bool loadAllAttributes = false;
if(!TeSTOSetBuildDB(&steSet, loadGeometries, loadAllAttributes, attrs))
{
        cout << "Error! " << endl;
        cout << endl << "Press Enter\n";
        getchar();
        return 1;
}

// Prints the number of elements of the elementSet
cout << "Number of elements: " << steSet.numElements() << endl;

TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end())
{
        TeSTInstance st = (*it);
        // returns the attributes
        TePropertyVector vectp = st.getPropertyVector();

        cout << "Id: " << st.objectId() << " ---------------- " << endl;
        for (unsigned int i=0; i<vectp.size(); i++)
        {
                cout << vectp[i].attr_.rep_.name_ << "  =  ";
                cout << vectp[i].value_ << endl;
        }
        cout << endl;
        ++it;
}
db->close();
```

Example 12.4 Creating a Theme from a Layer, with spatial and attributes restrictions, and retrieving all its objects and elements usin STElementSet. In this example, the Theme is created in memory, that is, it is not "stored in a TerraLib database and will have all crime occurr of the type "threaten" (attribute restriction) occurring at Santo Antonio district (spatial restriction).

```
// Connects to TerraLib database...
// Loads a layer named BairrosPoA
TeLayer* bairrosPA =  new TeLayer("BairrosPoA");
db_->loadLayer(bairrosPA))

// Loads the geometry of Santo Antonio district (id = 48)
TePolygonSet ps;
bairrosPA->loadGeometrySet("48", ps );

// Loads a layer named OcorrenciaPoA
TeLayer* OcorrenciaPoA =  new TeLayer("OcorrenciasPoA");
db->loadLayer(OcorrenciaPoA);

// Create a theme with restrictions from Ocorrencias layer, in memory
TeTheme* Ocorrencias = new TeTheme("Ocorrencias", OcorrenciaPoA);
TeAttrTableVector attrTables;
OcorrenciaPoA->getAttrTables(attrTables);
Ocorrencias->setAttTables(attrTables);

//spatial restriction: within Santo Antonio district
Ocorrencias->setSpatialRest (&ps, TePOINTS, TeWITHIN);
//attribute restriction: type "ameaça"
Ocorrencias->attributeRest(" EVENTO = 'Ameaça' ");

// Inits querier strategies
TeInitQuerierStrategies();

// Creates a elementSet from theme
```

```
TeSTElementSet  steSet(Ocorrencias);

// Builds the elementSet with geometries and all attributes
bool loadGeometries = true;
bool loadAllAttributes = true;
if(!TeSTOSetBuildDB(&steSet, loadGeometries, loadAllAttributes))
{
        cout << "Error! " << endl;
        cout << endl << "Press Enter\n";
        getchar();
        return 1;
}
// Shows how many elements the elementSet has
cout << "Number of elements: " <<  steSet.numElements() << endl;

TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end())
{
        TeSTInstance st = (*it);
        //Gets attribute value
        string event;
        st.getPropertyValue("EVENTO", event);
        cout << " Object Identifier :  " << st.objectId() << endl;
        cout << " Event     :  " << event  << endl ;

        //Gets geometry
        if(st.hasPoints())
        {
                TePointSet pset;
                st.getGeometry (pset);
                cout<< "     point id: "<< pset[0].objectId ()        << endl;
                for(unsigned int j=0; j<pset.size (); ++j)
                {
                        string point =  Te2String(pset[j].location().x()) + ";" + Te2String(pset[j].location().y()
                        cout<< "          point: " << " ("+ point + ") "<< endl;
                }
        }
        ++it;
}
db->close();
delete (Ocorrencias);
```

12.2. A querier mechanism to retrieve spatio-temporal data: `TeQuerier`

The `TeQuerier` class implements a mechanism for retrieving and traversing instances (`TeSTInstance`) of a certain Layer or Theme responsible for accessing a TerraLib database, applying the restrictions defined by the user returning the instances that satisfy each one of restrictions. The `TeSTElementSet` class, shown on item 12.1, is a container storing in memory all instances of elements of a Theme or Layer `TeQuerier` class is internally used to fulfill the TeSTElementSet class. The `TeQuerier` class should be used for storing returned instanc specific data structures, different from the `TeSTElementSet` class, or when there is a need for traversing instances without storing them in me The `TeQuerier` class has a set of parameters that defines what will be fulfilled on the retrieved instances. An instance might contain: all attril some of them or none, all geometries or none and the valid time if the Theme or Layer are temporal. The examples 12.5 and 12.6 will show h retrieve instances of a Layer using `TeQuerier`.

Example 12.5 - Retrieving all instances of a Layer, with all its attributes but no geometries.

