A Spatial Query Evaluator for GEO++

F. H. Ariës[1]

January 31, 1996

[1]Freddy Ariës, Lindeboomweg 7, 7135 KE Harreveld, afstudeernummer: 363

Contents

Chapter 1

Introduction

> *"Geographic Information Systems (GISs) are database systems that allow the manipulation, storage, retrieval and analysis of geographic data as well as the display of data in the form of maps."* [5]

Geographic Information Systems are comparable to normal DataBase Management Systems, but they usually have a different architecture which is better suited for handling the geographic (spatial) data. In a time that both the speed of the average computer and the graphical demands on the computers are increasing fast, setting up a GIS is becoming more and more feasible, and also increasingly popular. Some of the fields in which GIS techniques are used are cartography, demography, town/urban planning, navigation and natural resource management. Examples of questions (queries) that may be answered using GISs, include "What is the shortest route from Rome to Berlin", "Which rivers are within a radius of 10 km from Nijmegen", and "Which city with a population of at least 100,000 citizens is closest to Mount Rushmore".

One such Geographic Information System is the GEO++ system [1, 2], developed at the human factors laboratory of the TNO institute in the Netherlands. It is a graphical general purpose GIS frontend for the Postgres object oriented and relational DataBase Management System (DBMS). With GEO++, the user can compose queries and look at Postgres relations. Figure 1.1 shows how the the query "Give all cities in any country in Europe other than the Netherlands, that have a population of at least 250,000" can be expressed in GEO++. The results of the queries are graphically displayed on a map or shown in a tabular format in a scrollable window. Expected fields of application include electronic nautical charts and Command and Control (C2) systems. In the current version, the part of the GEO++ system is hardly more than an interface, because almost all data and query processing is done by Postgres, and Postgres databases should contain all data that the user wants to display.

When solving a question (query) with this Geographic Information System, it is sometimes a disadvantage that the response times of the GIS are too high to be able to determinate the correct parameter values in an iterative process.

People would like to be able to set the values of certain parameters in an interactive way, after which the system should show the results of the new parameter settings in a direct feedback. A general GIS ('general' referring to the classes of questions that could be answered) like GEO++ isn't fast enough for such tasks. This is inherent to the flexible setup of the GEO++ system. Universality and speed are often exchangable.

For the above mentioned reasons, the idea arose to implement a Spatial Query Evaluator (SQE) that should be able to quickly evaluate a certain class of (spatial) questions and that should directly visualize the effects of changes to the query parameters. It is clear that such a tool can only evaluate a part of the possible GIS questions.

It should also be obvious that for such a tool, the current Postgres system and the interface to it would have to be replaced by a new database level with fast response times for the above mentioned class of (spatial) questions, and a new and fast interface to this database. Other than the database level, though, it would be preferable if as few changes as possible would be made to the GEO++ system.

The purpose of this text is to document the author's attempts to implement such a Spatial Query Evaluator into the existing GEO++ Geographic Information System.

Chapter 2 will provide a bit more information on Geographic Information Systems in general, as well as a description of GEO++ and the currently used DataBase Management System (DBMS) POSTGRES.

In addition to an extensive study of spatial data structures and search algorithms, the project also involved study on the following support tools: the C++ programming language [6], which is the language in which GEO++ was written, and the ET++ class library [7, 8] that was used for the implementation of the GEO++ graphical system. As reading material on C++ is readily available, it will not be included in this document, and instead the assumption is made that the reader already has some basic knowledge of that programming language. However, the same can not be assumed for the as yet little known ET++ package, so a small introduction to ET++ will be

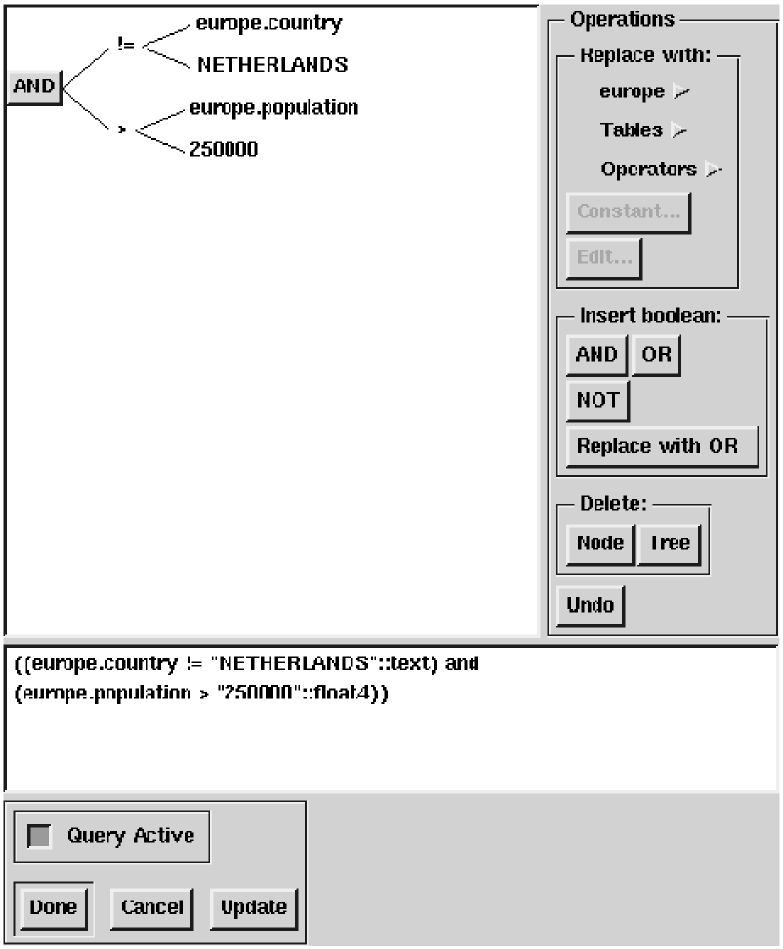


Figure 1.1: An example query

given in chapter 3.

The Spatial Query Evaluator, that is, the new database level, is described in chapter 4, and the changes and additions to the user interface are described in chapter 5. Finally, chapter 6 contains conclusions and remarks.

Chapter 2

The GEO++ System

2.1 Geographic Information Systems

Nowadays, systems that must be able to store and process large amounts of data are quite commonplace. This is also true for geographic or spatial data. Systems which are designed to handle spatial data are called Geographic Information Systems or GISs. In addition to the database methodology that is included in DataBase Management Systems, a GIS will also have to include techniques for cartography, demography, town/urban planning, and navigation. In addition, computer graphics and image processing are a common part of current GIS's.

As is the case in aspatial (non-geographic) database systems, an object in a spatial database usually belongs to a certain class. Examples of such classes are *road*, *city* and *lake*. On top of the aspatial or alphanumeric attributes in these classes, such as *name* and *population*, the objects in a GIS's database also contain spatial attributes, such as *location* and *height*; they contain coordinates which describe the object's position on a multi-dimensional map. These spatial attributes are based on the three geometric attribute types **point**, **line** and **region**. However, into which of the above categories an object is classified, is dependent on the *extent*, that is, the area of interest. The extent may vary depending on the situation. For example, the city of Amsterdam may be a point when viewed on the map of Europe, but it may be a region when viewed on the map of its home country, the Netherlands. Independent of these attributes, there are several *spatial relations* possible between objects. Objects may *intersect*, they may be *adjacent*, or they may *contain* other objects. These spatial relationships are used to answer the user's queries, such as the question "Which roads intersect with the A12 highway".

2.2 Definitions

Before we proceed, it seems best to first define what we mean with some terms that will be used frequently throughout the rest of this document, and some of which already have been used in the previous sections.

First of all, we have to distinguish between the types of data that a GIS must be able to store. These types are:

- aspatial; aspatial or non-geographic data is data based on attributes, which are usually (alpha)numeric. Examples of such data are the names of cities, the population of these cities, etc.

- spatial; spatial or geographic data is data dependent on geographical information, that is, object-oriented rather than attribute-based. It contains coordinates and/or other geographical properties. Examples include the distance between two cities, and the location of a certain city.

- graphical; graphical data is especially for display purposes, and usually in the form of maps. This data is closely related to the spatial attributes of spatial data, as the spatial data must be within the space of these maps. This type of data is fixed, and not for manipulation by the queries.

In addition to these types of data, there are two main types of search operations that will be used throughout this document:

- point queries; a search operation where, for a certain point in the data-space, you have to find all objects in which that point is contained.

- region queries; a search operation where, for a certain query region, you have to find all objects that intersect with that region.

Note that containing a point is basically the same thing as intersecting with a region that has zero area.

Closely related to the above described search operations, we use three different terms to describe the searches:

- intersection search; either type of search.

- point search; exact-match search of a point.

- containment search; search for all regions which are contained in a given query region (note that containment is possible only if two region areas intersect). Some data-structures, like R-trees (see chapter 3), do not directly support containment search, however, this is not a big problem, as containment search is basically just a special form of intersection search.

2.3 GIS Architectures

In this section, we will take a closer look at GIS architectures and the problems involved in their development, to get a better understanding of the architecture currently used in the GEO++-Postgres combination.

Of the currently existing commercial Geographic Information Systems, most are based on a relational DBMS (DataBase Management System). Examples of such systems are Oracle [38] and Ingres [39, 40]. However, these DBMS's aren't particularly suited to handle geographic data, as they lack special spatial types and operators. There are several possible solutions to this problem. Three alternative architectures that have been proposed to handle the problem, are the *"dual architecture"*, the *"layered architecture"* and the *"integrated architecture"*. In the following sections we will have a look at these architectures and discuss their main advantages and drawbacks. Note that some existing systems may contain elements from multiple of these architectures, and thus will be hard to classify.

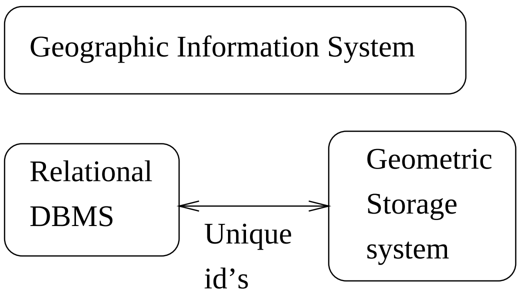


Figure 2.1: Dual GIS Architecture

2.3.1 Dual Architecture

One of the architectures that can be used to store geographic data is called "dual architecture" [2]. This kind of architecture (see figure 2.1), has separate parts for handling geographic (spatial) data and non-geographic data, and is the most common of these three architectures. In this system, the non-geographic data is stored in a relational DBMS, whereas the geographic data is stored in special subsystem. Some characteristic examples of GISs with a dual architecture are ARGIS 4GE [12], ARC/INFO [13, 14] and MGE [15]. The main advantages of the dual architecture are that the processing of spatial data can be done efficiently, and that for non-geographic data, the standard DBMS can be used. The disadvantages of this approach are that it is not an elegant solution, and moreover that overall performance suffers, because objects with both a spatial and an aspatial component have data in both subsystems, linked by a *unique identifier*, and when processing such an object, both subsystems have to be queried in order to obtain the answer.