```
// Opens a connection to a TerraLib database
// Load the layer
TeLayer* armadilhas = new TeLayer("LAYER_ARMADILHAS");
db_->loadLayer(armadilhas);

bool loadAllAttributes = true;
bool loadGeometries = false;
```

```
//Init querier strategies
TeInitQuerierStrategies();

// Set querier parameters - load all attributes and
// no geometries of the layer "armadilhas"
TeQuerierParams querierParams(loadGeometries, loadAllAttributes);
querierParams.setParams(armadilhas);
TeQuerier  querier(querierParams);

// Load instances from layer based in the querier parameters
querier.loadInstances();

// Return a list of the loaded attributes
TeAttributeList attrList = querier.getAttrList();
cout << " Loaded Attributes  -------------------  " << endl;

// Plot the attribute names
for(unsigned int i=0; i<attrList.size(); ++i)
cout << attrList[i].rep_.name_ << endl;
cout << endl;

// Traverse all the instances
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
        cout << " Object: " << sti.objectId() << endl << endl;

        // Plot each attribute, its name and value
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
}
db_->close ();
return;
```

Example 12.6 - Retrieving all instances of a Layer, with geometries and only two attributes

```
// Opens a connection to a TerraLib database
// Load the layer "armadilhas"

bool loadGeometries = true;
vector<string> attributes;
attributes.push_back ("ARMADILHAS.COD_SITIO");
attributes.push_back ("ARMADILHAS.BAIRRO");

//Init querier strategies
TeInitQuerierStrategies();

// Set querier parameters - load only the attributes "COD_SITIO"
// and "BAIRRO" and geometries of the layer "armadilhas"
TeQuerierParams querierParams(loadGeometries, attributes);
querierParams.setParams(armadilhas);
TeQuerier  querier(querierParams);

// Load instances from layer based in the querier parameters
querier.loadInstances();
```

```
// Traverse all the instances
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
        cout << " Object: " << sti.objectId() <<  endl << endl;

        // Plot each attribute, its name and value
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }

        // Plot geometry
        if(sti.hasPoints())
        {
                TePointSet pointSet;
                sti.getGeometry(pointSet);

                for(unsigned int j=0; j<pointSet.size(); ++j)
                        cout << " Point : ( "  << pointSet[j].location().x()
                 <<     "," << pointSet[j].location().y() << " )";
        }
        cout << endl << endl;
}
cout << " End " << endl;
getchar();
db_->close();
return;
```

The following examples show how to retrieve instances from a Theme using `TeQuerier`. In this case, the `TeQuerier` takes into account all T restrictions (attributes, temporal and spatial).

Example 12.7 Retrieving all instances of a Theme, with all attributes and geometries. In this example, the districts have two geo representations - point and polygon.

```
// Connect to the database...
//Load the theme "DistritosSaoPaulo"
TeTheme* bairros = new TeTheme("DistritosSaoPaulo");
db_->loadTheme(bairros);

// Init the querier mechanism
TeInitQuerierStrategies();

// All attributes and geometries
bool loadGeometries = true;
bool loadAllAttributes = true;

// Seta os parametros do querier - carrega todos os atributos e
// geometrias
TeQuerierParams querierParams(loadGeometries, loadAllAttributes);
querierParams.setParams(bairros);

TeQuerier  querier(querierParams);

// Load instances from layer based in the querier parameters
if(!querier.loadInstances())
        return 1;
// Traverse all the instances
TeSTInstance sti;
```

```
while(querier.fetchInstance(sti))
{
        cout << " Objeto: " << sti.objectId() << " ---- " << endl << endl;

        // Plot each attribute, its name and value
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
        //Get geometries
        if(sti.hasPolygons())
        {
                TePolygonSet polSet;
                sti.getGeometry(polSet);
                for(unsigned int i=0; i<polSet.size(); ++i)
                {
                        TeCoord2D centroid = TeFindCentroid(polSet[i]);
                        string p = "( "+ Te2String(centroid.x(), 7) +", "+
                  Te2String(centroid.y(), 7) +")";
                        cout << " Centroide do Poligono : " << p << endl;
                }
        }
        if(sti.hasPoints())
        {
                TePointSet ponSet;
                sti.getGeometry(ponSet);
                for(unsigned int i=0; i<ponSet.size(); ++i)
                {
                        string p = "( "+ Te2String(ponSet[i].location().x())
                     +", "+  Te2String(ponSet[i].location().y()) +")";
                        cout << " Ponto : " << p << endl;
                }
        }
        cout << endl << endl;
}
delete(bairros);
return 1;
```

The examples 12.8 and 12.9 show how to use the `TeQuerier` spatial restriction. When a spatial restriction is given to the `TeQuerier` class the instances satisfying this restriction are retrieved. In the example 12.8, the spatial restriction is defined by a rectangle or `TeBox`. In the exa 12.9, the spatial restriction is defined by the geometries of another Theme.

Example 12.8 - Given a rectangle or box (`TeBox`), retrieve all instances of a Theme that are inside this rectangle.

```
// Connect to the database...
// Load the Layer OcorrenciasBH
// Init the querier mechanism
TeInitQuerierStrategies();

// All attributes and geometries
bool loadGeometries = false;
bool loadAllAttributes = true;

// Define the querier parameters
TeQuerierParams querierParams(loadGeometries, loadAllAttributes);
querierParams.setParams(ocorrencias);

// Define the spatial restriction
TeBox box(609033.62, 794723.35, 613455.34, 800417.99);
```

```
querierParams.setSpatialRest(box, TeWITHIN);

TeQuerier  querier(querierParams);

//Load instances from layer based in the querier parameters
querier.loadInstances();

// Traverse all the instances
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
        cout << " Object: " << sti.objectId() << " ----- " << endl;
        // Plot each attribute, its name and value
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
        cout << endl << endl;
}
db_->close ();
```

Example 12.9 - For each element of a Theme X, retrieve all instance of another Theme Y that are inside the geometry of this element. I example, there will be two `TeQuerier` classes: one for the Theme Districts and another for the Theme Traps. All traps inside of each district v retrieved.

```
// Connect to the database...
// Loads the theme "OcorrenciasBH"
// Loads the theme "BairrosBH"

//Init the querier mechanism
TeInitQuerierStrategies();

// ------------ querier for the theme "bairros"
bool loadGeometries = true;
vector<string> attrs;
attrs.push_back ("Bairrobh.nobai");

TeQuerierParams querParamsBair(loadGeometries, attrs);
querParamsBair.setParams(bairros);

TeQuerier  querBairros(querParamsBair);
if(!querBairros.loadInstances())
{
        cout << " There is no data!!! " << endl;
        return 1;
}

// Loads each element and pass to query as a spatial restriction
TeSTInstance bairro;
while(querBairros.fetchInstance (bairro))
{
        string name;
        bairro.getPropertyValue(name, 0);
        cout << " Bairro: " << name << endl;

        TePolygonSet setP;
        if(!bairro.getGeometry (setP))  // access the polygon set
                continue;
```

```
        // ------------ querier for the theme ocorrências
        loadGeometries = false;
        vector<string>attributes;
        attributes.push_back ("ocorrenciasbh.Nrbo");
        TeQuerierParams querParamsOcr(loadGeometries, attributes);
        querParamsOcr.setParams(ocorrencias);

        // spatial restriction
        querParamsOcr.setSpatialRest(&setP);

        TeQuerier  querOcorrencias(querParamsOcr);
        if(!querOcorrencias.loadInstances())
        {
                cout << " There is no data!!! " << endl;
                continue;
        }
        TeSTInstance sto;
        while(querOcorrencias.fetchInstance(sto))
        {
                // Plot each attribute, its name and value
                TePropertyVector vec = sto.getPropertyVector();
                for(unsigned int i=0; i<vec.size(); ++i)
                {
                        string  attrName = vec[i].attr_.rep_.name_;
                        string  attrValue = vec[i].value_;
                                cout << attrName  << " : " << attrValue << endl;
                }
                // Mostra a data
                cout << endl << " Data : " << sto.getInitialDateTime();
            cout << endl;
        }
}
db_->close ();
```

The `TeQuerier` class can group instances by element, by spatial restriction or by "chronon" (when the Theme is temporal). The Example groups instances by element while the Example 12.11 groups all instances satisfying a spatial restriction. The Example 12.12 groups all instanc "chronon" and by element. These examples are based on a data set originated from a epidemiological experiment. It deals with a data set t composed by a set of traps to collect mosquito eggs. For each trap (the object) a new egg counting is annotated in the database in a week basi

Example 12.10 - Grouping instances by element - all collection (instances) by trap (element). That is, for each trap, the `TeQuerier` class sur the number of collected eggs on this trap.