Apart from that, the dual architecture can also put some severe restrictions on query optimization, as the spatial data may not satisfy the transaction semantics of the DBMS, as well as on the integrity constraints, because for example one may not be able to guarantee that an object that has been deleted from the relational DBMS, is also deleted from the spatial storage part of the system.

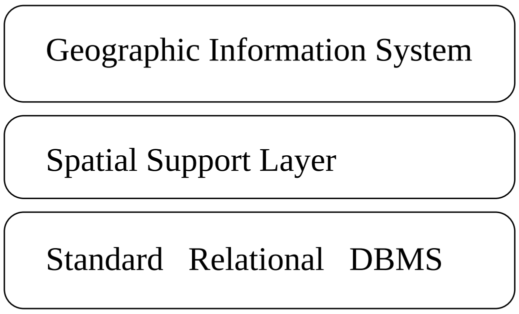


Figure 2.2: Layered GIS Architecture

2.3.2 Layered Architecture

The "layered architecture" (see figure 2.2), removes the most important cause of the problems with the dual architecture, namely the two different storage subsystems each with their own control programs. For that, spatial data is stored in a relational model. This means that the integrity constraints and the transaction semantics provided by the DBMS are maintained, but it also means that for storage of the spatial elements, it is necessary to break up the geographic data into non-geographic parts - eg. coordinates will be stored as simple numbers - and for retrieval these parts have to be joined back to the original objects. This mapping will usually make the overall performance slower and it will usually also mean that the system becomes more difficult to use (ie. the queries can become quite complicated). Up to a certain level, it is possible to shield the difficult mappings of spatial queries to relational queries from the user - eg. by allowing the user to enter the queries in the popular GEOQL [36] language and creating some clever translation mechanism that converts these geographic queries into standard queries in the *Structured Query Language* (SQL) [38, 41] in the upper layer - but the translations will almost always have an effect on query specifications that will remain visible to the user and that makes these queries harder to use. Typical examples of GISs with a layered architectures are SIRO-DBMS [16] from CSIRO, Australia, GEOVIEW [17] from the University of Edinburgh and System 9 [18] from Prime.

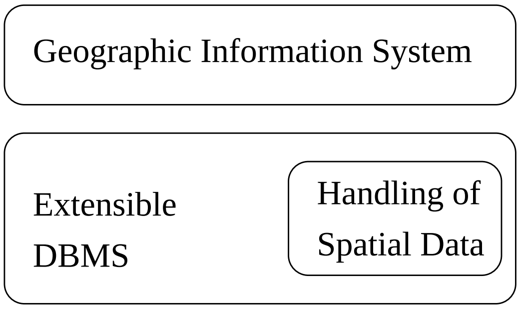


Figure 2.3: Integrated GIS Architecture

2.3.3 Integrated Architecture

To take away the disadvantages of the layered architecture, a third type of architecture can be used, called the "integrated architecture", see figure 2.3. This kind of architecture cannot simply be created with the usual relational DBMS's. Instead an extensible DBMS (preferably an object-oriented one) is required. The user then has to create his own spatial types, operators and sometimes his own access methods in this DBMS environment, which can be quite hard and clearly is a disadvantage of this method. However, once these types have been created, they can be treated fully as part of the DBMS, and they have a high efficiency (ie. performance generally is quite good). In this way the user can now implement his data, where he can use the new spatial types as well as previously available types, which makes the implementation process very easy. Some examples of GISs with an integrated architecture are TIGRIS [19] from Intergraph and GéoTropics [20] from the University of Paris VI and IGN France. GEO++ in its current form also has this integrated architecture, as it is based on the extensible DBMS Postgres. The next section will provide a more in-depth look at the GEO++ system and its interface to POSTGRES.

2.4 Postgres

As was briefly mentioned in the previous section, POSTGRES is an extensible relational and object oriented DataBase Management System (DBMS). It was developed at the University of Berkeley, California, under the supervision of Prof. Michael Stonebraker [4]. POSTGRES (POST inGRES) was developed as the successor of the relational DBMS Ingres [40], which was developed at the same institute. However, these systems are not compatible. Postgres uses its own query language, Postquel, which is an incompatible superset of Quel, the language used by Ingres.

Postgres is relational, but it also has some object oriented concepts integrated into the system, like inheritance. The four concepts on which Postgres is based, are: *classes, inheritance, types* and *functions*.

2.4.1 The different parts

The Postgres system is basically a collection of files and processes that operate on these files. In the files, the relations and the required data are stored. The Postgres environment consists of three main parts: the *Postmaster*, the Postgres *backend*, and *user applications*.

The Postmaster creates a Postgres backend for every user application that is started. It also takes care of most of the networking procedures, ie. the communication between this backend and the user application or *frontend*. The rest of the time, the Postmaster just waits quietly in the background. There can only be one Postmaster process running at the same time.

The Postgres backend, a process called *Postgres*, is the part of the program that does the real database management work, like receiving queries from the user application, and sending back the answers. For every application that communicates with the Postgres interface, another backend process is started by the Postmaster. Due to this approach, the Postgres system can handle an almost unlimited amount of databases at the same time.

A user application is any program that uses the `'libpq'` interface (see below) to communicate with the Postgres backend. The standard Postgres distribution already includes a number of user applications, including 'createdb' (a database creation program), 'destroydb' (a database deletion program), and the 'monitor' (a terminal monitor). The monitor provides a simple alphanumeric user-interface for Postgres. Through it people can formulate, edit and review queries in Postquel, and send them to the Postgres process.

2.4.2 Example

With the beforementioned user commands, a typical session could look like this:

```
% createdb foo
% monitor foo
```

This would create a new database named 'foo', and then call the monitor on that newly created database. '%' denotes the standard Unix shell prompt. Now, in the monitor, you could enter:

```
* create EMP (name=char[12], salary=float8, age=int4, dept=
  char[12])
* \g
```

This will create a new class in the foo database, and define all the attributes that make up this class.

The '*' character denotes the monitor command prompt, and the '\g' command is required to tell the monitor to execute the previously entered lines.

```
* append EMP (name="Joe",salary=1400.0,age=40,dept="shoe")
* append EMP (name="sam",salary=1200.0,age=29,dept="toy")
* append EMP (name="Bill",salary=1600.0,age=36,dept="candy")
* \g
```

This will add 3 instances to the EMP class, one for each command.

```
* retrieve (EMP.all) where EMP.age<35 and EMP.salary>1000
* \g
```

This would return all information on employees younger than 35, who earn more than 1000. In this case, it would only return the data of the employee named "sam".

```
* delete EMP where EMP.salary > 0
* \g
```

Remove all employees that have positive salaries from the EMP class. As all employees have positive salaries, this leaves the EMP class empty.

```
* \q
```

Leave the monitor and return to the Unix shell. If you have the required priviliges - eg. if you are a system administrator - you can also enter

```
% destroydb foo
```

at the Unix shell prompt to remove the database.

2.4.3 LIBPQ

The Postgres DBMS does not have a graphic user interface or a report generator, but instead it has a C-interface with an extensive set of functions that can be called by user-applications. This communication interface is a library called **libpq**. It provides routines to pass Postquel queries to the backend and to retrieve the results which are returned in buffers called *portals*. Because of this C-interface, Postgres can be called an open DBMS, which also means that the user has the ability to define and add additional objects, operators and functions to the system. Thus, handling of data that uses these additions can be implemented more efficient that in a standard DBMS.

2.4.4 GEO++

The current GEO++ system is a Postgres frontend. It provides a general purpose frontend for the Postgres database system, using the LIBPQ interface. GEO++ enables users to interact in a graphical way with the Postgres DBMS, which means it takes care of displaying the graphical information for the user, and obtaining all the input from the user. This graphical user interface is shown in figure 2.4. In this system, all query processing and maintenance of the data is still diverted to the Postgres process. In addition, it expands the Postgres system with a number of self-defined spatial types and functions.

The architecture of the GEO++ system is shown in figure 2.5. As is shown, the system's capabilities can be divided into several components. Following is a short description of each of these components:

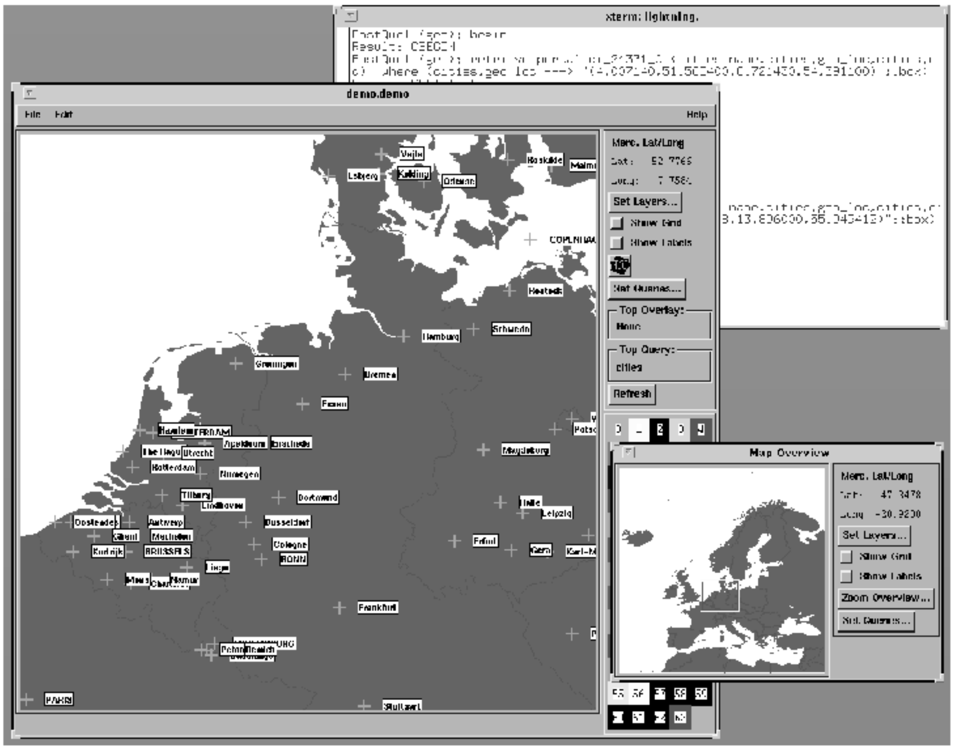


Figure 2.4: The GEO++ user interface

- Query Composer; You can graphically compose *where clause* restrictions for the relation being displayed, in the structure of a tree and using an advanced graphical editor. This editor will automatically detect and report syntax errors in the clauses, by consulting the Postgres Meta Relations.
- Query Parameter Control; this component will automatically check the types of arguments to operators. It does not allow the creation of syntactically incorrect clauses.
- Relation Viewer; You can view Postgres relations.
- Tuple Viewer/Editor; You can add or edit Postgres tuples in relations that are being displayed.
- Postgres Relation Cache; this component maintains information regarding the Postgres relations. With the help of this component, you can eg. use an automatical redraw option so that relations that have changed will automatically be redrawn, allowing you to create moving objects. This feature uses the Postgres Asynchronous rules (triggers).
- Display Control; You can view the results of these queries on a graphical map, where you also have the ability to zoom in or out, or pick displayed objects to view their corresponding Postgres tuple. A second map is displayed showing the context of the main map (see also figure 2.4). You can also add labels to the displayed objects, using their attribute names and/or values. And you can change some other display properties, like the colours of the objects, or the width of polylines, or even the drawing order of the objects.
- Meta Relation Cache; Meta Relations contain information about the existing operators. They are used for several purposes, like to describe the Regular Expression that all used Postgres type constants must match, thus ensuring the construction of correct *where clauses*.
- Dynamic Linker; Just as you can extend POSTGRES with new data types, relations, functions and operations written in the C programming language, GEO++ can be extended with new GEO types (the so called *QueryShapes*), which are written in C++ and in which the user can specify how he wants certain POSTGRES objects (tuples) to be displayed.

Note that the above description only mentions some of the features that the system offers. Each of the components is capable of doing much more than what is mentioned above.

Postgres already has some spatial access types by default. These include **point**, **lseg** (for lines), **path** (for polylines and polygons), **box** (for rectangles) and support for several indexing structures, like B-trees [28, 29] and R-trees [11]. A limited set of functions and operations is also supplied. For GEO++ however, instead of using the supplied **point** and **path** types, alternative datatypes were created, 2-dimensional as well as 3-dimensional. Thus in 2D there is **POINT2**, and to distinguish between polygons and polylines, **POLYGON2** and **POLYLINE2** were created. A series of matching functions was also created, with names that clearly point out the purpose, the dimension and the operand types. For example the function for checking 2-dimensional points on equality was called **Equal2PntPnt**, and denoted with the operator symbol **==**, and the 2D function that checked if a point was on a polyline was called **On2PntPln** and denoted with $-->$. Similar types and functions are also available for 3-dimensional access.

The link between GEO++ and Postgres has advantages as well as disadvantages. One major advantage is that GEO++ does not have to worry about processing the queries, which can all be done by Postgres. Another advantage is that, if a user wants some additional functionality to process the data, this can be achieved by adding new types, relations, operators/functions and/or rules to the Postgres system. The GEO program can remain unmodified. However, the disadvantage is that the program has to wait for the Postgres system to complete, which can take quite long for larger queries. Apart from that, the Postgres system that was used with GEO++ at the time of this project (ie. version 3.0) was not very well suited for geographic query processing, because internally, the entire system operated on an aspatial basis, and it provided no operations or data structures for the processing of spatial data. The current Postgres version 4.0 already provides a significant speedup compared to version 3.0, meaning that one of the major disadvantages of the previous version has been reduced significantly.

For more info on Postgres, see also [4].

2.5 GEO++ and the Query Composer

An important part of the GEO++ interface consists of the so-called *"Query Composer"*, the part of the system that allows the user to define a query, and that passes the queries to the database level. Because of the importance of the Query Composer to this project, I will provide a description of its basic functionality and its user interface here. In a later chapter I will come

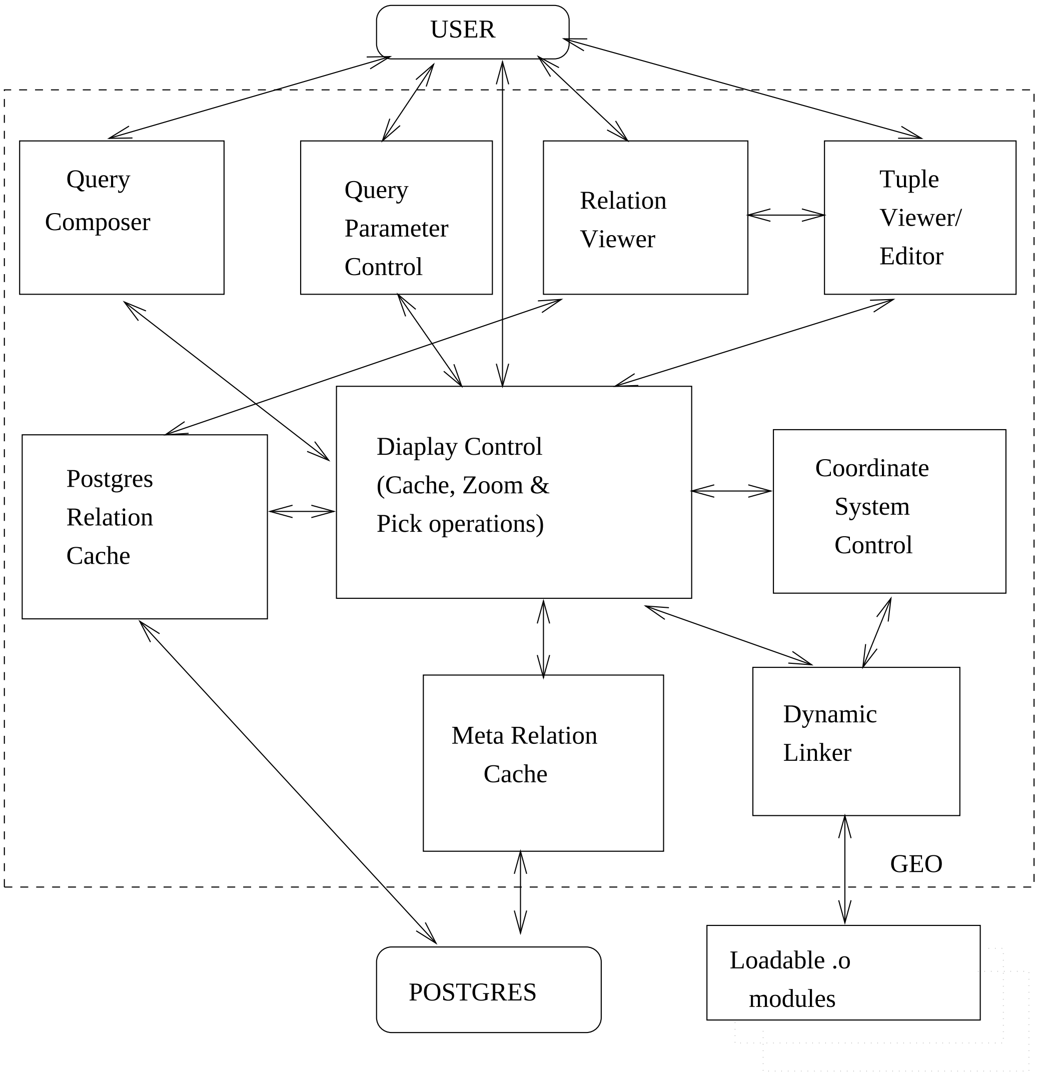


Figure 2.5: The old GEO++ functional decomposition

back to this topic and describe the changes that were made to the Query Composer during this project.

This section will only give a brief overview of the Query layer; for a more extensive description of the Query layer, see the documentation, provided in [10]. Even though that documentation is not completely up to date, it will explain the most important functions to you.

The Query layer consists of several parts. After clicking on 'Set Queries...' in the Map Window, a query layer selector dialog will pop up. Select a name and click on 'Edit Item', or click on 'New Item', to bring up the 'Query Dialog' (see figure 2.6).

2.5.1 Query Dialog

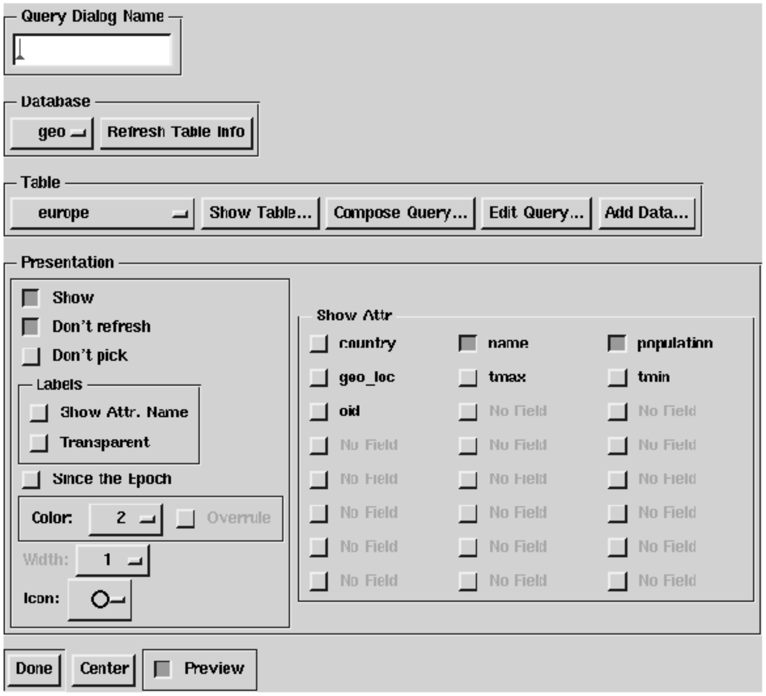


Figure 2.6: The Query Dialog

On the Query Dialog window, the user can select a database and a table, and specify which attributes from that table (relation) he wants to have displayed. Eg. the table of cities consists of the attributes **country**, the **name** of the city, its **population** and its geographical location **geo_loc** (coordinates). All of these attributes can be displayed at the same time, but the user can also decide that eg. only the name is relevant, in which case he can toggle (turn on) only the name, and the rest of the attributes will not be displayed. The user might also want to change the colour of the displayed items, or the graphical images used for them. Both of these changes can be made on the Query Dialog window.

In addition, the Query Dialog window offers four buttons that will bring up other windows:

- Show Table...; This will bring up the 'Table Dialog', displaying the values of all the objects of the specified relation (see figure 2.7). If the 'show' flag on the Query Dialog is turned on, only those objects that are currently visible on the map window, will be displayed. There is also a possibility to modify values in the table.

- Compose Query...; This will bring up the 'Query Composer Dialog' (see figure 1.1 from chapter 1). This is where the actual creating and editing of the query's 'where clause' is done. See the next section for more information.