```
// Connect to the database...
//Loads the theme "coletas" (the collects of eggs in the traps)

//Init the querier mechanism
TeInitQuerierStrategies();

// Selects a function to group the countings in each trap
bool loadGeometries = false;
TeGroupingAttr attributes;

pair<TeAttributeRep,TeStatisticType> attr1(TeAttributeRep("Coletas.NRO_OVOS_PAL1"), TeSUM);
attributes.insert(attr1);

pair<TeAttributeRep, TeStatisticType> attr2(TeAttributeRep("Coletas.NRO_OVOS_PAL2"), TeSUM);
attributes.insert(attr2);

pair<TeAttributeRep, TeStatisticType> attr3(TeAttributeRep("Coletas.NRO_OVOS_PAL3"), TeSUM);
attributes.insert(attr3);

pair<TeAttributeRep, TeStatisticType> attr4(TeAttributeRep("Coletas.NRO_OVOS"), TeSUM);
```

```
attributes.insert(attr4);

// Defines the querier parameters: loads only the grouped attributes
TeQuerierParams querierParams(loadGeometries, attributes);
querierParams.setParams(coletas);

TeQuerier  querier(querierParams);

// Loads the data from the layer based on the querier parameters
if(!querier.loadInstances())
{
        db_->close ();
        return;
}

// Traversal all instances
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
        cout << " Object: " << sti.objectId()<< endl;

        // Plot each attribute, its name and value
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
        cout << endl << endl;
}
db_->close();
return;
```

Example 12.11 - Grouping all instances satisfying a spatial restriction. This example groups all the collected data by the district that contains it is, the spatial restriction will be defined by each district of the Districts Theme and the `TeQuerier` class sums up the number of collected eg each district

```
// Connects to the database and load the themes...
TeTheme* coletas = new TeTheme("Coletas");
db_->loadTheme(coletas);
TeTheme* bairros = new TeTheme("ibge_bairros");
db_->loadTheme(bairros);

TeInitQuerierStrategies();

// statistics about the counting
TeGroupingAttr attributes;

pair<TeAttributeRep, TeStatisticType> attr1(TeAttributeRep("Coletas.NRO_OVOS_PAL1"), TeSUM);
attributes.insert(attr1);

pair<TeAttributeRep, TeStatisticType> attr2(TeAttributeRep("Coletas.NRO_OVOS_PAL2"), TeSUM);
attributes.insert(attr2);

pair<TeAttributeRep, TeStatisticType> attr3(TeAttributeRep("Coletas.NRO_OVOS_PAL3"), TeSUM);
attributes.insert(attr3);

pair<TeAttributeRep, TeStatisticType> attr4(TeAttributeRep("Coletas.NRO_OVOS"), TeSUM);
attributes.insert(attr4);

// ------------ querier for the theme "bairros" (the districts)
```

```
bool loadGeometries = true;
vector<string> attrs;
attrs.push_back ("BAIRRO2000_REC.NOME");

TeQuerierParams querParamsBair(loadGeometries, attrs);
querParamsBair.setParams(bairros);

TeQuerier  querBairros(querParamsBair);
if(!querBairros.loadInstances())
{
        cout << " there is no data!!! " << endl;
        return;
}

// Loads each district passing its geometry as a spatial restriction
TeSTInstance bairro;
while(querBairros.fetchInstance (bairro))
{
        string name;
        bairro.getPropertyValue(name, 0);
        cout << " Bairro: " << name << endl;

        TePolygonSet setP;
        if(!bairro.getGeometry (setP))  // Access o polygon set
                continue;

        // ------------ querier for the theme "coletas"
        loadGeometries = false;

        TeQuerierParams querParamsCol(loadGeometries, attributes);
        querParamsCol.setParams(coletas);

        // spatial restriction
        querParamsCol.setSpatialRest(&setP);
        TeQuerier  querColetas(querParamsCol);

        if(!querColetas.loadInstances())
        {
                cout << " There is no data!!! " << endl;
                continue;
        }
        TeSTInstance coleta;
        while(querColetas.fetchInstance(coleta))
        {
                // Plot each attribute, its name and value
                TePropertyVector vec = coleta.getPropertyVector();
                for(unsigned int i=0; i<vec.size(); ++i)
                {
                        string  attrName = vec[i].attr_.rep_.name_;
                        string  attrValue = vec[i].value_;
                        cout << attrName  << " : " << attrValue << endl;
                }
        }
}
db_->close ();
return;
```

Example 12.12 - Grouping instances by "chronon" and by element. This example groups all collection by month and by trap.