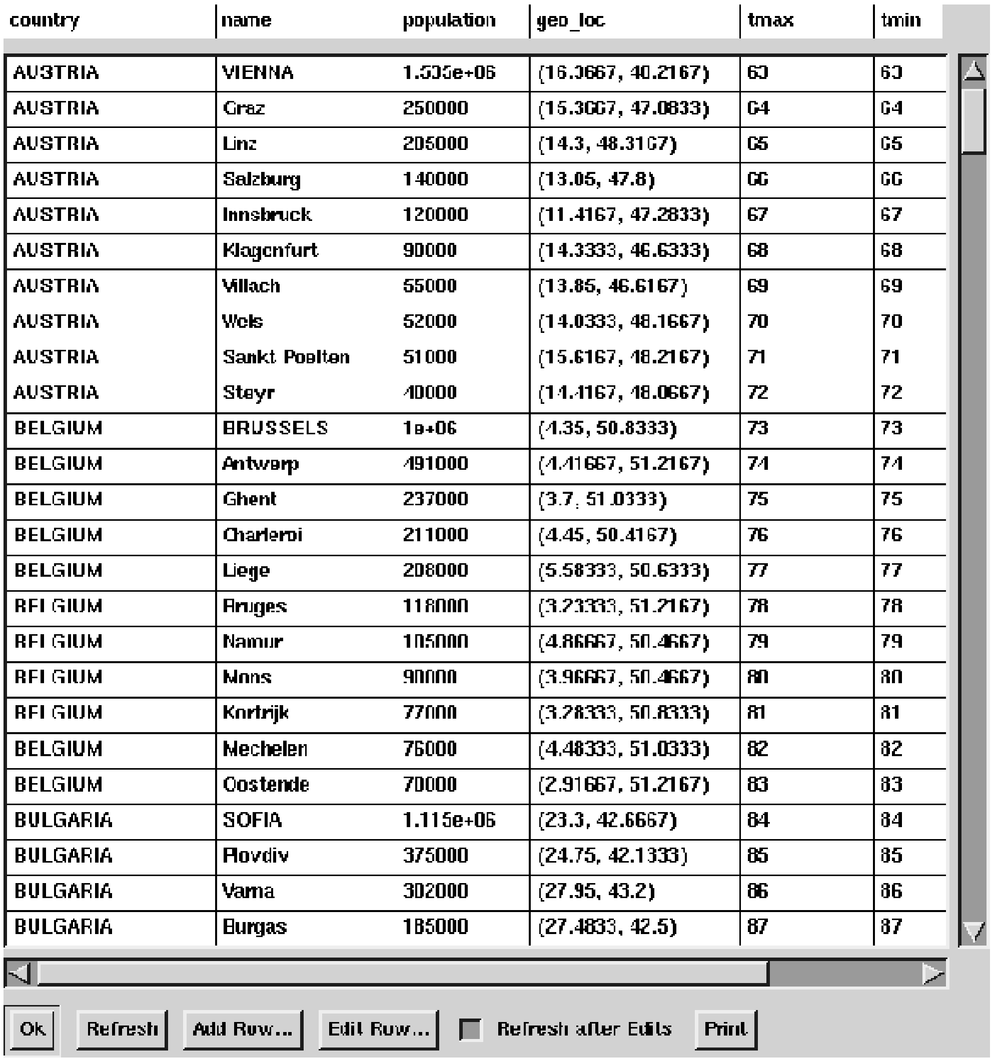
- Edit Query...; Although this button is displayed in figure 2.6, this is a new button, not present in the original GEO++ system. Its function will be described in detail in chapter 5.

- Add Data...; Here the user can add new data to the table, like new cities to the table of cities, but he can also add a completely new attribute to a relation. For example, the user could add an attribute 'Mayor' to the cities table, in which the name of the current mayor can be stored.

2.5.2 Query Composer Dialog

As stated above, clicking on the 'Compose Query' button on the Query Dialog will bring up the Query Composer Dialog, where the actual composing and editing of 'where clauses' is done (see figure 1.1 in chapter 1).

Here the user can create 'where clauses' by performing certain transformations on the current 'query tree'. Initially, the query tree contains only one single node, titled 'bool'. The user can now select this node, and change it into an 'Operator' that results in a boolean value, such as any boolean operator on a certain type of values, like `<=` on (int2, int2) value pairs, or an



country	name	population	geo_loc	tmax	tmin
AUSTRIA	VIENNA	1.535e+06	(16.3667, 48.2167)	63	63
AUSTRIA	Graz	250000	(15.3667, 47.0833)	64	64
AUSTRIA	Linz	205000	(14.3, 48.3167)	65	65
AUSTRIA	Salzburg	140000	(13.05, 47.8)	66	66
AUSTRIA	Innsbruck	120000	(11.4167, 47.2833)	67	67
AUSTRIA	Klagenfurt	90000	(14.3333, 46.6333)	68	68
AUSTRIA	Villach	55000	(13.85, 46.6167)	69	69
AUSTRIA	Wels	52000	(14.0333, 48.1667)	70	70
AUSTRIA	Sankt Poelten	51000	(15.6167, 48.2167)	71	71
AUSTRIA	Steyr	40000	(14.4167, 48.0667)	72	72
BELGIUM	BRUSSELS	1e+06	(4.35, 50.8333)	73	73
BELGIUM	Antwerp	491000	(4.41667, 51.2167)	74	74
BELGIUM	Ghent	237000	(3.7, 51.0333)	75	75
BELGIUM	Charleroi	211000	(4.45, 50.4167)	76	76
BELGIUM	Liege	208000	(5.58333, 50.6333)	77	77
BELGIUM	Bruges	118000	(3.23333, 51.2167)	78	78
BELGIUM	Namur	105000	(4.86667, 50.4667)	79	79
BELGIUM	Mons	90000	(3.96667, 50.4667)	80	80
BELGIUM	Kortrijk	77000	(3.28333, 50.8333)	81	81
BELGIUM	Mechelen	76000	(4.48333, 51.0333)	82	82
BELGIUM	Oostende	70000	(2.91667, 51.2167)	83	83
BULGARIA	SOFIA	1.115e+06	(23.3, 42.6667)	84	84
BULGARIA	Plovdiv	375000	(24.75, 42.1333)	85	85
BULGARIA	Varna	302000	(27.95, 43.2)	86	86
BULGARIA	Burgas	185000	(27.4833, 42.5)	87	87

Figure 2.7: The Table Dialog

attribute returning a boolean value, but also 'AND', 'OR' and 'NOT' operations resulting in a boolean. Let's assume that the user has selected 'AND'. Now the node labeled 'bool' will be replaced by a node labeled 'AND', and two lines will be attached to that new node, one leading diagonally up, one diagonally down, to new nodes both labeled 'bool'. The user can again select one of these nodes, and replace it with an attribute, operator or boolean, as long as the selected operation returns a value that is of the expected type. By each time selecting a node, and changing it into the kind of operation the user wants, the query tree can be transformed into a complex where clause.

Once the user is satisfied with the resulting where clause, he can select 'Ok', and if the 'Show' gadget on the Query Dialog is activated, the results of the selected query will immediately be calculated and displayed on the screen.

A more complete description of each of the elements of the GEO++ interface can be found in [10].

2.6 Requirements

So far I have described the existing programs and interfaces. Now I will briefly describe what will have to be done in order to implement the desired Spatial Query Evaluator.

Instead of the external Postgres interface, an internal database level with a DataBase Manager will have to be set up (see figure 2.8). The tasks of the Postgres Relation Cache and the Meta Relation Cache modules will be taken over by Display Control, or by the new Spatial Query Evaluator, which will be the communication port between the DataBase Manager and the rest of the application (ie. the Relation Viewer and the Display Control modules). All the other modules will be kept intact, as far as possible. Some changes will have to be made to the Query Composer, to allow for a direct communication with the Spatial Query Evaluator, and for direct manipulation of sets of query results. However, for the user, there should still be no apparent difference between the handling of spatial and aspatial data.

The storage architecture has to handle both spatial and aspatial queries, and arbitrary combinations thereof. However, the user should not have to be aware of the differences in how these two types of data are treated, and should still be able to treat them exactly the same. Also, the new storage and parsing system should have no effect on the way the data are treated and viewed at the user interface level. In other words, for all the operations from the old GEO++ system, it should make no difference to the user if he uses the new or the old architecture. And with regard to the new Spatial Query Evaluator

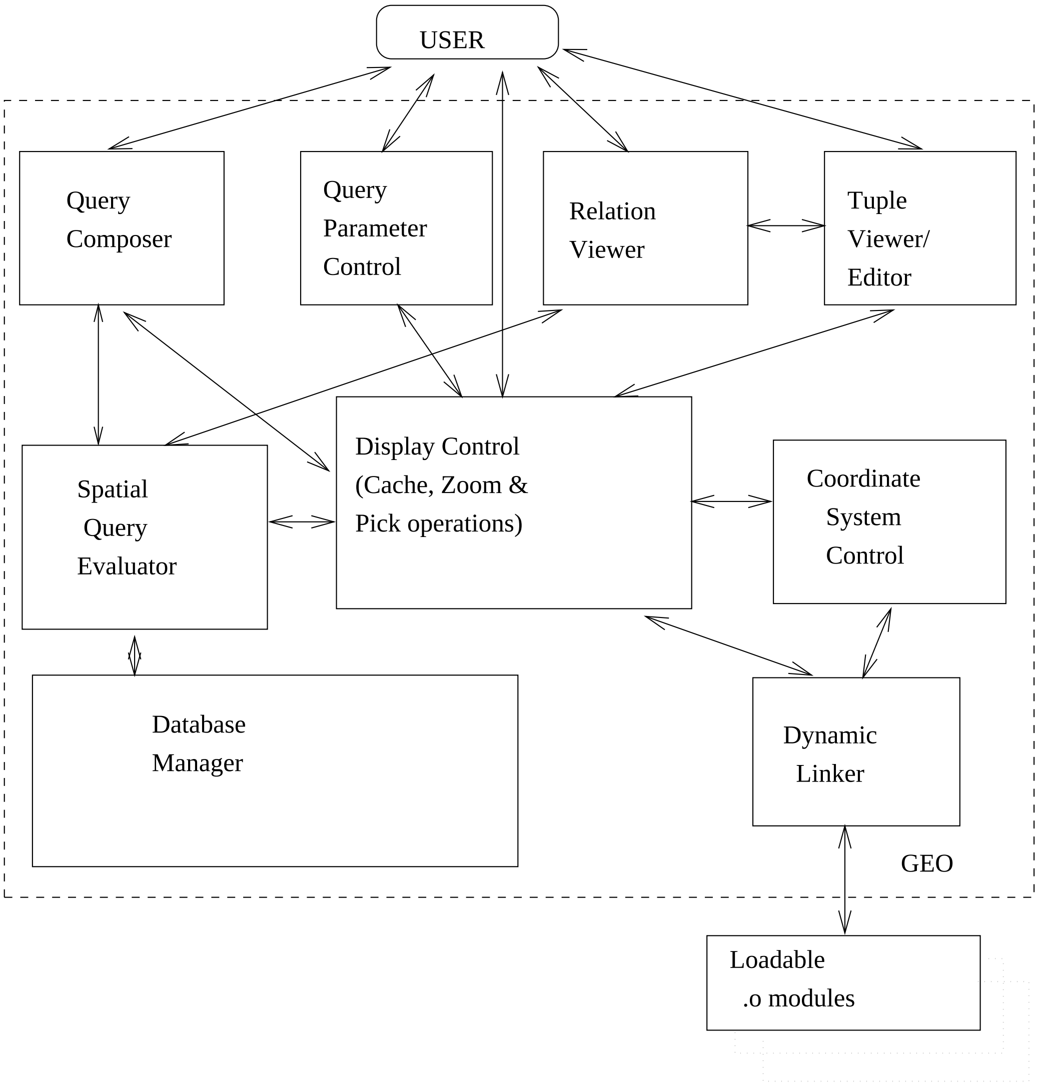


Figure 2.8: The new GEO++ functional decomposition

module: it should allow the user to experiment with the parameters of a query, and provide fast feedback on the effects of such parameter changes.