```
// Connects to the database and load the themes...>
TeInitQuerierStrategies();
```

```
// The grouping attribute
bool loadGeometries = false;
TeGroupingAttr attributes;

pair<TeAttributeRep, TeStatisticType> attr1(TeAttributeRep("Coletas.NRO_OVOS_PAL1"), TeSUM);
attributes.insert(attr1);

pair<TeAttributeRep, TeStatisticType> attr2(TeAttributeRep("Coletas.NRO_OVOS_PAL2"), TeSUM);
attributes.insert(attr2);

pair<TeAttributeRep, TeStatisticType> attr3(TeAttributeRep("Coletas.NRO_OVOS_PAL3"), TeSUM);
attributes.insert(attr3);

pair<TeAttributeRep, TeStatisticType> attr4(TeAttributeRep("Coletas.NRO_OVOS"), TeSUM);
attributes.insert(attr4);

// Loads only the grouping attributes
TeQuerierParams querierParams(loadGeometries, attributes);
querierParams.setParams(coletas, TeMONTH);

TeQuerier  querier(querierParams);
// Retrieves the number of time frames when seeing the data by TeMONTH
int numTimeFrames = querier.getNumTimeFrames();

// Loads all the instances in each time frame
for(int frame=0; frame < numTimeFrames; ++frame)
{
        TeTSEntry ts;
        querier.getTSEntry(ts, frame);

        cout << " Time frame: " << Te2String(frame) <<  endl << endl;
        string initialDate = ts.time_.getInitialDateTime("DDsMMsYYYY");
        string finalDate = ts.time_.getFinalDateTime("DDsMMsYYYY");

        cout << " Time interval: " << initialDate << " to " << finalDate;
        if(!querier.loadInstances(frame))
                continue;

        TeSTInstance sti;
        while(querier.fetchInstance(sti))
        {
                cout << " Object " << sti.objectId() << endl;
                TePropertyVector vec = sti.getPropertyVector();
                for(unsigned int i=0; i<vec.size(); ++i)
                {
                        string  attrName = vec[i].attr_.rep_.name_;
                        string  attrValue = vec[i].value_;
                        cout << attrName  << " : " << attrValue << endl;
                }
        }
        cout << endl << endl;
}
db_->close ();
return;
```

Back to top.

13. Analysis and spatial statistics

13.1. Proximity matrix

The Proximity Matrix is a very important concept for spatial analysis and spatial statistics. With TerraLib it is possible to generate a proximity m according to different criteria such as "adjacency" and minimum distance. It also possible to weight and slice the matrix according to an attr The next examples show how to build a proximity matrix using a `TeSTElementSet`, following different criteria.

Example 13.1 -Creating a proximity matrix according to the minimum distance criterion traversing each objects´ neighbors. The minimum dis criterion considers two objects neighbors when the distance between them is less than a pre-defined distance. For this example, this distar 12000 meters.

```
// Connect to the database and loads the layer of districts...
TeLayer* DistritosSP = new TeLayer("DistritosSP");
db->loadLayer(DistritosSP);

// Init the querier strategies
TeInitQuerierStrategies();

// Create a empty STElementSet for the layer of districts
TeSTElementSet steSet(DistritosSP);

// Fills the STElementSet with the geometries
vector<string> attrs;
if(!TeSTOSetBuildDB(&steSet, true, false, attrs))
{
        cout << "Error! " << endl;
        return 1;
}

// Shows the number of elements in the set
cout << "Number of elements: " <<  steSet.numElements() << endl;

// Create a proximity matrix strategy
TeProxMatrixLocalDistanceStrategy<TeSTElementSet> sc_dist(&steSet,TePOLYGONS, 12000.00);
TeGeneralizedProxMatrix<TeSTElementSet> proxMat(&sc_dist);
proxMat.setCurrentConstructionStrategy(&sc_dist);

// Builds the proximity matrix
if(!proxMat.constructMatrix())
{
        cout << "Error creating the proximity matrix! " << endl;
        db->close ();
        return 1;
}

// Shows the neighbors of each district
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end())
{
        cout << " The neighbor of  "<< (*it).objectId() << "
                 are: " << endl;

        TeNeighboursMap neighbors = proxMat.getMapNeighbours((*it).objectId());
        TeNeighboursMap::iterator itN = neighbors.begin();
        while (itN != neighbors.end())      {
                 cout<< "         "<<(*itN).first << endl;
                 ++itN;
        }
        cout << endl;
        ++it;
}
db->close();
```

13.2. Spatial Statistics

Example 13.2 - Calculating spatial statistics using the proximity matrix generated following the adjacency criterion and recording the gene

statistics on a TerraLib database. The adjacency criterion says that two objects are neighbors if their borders touch each other (or themselves?)