There will be some restrictions, though: the new system should be able to handle point and line data, in addition to the basic numeric and alphanumeric data types (integer values, floating point values and character strings), but it would not have to be able to handle more complicated types, like boxes or polygons. Although the original GEO++ system had the possibility to handle box and polygon type data, and even enabled the user to extend or modify the system functions and datatypes by means of dynamic linking, both these additional data types and the powerful dynamic linking feature have been disabled, because allowing such flexibility would have a negative effect on the speed of the system in general, and the Spatial Query Evaluator in particular, and it would severely complicate the implementation of the new system.

Chapter 3

ET++

The GEO++ system is written in the C++ programming language [6], with the help of a large class library called ET++ [7]. This chapter tries to give a short insight into the background, the capabilities, and the pros and cons of ET++ and the 3.0 alpha version used of that class library - the latest available version at the time of this project.

ET++ is an extensive and powerful class library with mainly graphically oriented functions for the C++ programming language. It is especially suited for developing large applications like Graphical User Interfaces and Case Tools. It was developed at the Institute for Computer Science of the University of Zurich by Erich Gamma, Rudolf Marty and André Weinand, and is based on the concepts of MacApp [43] and ET. ET was a non-object oriented **E**ditor **T**oolkit, designed at the same institute. As the name suggests, ET++ is written in C++ and is, unlike its predecessor ET, fully object oriented, making it a lot more flexible and powerful than both ET and MacApp. ET++ requires a computer running Sun OS 4.1 or higher, and SunWindow/SunView, X11R4 or OpenWindows 3.

Of course, there already exist several other libraries and low level toolkits that provide similar functions, and it is possible to use these libraries or toolkits instead of ET++ for building User Interfaces and Case Tools, but the disadvantage of those methods is usually that although they aid in building the user interface itself, they provide no help for the creation of the underlying application. Because of that, you may still need to have several duplicates of the same pieces of code in your program, otherwise you may get things like inconsistent menus. ET++ was built to overcome these disadvantages of the old methods.

In its current form, the ET++ class library holds over 200 classes and over 2000 methods. A class library with such a large size and functionality has one major drawback however, and that is the large initiation period. To solve this problem, the creators have supplied a development environment containing a powerful Browser, and in addition to that, a tutorial was developed that should enable new users to familiarise themselves with ET++ within a period of approximately two weeks.

As stated above, the version of ET++ used for this project was 3.0 alpha, although all available documentation was for version 2 of the class library, because at the time of the project, there was no documentation available specific to ET++ version 3. Under such circumstances, ie. with such a lack of documentation, and with the large number of bugs present in this alpha version and in the included example programs, the aimed at initiation period of 2 weeks is almost impossible to reach; in most cases, including my own, the initiation period will be much longer.

Anyway, below I will try to give some insight into the concepts behind the class library, a short overview of the most important Classes and Methods used in the realization of this project, as well as provide a global description of the basic ET++ hierarchy. The class library is far too large to provide an even near complete description of its classes and functions. For more information about the ET++ package and its functionality, please refer to other literature, such as [7, 8, 3].

3.1 Basic knowledge about ET++

3.1.1 Application Framework

The ET++ package provides an *Application Framework*. In an Application Framework, control-flow lies in the hands of the framework, and the framework functions are intertwined with the application's functions, that is, no strict separation is available, and framework functions can be used just like - and alongside with - ordinary functions. Alternatives would be:

- Libraries; The library functions are called from the application, and all control-flow still remains in the hands of the application.
- Toolkits; The control-flow lies with the toolkit, but the toolkit functions and the other application functions are strictly separated.

3.1.2 Model View Controller

ET++ Applications follow the Model View Controller principle, which means that an Application is divided into *Model* (for data), *View* (for displaying "virtual" drawing-planes and events), and *Controller* (for the processing of events).

3.1.3 User Interface

ET++ utilizes X-window-like graphical representations, which means it implements the typical *point & click* interface. For the control of events, ET++ depends heavily on *Popup*- and *Pulldown*- menus, as well as *Dialog* boxes.

Each Application has to create at least one `Window`. The virtual drawing-plane, which is an object of the ET++ class `View`, can be displayed in several planes, which may overlap, and which are displayed in (a part of) the `Window`. Such a `Window` represents a frame for displaying data, but not the contents of the frame. For the contents, ET++ objects called `VObjects` are used. There is a range of different `VObjects` available (some of which are shown in figure 3.1), including:

- `Clipper`; this shows a rectangular section of the virtual drawing-plane of a `View`.
- `Scroller`; this shows a rectangular section of the virtual drawing-plane of a `View`. With scrollbars, the user can scroll this section both horizontally and vertically.
- `Splitter`; this shows up to four sections of the virtual drawing-plane of a `View`. Two of these sections share a vertical scrollbar, and the two others share a horizontal scrollbar.
- `Expander`; an `Expander` allows the user to divide the virtual drawing-plane in several sections, with different `VObjects` handling each of these sections. Thus, the user could eg. use a `Clipper` or a `Scroller` for one part of the `Window`, and a `Splitter`, or even another `Expander`, for another part of the `Window`.

ET++ supports several types of *Dialogs* with the user: editable text strings (`TextItems`), `Menu`'s and `Buttons`. There are different `TextItem` classes for entering text strings, floating point numbers and for integer numbers. The items in a `Menu` may be `TextItems` (text strings), submenu's, or even images. And there are different kinds of `Buttons` as well:

- `ToggleButton`; a small square button, when activated it will be filled with a toggle mark ($\surd$). It is used to switch between two states, eg. TRUE and FALSE of a boolean value.
- `RadioButton`; a small square button, when activated it will be black. These are usually used in groups, so that only one of the radio buttons is activated at the same time, ie. activating one will automatically deactivate another.

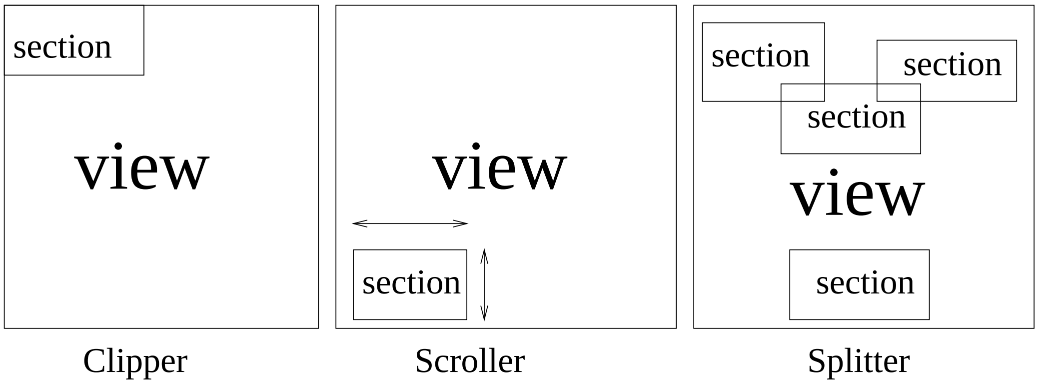


Figure 3.1: ET++ View objects

- `ImageButton`; a button with an image in it, like the "arrow" buttons in a scrollbar. They are used to initiate a certain action, like in the case of the scrollbar arrows, "scroll to the left" or "scroll to the right".
- `ActionButton`; an oval button, containing a text string. This kind of button is also used to initiate a certain action.

3.1.4 Classes

ET++ is written in C++, using classes. A very important class for creating an application is the `Manager` class. An ET++ application manages a collection of `Managers`. The `Application` class itself is a subclass of `Manager`. The application model of ET++ consists of a hierarchy of objects based on the `Manager` class, with the `Application` object as root. Some other frequently used classes based on the `Manager` class are `Dialog` - for dialog boxes, `Document` - for load and save functionality, and `Menu` - for a menubar (see also figure 3.2).

Most of the ET++ classes are directly related to the user-interface, however, the ET++ package also has some classes that provide other functionalities, like data structures. Classes for the most common data structures like arrays, hashtables, dictionaries, lists and ordinary collections are provided, and if the programmer needs more complex structures, they can be created using the available classes.

For more information, refer to eg. [7, 8]

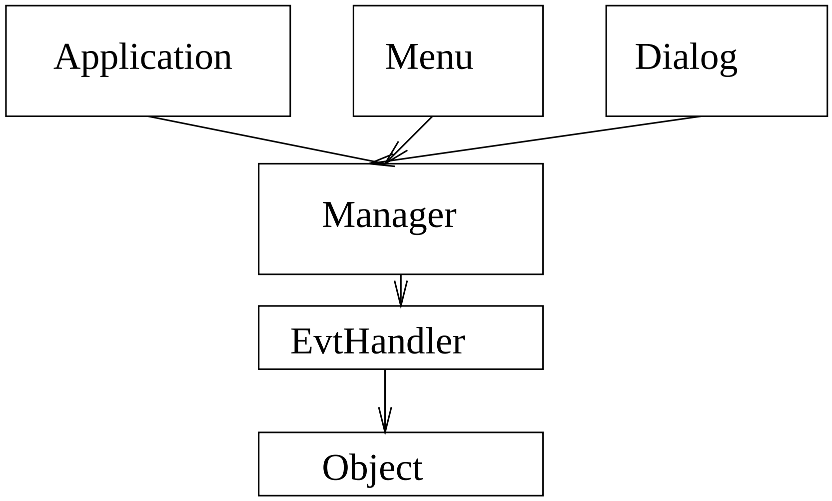


Figure 3.2: ET++ class hierarchy

3.1.5 Data structures

The ET++ system provides a limited number of classes describing data structures, mainly for the handling of data in the graphical parts of the system. An example is displaying a list of items in a `View`, and enabling the user to scroll through them. Some of these classes are `Dictionary` - for a collection of ⟨key, value⟩ pairs, `HashTable` - for a hashtable structure, `ObjArray` - for an indexed array, `ObjList` - for a simple list, `OrdColl` and `SeqColl` - for array-type structures obeying some kind of ordering, `Set` - for simple set structures, and `SortedOList`, for a list of objects that can be sorted using some predefined sorting algorithm. As can be seen, all of these data structures are mainly for storing aspatial data. No special data structures for handling spatial data are included. It appears that the provided data structures were only meant for handling internal data directly related to the graphical interface, and were not meant to be used in database management systems.

3.2 Additions

3.2.1 ValSlider

Because the default ET++ `Slider` class didn't offer enough flexibility, I have added an additional class, named `ValSlider`. This new `ValSlider` class is used within the `QEController` class (see chapter 5). It contains a slider bar, based on the ET++ class `Slider`. `ValSlider` creates a bar with two slider buttons in it, and displays the upperbound and the lowerbound value of a given range of floating point values. By using these two sliders, the user can set a minimum and maximum value that suit his own taste, as long as the values do not go outside of the initial range. That is, the lowerbound value cannot be made smaller than the initial minimum value, and the upperbound value can not be made larger than the initial maximum value. Also, the upperbound value can not be smaller than the lowerbound value. However, the lowerbound value can be equal to the upperbound value, in which case the range will consist of only one value.

See also section 5.2 for more information on how and where this new `ValSlider` class is used.

Chapter 4

The Spatial Query Evaluator: Database Level

The **Spatial Query Evaluator** (SQE) module can roughly be divided into two components, namely the database component, and the user interface component. As GEO++ already had a good user interface, the new user interface stuff would be additions to and modifications of the existing interface. A detailed description of the new user interface can be found in the next chapter (ie. chapter 5). On the other hand, the old database code consisted of calls to and use of the external Postgres interpreter. Because those calls were relatively slow for large spatial queries, this code had to be rewritten completely and replaced by an entirely new database level, without depending on exterior programs. In the following sections, the new database level will be described, taking a look at the initial goals, which design decisions were made, which problems were encountered and what solutions were used for them, as well as trying to provide a detailed description of the new database level's hierarchy and data structure.

In the first section I will briefly set out the goals that I have tried to reach during the realisation of this project. The next section will explain in detail the planning stage of the database level, which choices were made and why. The last section will describe the actual implementation of the database level.

4.1 Goals

At the start of the project, I established a series of design goals that an ideal SQE module should satisfy. First of all, I wanted to obtain a fast and easy-to-use interface (see chapter 5). In addition to that, I intended to remain as compatible as possible with all currently existing parts of the GEO program, so that the SQE module could be fit into the already existing parts without much work. Preferably, the module should also be easily extendible, thus completely removing the need for the previous Postgres interface. Furthermore, things like easy importing and exporting of data also played a part in these considerations. And ofcourse, it should be as fast as possible, and in those situations where response cannot be instantaneous, feedback to the user should always indicate what the program is doing at the time. This also means that a data structure has to be found that is fit for processing both spatial and aspatial data. On the other hand, for the user, there should be no apparent difference between spatial and aspatial data. Also, the selected data structure(s) should preferably not require a lot of reorganisation, balancing, splitting and merging (of subtrees).

4.2 Terms

This section is dedicated to the explanation of some terms that will be used throughout the rest of the chapter.

4.2.1 Access methods

As most objects, such as lakes, roads, etc., do not fit one fixed shape, spatial tests on these objects, such as containment or intersection searches, can be expensive. For that reason, different methods of approximation have been developed to approximate such irregularly shaped objects. The two most common of these access methods are *region decomposition* and *minimum bounding rectangles*. Region decomposition is especially suitable for image processing, but less suitable for general purpose databases. For that reason, I will not describe it in detail here. That leaves only minimum bounding rectangles.

A minimum bounding rectangle (short: MBR) is the smallest rectangle that encloses an object. MBR's are most suitable for proximity query processing. For region type search operations, the MBR can offer an efficient way of query processing, as it limits the area that has to be searched to the area around the object. However, for diagonal objects (like diagonal lines), the MBR may give bad approximations and false matches, for example when trying to determine if two objects intersect, and it may lead to unnecessary searches in subtrees where no relevant points are stored. One way to limit the risk that this may happen, is to split objects up in parts - so-called *'cells'* - and compute the MBR for each of these cells. However, this leads to an even more serious problem, because the search index has to identify all these cells as belonging together, so it both increases the required amount of storage, and complicates the searches. For those reasons, and for lack of a better alternative, I have chosen to use MBR's.

4.2.2 Spatial Indexing Techniques

Spatial searches can be compared to aspatial searches on multiple keys, when you map the coordinates onto the key attributes, and when the key values are used to store a point in a multi-dimensional space. But the problem is that not every spatial object can be represented by (zero-sized) point values, as there can also be non-zero sized irregular shaped areas in such a multidimensional space. Although such objects are basically outside of the scope of this project, I want to mention the three major techniques that are used to handle multi-dimensional point structures, because they will become relevant should attempts be made in the future to enhance the system to be able to handle such objects. They are:

- Object Mapping or Transformation; objects are mapped from a k-dimensional space to points in a 2k-dimensional space. Thus, for example a 2-dimensional rectangle can be stored as a point in 4-dimensional space. Point indexing structures can be used to store the data, but special search algorithms will have to be written for things like containment and intersection searches.
- Object Duplication or Object Clipping; both are straight forward extensions to the point indexing structures. In the case of duplication, object identifiers are stored in all the subspaces with which the object intersects. In the case of clipping, the object is split up into smaller objects and such a smaller object is stored in a subspace that entirely includes it.
- Object Bounding; this method keeps track of the boundaries, or the overlapping bounding rectangles, in a subspace by which the object is totally enclosed.

4.2.3 Miscellaneous

Other relevant design decisions include the question what way will be used to measure the distance to an object. Here the choices are:

1. You can take the minimum distance between any point on the boundary of the involved objects.

2. You can establish a center point for each of the objects, and take the distance between these center points.

3. You can take one of the two methods described above, and apply it to the minimum bounding rectangle of the object.

When people use queries to determine the distance between for example two cities, or from a river to a mountain, they usually want to know the distance between the boundaries of the objects, and not the distance between the centres. Therefore, I have chosen to use the first alternative.

4.3 Planning the database level

In a Geographic Information System, the opportunity must be given to store both spatial and aspatial data. Thus a storage and information architecture must be provided that can deal with both kinds of data, preferably one where the two kinds of data are integrated into one architecture. Apart from the storage architecture itself, I also had to create an interface mechanism to handle the entered queries and retrieve the appropriate data for them.

Because of the entirely different nature of spatial and aspatial data, the decision was made to use *query decomposition*, which means that each query would be decomposed into several simpler subqueries, in such a way that each subquery is either entirely spatial, or entirely aspatial, and so that the semantics of this sequence would stay the same. Obviously, the program has to be able to distinguish between these spatial and aspatial subqueries, and deal with them appropriately. What has to be taken into account, when using this decomposition method, is the order in which the subqueries are evaluated. It is vitally important that the subqueries are evaluated in such an order, that they yield the correct end result. Also, there are cases in which a subquery is dependent on the results from another subquery. In those cases, the latter subquery has to be evaluated first. In addition, the order in which the subqueries can be evaluated can make a big difference in terms of speed; some orders can be processed a lot more efficient than others.

In most common GIS systems, aspatial data is processed by an independent conventional database management system, but in this case, a method had to be developed to efficiently process both kinds of data within the same system.

For processing aspatial data, something like an alphabetically sorted list will often be best. But for efficiently processing geometric (spatial) data, it is necessary to provide operations for locating the neighbors of an object, as well as for identifying objects in a certain query region. For the first, proximity-type, operations, it is necessary to provide a way of indexing on the spatial location. There are several possible solutions for this kind of indexing, however, the data has to be stored in such a way that the second, region type operations, do not become very costly.

For proximity purposes, we need a multi-dimensional (point) indexing structure. Lots of different data structures have been proposed for this, such as multi-dimensional B-trees, kd-trees, and quad trees. There are two different types of queries on such structures: exact match queries (search for a particular point), and range queries (search for all points that lie within a certain range).

4.3.1 Data structures

The differences between spatial and aspatial data are large. Special relationships exist between spatial data objects, and although it is possible to create conventional links for the most frequently used objects and relations, it is usually not feasible to store the entire data-collection that way. So it will be necessary to dynamically evaluate the spatial relationships at run-time. For that purpose, it is necessary that the program supports some kind of spatial indexing mechanism. If the standard indexing techniques from a conventional DBMS would be used, then even the evaluation of relatively simple queries such as *"Find all cities that are on a distance of at most 10km from Amsterdam"* may well require traversing the entire database. It would be much better if the data-structure would already support a restricted search, based on proximity, so that only objects near Amsterdam would have to be checked.

Many data structures/indexing structures for spatial data are proposed in literature. Globally, these can be divided into two categories: topological structures (with a linked organization) and grid structures (with a cellular organization). Topological structures order the sets of points and line segments, while grid structures subdivide the area of into a grid of rectangular planes. Because of that, these grid structures have two major disadvantages. First, all spatial entities must be viewed in terms of coordinate strings or grids, which is extremely low level. And second, for query processing, proximity search cannot be used, so basically, all entities have to be compared. I was not prepared to accept these disadvantages, so I have decided to entirely skip grid structures and focus completely on topological structures. This section will provide a brief overview of a number of topological structures, and try to sum up the main advantages and disadvantages of each of them. It will not provide an extensive analysis of each of the data structures. For that, and for additional information about the structures, please refer to the original authors.

The Point Quad-Tree

The point quad-tree [21] can be seen as a more general, multi-dimensional version of the binary search tree [22]. Each node in a k-dimensional tree contains a data point and 2^k leaf nodes, each one referring to a so-called 'quadrant' of the tree - even though only a 2-dimensional quad-tree actually has 4 leaves. This kind of structure supports both exact match queries and range queries. Insertion of new points is relatively easy, but deletion of a point usually requires reorganization of the tree. It is a simple structure, with relatively good performance. However, the most important restriction of this structure is that no two points may have the same coordinates. As I cannot make this assumption for all the kinds of data that can be stored in the database, this makes this structure - in its basic form - unusable. Approximately the same restrictions hold for related structures such as the **Region Quad-Trees** - which are most suited for image processing purposes, and will not be further discussed here.

Other quad-tree based structures

Quad-trees use region decomposition for partitioning regions. There are several different types of quad-trees, like the region quad-tree, the quad-CIF-tree [23, 24], and the PR quad-tree [25]. Such structures are especially suitable for image processing, and for VLSI applications. The disadvantage of these structures is that intersecion search as well as insertion of a new object can be quite costly.