```
// Connect to the database and loads the layer of districts...
TeLayer* DistritosSP =  new TeLayer("DistritosSP");
db->loadLayer(DistritosSP);

TeInitQuerierStrategies();

// Create a empty STElementSet for the layer of districts
TeSTElementSet steSet(DistritosSP);

// Fills the STElementSet only with the attribute "Pop91" to calculate the spatial statistics
vector<string> attrs;

attrs.push_back ("Distritos.Pop91");
if(!TeSTOSetBuildDB(&steSet, false, false, attrs)){
         cout << "Error! " << endl;
         return 1;
}

// Shows the number of elements in the set
cout << "Number of elements: " <<  steSet.numElements() << endl;

// Builds the proximity matrix
TeProxMatrixLocalAdjacencyStrategy sc_adj (&steSet,  TePOLYGONS);
TeGeneralizedProxMatrix<TeSTElementSet> proxMat(&sc_adj);

if (!proxMat.constructMatrix())
{
         cout << "Error building the proximity matrix! " << endl;
         return 1;
}

// Calculate a the global mean
double mean = TeFirstMoment (steSet.begin(), steSet.end(), 0);
cout << "Média Global " << mean << endl << endl;

// Calculate the deviation (Z) for each object of the STElementSet
if(!TeDeviation (steSet.begin(), steSet.end(), mean))
{
         cout << "Error calculating Z! " << endl;
         return 1;
}

// index for the attribute deviation (Z) in the STElementSet
int indexZ = 1;

// Calculates the local mean (WZ) for each object of the STElementSet
if(!TeLocalMean (&steSet, &proxMat, indexZ))
{
         cout << "Error calculating the Local Mean! " << endl;
         return 1;
}

// index for the attribute Local Mean (WZ) in the STElementSet
int indexWZ = 2;

// Shows the Z and WZ indexes generated for each object
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end()){
         TeSTInstance obj = (*it);
```

```
        //Gets attribute value
        string valZ, valWZ;
        obj.getPropertyValue(valZ, indexZ);
        obj.getPropertyValue(valWZ, indexWZ);
        cout << " Identificador do Objeto :  " << obj.objectId() << endl;
        cout << " Z               :  " << valZ   << endl;
        cout << " WZ      :  " << valWZ  << endl << endl;
        ++it;
}

// Save the attributes (Z e WZ) in the table DistritosSP
if(!TeUpdateDBFromSet (&steSet, "DistritosSP"))
{
        cout << "Error updating the database! " << endl;
        return 1;
}
db->close();
```

Back to top.

14. Bibliography

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. Ter 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. (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 Banc 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 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 INPE - Instituto Nacional de Pesquisas Espaciais, 2003. Dissertação de Mestrado, Computação Aplicada, 2003.

QUEIROZ, G. R.; FERREIRA, K. R., 2005. SGBD com extensões espaciais. In: CASANOVA, M. A.; CÂMARA, G.; DAVIS, C.; VINHA QUEIROZ, G. R., eds., **Bancos de Dados Geográficos**, v. 1: São José dos Campos, SP, Editora MundoGeo, p. 506.

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 Brasilei 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.

VINHAS, L.; FERREIRA, K. R.; , 2005. Descrição da TerraLib. In: CASANOVA, M. A.; CÂMARA, G.; DAVIS, C.; VINHAS, L.; QUEIROZ, G. R., **Bancos de Dados Geográficos**, v. 1: São José dos Campos, SP, Editora MundoGeo, p. 506.

[1] The term "spatial representation", in TerraLib context, is used with the same meaning as the term "geometry".

[2] The definition of this class is found in the file TeDatabase.h, in the directory kernel.

[3] The definition of this class is found in the file TeLayer.h, in the directory kernel.

[4] The definition of this class is found in the file TeProjection.h, in the directory kernel.

[5] The definition of this class is found in the file TeGeometry.h, in the directory kernel.

[6] The definition of this class is found in the file TeCoord2D.h, in the directory kernel.

[7] The definition of this class is found in the fileTeTable.h, in the directory kernel.

[8] O ponteiro must be copied because the next command getData can invalidate the pointer returned previously.

[9] The definition of this class is found in the TeRaster.h, in the directory kernel.

Back to top.