The 4-D-Tree

The 4-D-tree [26] is an extension to the standard K-D-B-tree [42], in which 2-dimensional rectangular objects are mapped onto points in a 4-dimensional space, like this is done with the kd-tree. However, a very big disadvantage of this structure is that intersection search can be very expensive (and slow), also because in most cases, multiple subtrees have to be searched.

The R-Tree

The R-tree [11, 27] is a version of the B-tree [28, 29], generalized to multiple dimensions. For inserting objects into the tree, and deleting objects from it, it has the same disadvantage as the B-tree, ie. it is very costly, because these operations require node splitting and merging. The main problem is to build the trees in such a way that overlap of bounding rectangles is minimized, because overlapping means that in searching, multiple subtrees may have to be searched, increasing the cost of search operations.

The R$^+$-Tree

The R$^+$-Tree [30] is a version of the R-tree that also contains features of the K-D-B-tree [42]. This structure provides a solution for the above-mentioned problem in R-trees with respect to overlapping rectangles, and has paging support built in. It usually is good for all types of data and queries, but it is especially suited for some classes of point-data. However, insertion and deletion of objects into/from the tree are even more expensive than in the R-tree, because they can require large tree reorganisations. Therefore, a clever algorithm is needed to overcome this drawback. However, there is an even more serious problem with this type of tree, and that is that in some cases, no suitable bounding rectangle can be found to store a new object into the tree, and in those cases, splitting and reorganisation of the entire existing tree becomes necessary, with the obvious negative effects on efficiency.

The kd-Tree

Another type of structure is the kd-Tree, a k-dimensional homogenous binary search tree. Nodes contain a data point (value) and a discriminator, which can have the value 1 upto k. The left branch of a node - also called the LOSON - with a discriminator value of j, then has data points where the value of the jth attribute is smaller than that value in the node, and the right branch, or HISON, has the data points where the value of the jth attribute is greater than that in the node. Exact match queries are very easy to do, and range queries are possible as well. Insertion of a node is no problem, deletion of a node may require some reorganisation of the tree, but less bad than with point quad-trees. There are ofcourse some disadvantages of this type of structure. First, it may not always be desirable that splitting (branching) occurs on different dimensions, and for range queries, several branches may have to be used to obtain the required range of points (values). Second, a clever splitting algorithm will have to be created that is able to split the set of points in two approximately equal halfs. Also, the indexing tree may become too large to hold in main memory, and may have to be paged to secondary storage. For the latter purpose, a slightly alternative version of the kd-Tree, namely the K-D-B-Tree [42], was designed, to which I already referred in some previous sections. However, in its basic form, none of these structures can be used for non-zero size objects, such as lines.

Matsuyama's kd-Tree

Matsuyama has proposed an alternative kd-tree, that is capable of storing non-zero size objects. This type of kd-tree will be refered to as Matsuyama's kd-tree, or simply mkd-tree. For a good description of this kind of tree I refer to [31]. Here, I will suffice with saying that the mkd-tree extensively uses the object duplication method, and copies of the same object can be present in several leaf nodes. Because of this, adding objects to the tree and deleting objects from the tree can be very expensive, and search operations may have to traverse multiple leaf nodes. Still, it is reasonably efficient for most kinds of search queries.

The Spatial kd-Tree

The spatial kd-tree [5] (short: skd-tree) is an extension of the kd-tree. This structure was designed to overcome the main problem with kd-trees, which is that the basic kd-tree is considered unsuitable for storing non-zero size objects, without having to use object mapping or duplication. The skd-tree provides an efficient way to store non-zero size objects without object mapping or duplication. In addition, containment search is directly supported, and even faster than the fairly efficient intersection search provided by this structure. Even secondary memory support (ie. paging) is possible. However, just like so many other structures, insertion and deletion of objects may require splitting and reorganising of parts of the tree. Still, from all the data structures I examined, this one seemed to be best suited to store and access the spatial data as required in the GEO++ system.

4.3.2 Query Optimization

GEO++ comes with a high level query language, which means that the user does not have to specify physical data access paths; he not even has to be aware of the difference between spatial and aspatial data. He need only concern with what he wants to know, not how it is stored. The system will take care of the rest. However, this means that the system is also responsible for efficient processing of the query, and therefore the system must try to find the most efficient way to map the user query to the underlying data structure. For this reason, it is wise to automatically optimize the given queries, before executing them. Optimization should always focus on response time, in order to keep the time required to process the query and supply the answer to the user, as short as possible. There are several ways of doing this. The best way is probably to add an additional *parser* level to the system, that can determine the most efficient way to split the query up in subqueries, and the 'best' evaluation order for these subqueries. This would also require the availability of a *meta-database*, in which information is stored regarding the cost (estimates) of all the available low-level operations. However, due to the limited time available for this project, and also due to the problems I had in other areas of the project, further research into this matter, and the actual implementation of this entire optimization-phase, had to be left out.

4.4 Implementing the database level

Beforehand, I was given the consent that I could assume that sufficient main memory is available so that all data can be kept in main memory, meaning there was no need to consider interaction with data on secondary storage (for example the grid file datastructure [32, 33] or the EXCELL Method [34, 35, 37]), which simplified matters considerably. This also means that the primary objective is speed of the operations and data access, and not memory usage. The Unix operating system already has a paging mechanism, so should memory become too sparse despite the available amount of memory, paging will be left entirely to the operating system.

Another assumption beforehand is that adding new objects to the data structure and deleting objects from the data structure are operations not used very often, thus more attention should be paid to fast data access and search operations than to fast insertion and deletion operations.

4.4.1 The main hierarchy

When the program is started, and when the user selects a new database to work with, or a new class, data is loaded into the program's global data structure. For this data structure, a layered (top-down) structure was chosen, consisting of several levels of data handling. The main reason for this was, that I wanted to be able to change the datastructure at one specific level, without affecting underlying levels - that is, without having to change any of the underlying data structures as well. The top level is taken by the `DataManager`, which is in charge of the entire database, and, apart from directing the underlying architecture, also keeps track of for example the name of the currently opened database. The `DataManager` is based on the ET++ class `Dictionary`, which means that it uses ⟨key,value⟩ pairs. In this structure, the key is the name of the database, and the value is a reference to the `DataColl` structure. Note that in the current setup, the assumption is made that at any particular time, at most one database is opened. However, it should not be difficult to change that assumption and allow for multiple databases to be open at the same time.

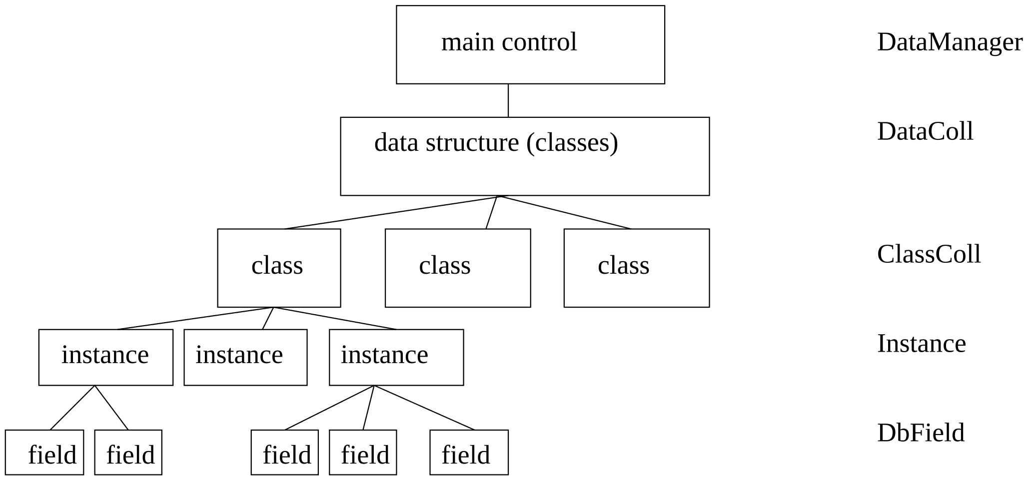


Figure 4.1: Main data-structure hierarchy

The second level in the hierarchy is the `DataColl` structure, which splits the data collection up into classes, and handle control over these separate classes. Again, this structure is based on the ET++ `Dictionary` datatype, where in this situation, the key is the name of the class, and the value is a reference to (the contents of) that class. Here, there is no imposed restriction on the amount of classes present in the dataset, apart from the sizes of available memory and secondary storage.

The next level in the hierarchy is taken by the `ClassColl` structure, which handles one separate class, and maintains the collection of instances that make up a class. Contrary to the two levels above this one, this structure is based on the ET++ `OrdCollection` (`SeqCollection`) datatype. This means that all the (references to) instances in the `ClassColl` are stored sequentially, as an indexed array.

Yet another level down in the hierarchy we can find the `Instance` structure, which handles one separate instance and splits it up into the fields that make up this instance. This structure is again based on the `Dictionary` datatype, meaning that the field references are stored under the individual names of those fields.

And the last level in the hierarchy is the `DbField` structure, which handles the control over (the contents of) the separate fields. This is the level at which the names, types and values of the fields are stored. This is also the level for which it is most important how the datastructure is set up. The structure has to be fit for both spatial and aspatial data. At the moment, the `DbField` structure is a relatively simple one, based on the `ByteArray` datatype. This means that all values are stored as simple character strings. An internal Convert routine makes sure that for each access type, a pointer to an object of the correct kind (character string, floating point value, integer value, point, line or otherwise) is returned. Due to this conversion routine, it doesn't matter whether spatial or aspatial data is stored. This datastructure does have one major drawback, however, which is that the conversion can slow down the entire system. An alternative would have been to create separate data formats for each implemented type. Although it might be a bit faster that way, this would also mean that lots of extra work would have to be done for each new data type that would be added to the system. Under the limited time I had available, I have not examine this further, however, if someone would want to try this, only this `DbField` level would have to changed.

4.4.2 Data structure for query processing

For efficient query processing and for the implementation of the Spatial Query Evaluator module, an additional data structure was implemented. I have examined a large number of data structures in my attempts to find a suitable one. In section 4.3.1, a number of those data structures were mentioned. Initially, the idea was to use a variation of the skd-tree structure, called the static skd-tree. It uses the assumption that the tree-structure in memory will not change much once the tree is created, in other words that the data collection is largely static, and it doesn't happen frequently that the user wants to add or delete new data to the system. Because this structure could become quite complicated and would take a lot of time, I started with an alternative that can be called a 'temporary hack', namely the `OrdDictColl` structure, based on the ET++ `Dictionary` class and the ET++ `OrdCollection` class, thus creating 2 arrays that can be accessed both sequentially - as an array - and as (key,value) pairs. That data structure was meant to be replaced by a faster, more intelligent, and more efficient data structure as soon as the query parser was finished. However, due to problems with the parser and serious lack of time, the structure was never replaced and is still present in the endresult. The advantages of this data structure are that it can handle both spatial and aspatial elements, and that it can easily be created using the underlying ET++ classes. However, disadvantages are that it does not support any special search facilities for spatial data, and that access to the structure is very slow.

Once a query has been evaluated and the results are stored in this structure, the OrdDictColl data-structure is relatively fast in providing the required feedback on the SQE level, however, internally, at the Database Manager level, the OrdDictColl is much slower than for example a kdb-tree structure. So the actual evaluation of the query and the gathering of the results is extremely slow.

4.4.3 Query Processing and Database Control

Another class, `FunctionParser`, was created, containing all the conversion and evaluation functions. This is also the location where all the operations on all kinds of data are executed. The `FunctionParser` recognizes a fixed set of data types and operations, and is able to perform any of those operations on those types of data. When an additional data type or an additional operation is added, functions to handle this type of data resp. this kind of operation will have to be added to this class. This also means that this class is the place where all queries are evaluated, unary and binary operators are recognized, where the query is split up in subqueries, the evaluation of all these subqueries is started, and where the final results are gathered. Any form of query optimization would have to be added at this level.

Note: because of the way in which datatypes are implemented now, meaning that - at the moment - all types have to be defined internally, I was forced to temporarily deactivate the ability for the user to add new datatypes to the system by way of `QueryShapes`, a method provided in the original GEO system for user-defined datatypes. This means that the external R-tree access method, which was used to provide the system with the ability to display raster data, and which was linked into the system by way of a `QueryShape`, was deactivated as well.

So how does it work? What happens internally after the user is finished composing or editing a query in the Query Composer? (see also chapter 2).

When a query has been created in the Query Composer and 'Ok'-ed by the user, the `QueryExec` function in the `QueryIter` class is called, which passes the query string on to the `FunctionParser` by calling the `DoEvalOper` function in that class. That function then processes the query and returns the initial list of all instances that match the query restrictions, as an OrdDictColl collection. That collection will then be sorted and stored in the SeqCollection variable 'insts'.

Once a query has been evaluated and a restricted dataset is created, the user can look at parts of that dataset by means of the `QueryEdit` dialog, where he can also set additional filters on the dataset. When the user has finished editing the (restricted) query set there and gives his 'Ok', the `DoRestricts` function in the `QueryIter` class is called, where for each instance in the original collection that matches the new query restrictions, a qualify flag is set to true. For each range value in the set of filters, both the old value and the new one are stored, and if the new values are within the boundaries of the old values, the current set of data is used. If some of the new values are outside of the range of the old filter, the qualify flags in the instances are cleared, and all the instances in the original (queried) set are used to calculate the new set of matching instances. (Note: `DoRestricts` is also called by `DoFetch`, to reset all the filters when a new query has been created and calculated)

4.4.4 Remarks and Future Development

At the moment we basically have an aspatial, ie. mainly alphanumeric-based datastructure to which were added some extra attributes and functions in order to improve the spatial performance. However, this structure is still not very well suited for processing spatial data, because it lacks the initial selection on location that is inherent to truely spatial datastructures. For spatial objects, an efficient dynamic evalution of the relationships is extremely important, otherwise the evaluation of queries that eg. attempt to locate the neighbors of an object or try to identify objects in a defined query region - such as any object within a certain distance of a given city - may require searching the entire database, instead of only having to check the objects in the vicinity of that object or region.

For that reason it would probably be better to replace the current datastructure with a spatial datastructure to which we could then add some aspatial indexes, attributes and functions, in order to also maintain a high performance on aspatial data.

Alternatively it is possible to maintain two different datastructures, one for storing the spatial data, and one for the aspatial data, and then adding an overarching structure with functions that can select between spatial and aspatial data, as well as combine elements from both of these types when that is required. This has the advantage that each set of data can be stored in that structure that is best suited for handling it, but the overarching structure can become quite complicated, and it has to be able to combine, using the object identifiers, aspatial attributes with the corresponding spatial aspects of the data.

A structure that would be very well suited for the spatial part of the above described data structures, is the spatial kd-tree (skd-tree) structure described in section 4.3.1, because that structure can easily handle both point objects and non-zero size objects, and because it supports separate search functions for intersection search and containment search. Also, due to the nature of the structure, object duplication and object mapping are avoided (see also section 4.2.2).

Another thing worth noting is that currently, the database files are still external and read from disk at startup - as they should. However, contrary to the Postgres approach, all datatypes must now be defined inside the program (ie. in the source) itself. Types that are not defined and thus not recognized by the program, are skipped, and their values cannot be retrieved or used by the program. This limits the user somewhat, as it does not allow for user-defined types, but leaving out this restriction and allowing user-defined types would require a lot of additional work on the module, and this was neither feasible within the given time limit, nor part of the original assignment.

Chapter 5

The User Interface

As the existing GEO++ program already had a good user interface, the new user interface consists mainly of small changes to the existing code, as well as some additions to take advantage of the new database level, and provide the user with more flexibility in selecting queries. This chapter will give a detailed description of which changes and additions were made, and why.

5.1 Changes

On the original Postgres-version of the Query Composer's Edit Dialog, there was a "Manual Edit" button, which, when pressed, would allow the user to manually enter the query string that he wanted. Because of the new implementation, this "Manual Edit" button had to be removed; allowing the user to enter a query string manually, would also require adding an extensive text parser for the queries. As adding such a parser would require spending too much time rewriting parts of the program that were not really top priority, it was left out. However, if the need should arise, it is always possible to allow manual entry of queries by writing such a parser, adding it to the existing program and putting the "Manual Edit" button back in the Query Composer. Currently, all queries will have to be built up by selecting the parts of which they are made up on the graphical interface.

One of the disadvantages of the new implementation is that in its current form, it is no longer possible to add new datatypes to the system by way of **QueryShapes**. Currently, all datatypes that are supported by the system, have to be defined internally. Although I did not investigate this further, due to the time restraints on my assignment, it should be possible to write an extension to the current functions, that will also allow linking in external datatypes.

5.2 Additions

To allow the user to experiment with the range of values returned by a certain query specified in the Query Composer, a new `QueryEdit` class was created. It consists of three parts: the `QEController` main unit, the `QEDialog` dialog window, and the `ValSlider` slider bars. The new `ValSlider` class was already described in section 3.2, the other two parts will be described below.

5.2.1 QEController

The QEController is the main control unit of the QueryEdit class. It is activated by clicking on the 'Edit Query...' button in the QueryDialog window. This is only possible after the user has specified a query on the current data in the Query Composer. It opens a QEDialog window, and takes care of the communication with the database level to get the names and types of the database fields, and the possible range of values for each of these fields in the currently active query. These names, types and values are then passed on to the created QEDialog.

Note that the ranges for all these fields are defined by the initial ranges returned by the query as set in the Query Composer. It is only possible to select a sub-range from the chosen range. It is not possible to add values from outside that initial range. If you want to increase the initial selection range, you will have to do that by modifying the query in the Composer.

5.2.2 QEDialog

The QEDialog class provides a Dialog window, based on the ET++ class `Dialog`. It is used by the QEController class to allow the user to vary the range of elements returned by the currently specified Query. The Dialog window is shown in figure 5.1. Different kinds of entry gadgets are used for different types of data. The QEDialog is currently able to handle boolean values, text values, numerical (floating point) values, point values and line values. For `TextItem` type objects, you can either enter one specific value, a constant, or you can enter a string containing wildcards to allow selection of a multitude of strings that match the entered pattern. Furthermore, buttons are available for defining whether certain boolean values should be TRUE or FALSE. It is also possible to allow both TRUE and FALSE values, in which case the value of that button doesn't influence the query, thus can be (is) omitted from the resulting query. For floating point values, point values and line values, `ValSliders` are used (see section 3.2). For point values, two

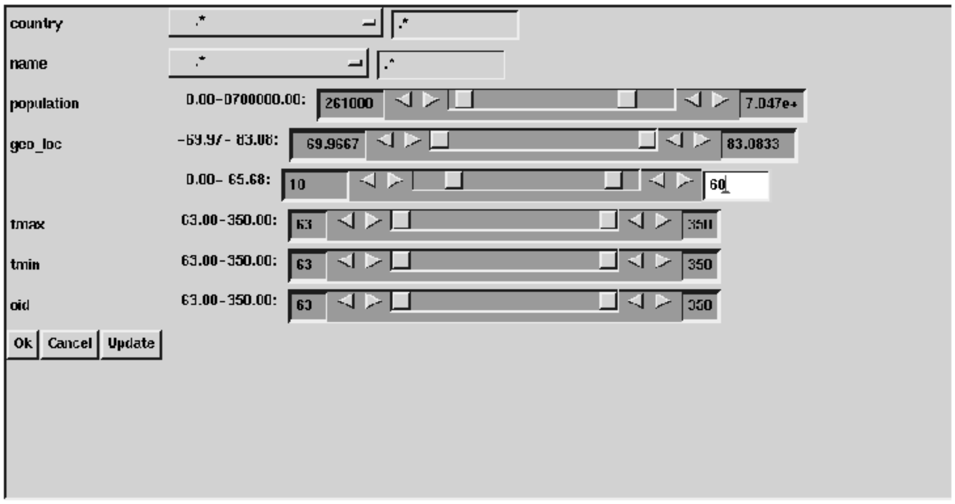


Figure 5.1: The QueryEdit Dialog

ValSliders are used, one for the X value and one for the Y value. And as line value ranges consist of two point value ranges, they require four ValSliders.

Chapter 6

Conclusion

6.1 Evaluation

Despite the tutorial and the programming environment supplied with ET++, getting familiar with the ET++ programs took me quite a bit longer than 2 weeks. This was probably caused by the fact that large parts of the programming environment and several of the programs from the tutorial contained code that didn't work in the new 3.0 alpha version of ET++, and because documentation of the new version was largely absent, so I had to go hunting for the 'bugs' without proper knowledge of ET++. This took up lots of time, but the knowledge acquired this way has aided me furtheron in the project.

In addition, ET++ does not offer sufficient functions for efficiently handling anything else than the user interface. Especially the functions offered to handle data structures, are too few and too slow.

All in all, the initial intentions to create a database layer that would be faster than the current interaction with POSTGRES, did not quite succeed. Part of this failure can be blamed on the used ET++ functions, because of the restrictions mentioned above. But another reason is that, when I started this project, I had little knowledge of database systems and no knowledge at all of Geographic Information Systems, and the extensive research required here, as well as the problems mentioned above, took up a lot of the time that could have been dedicated to the implementation of the database level. Other than that, it has to be taken into account that the POSTGRES system is an advanced and highly sophisticated system created by dozens of people with extensive experience in the database programming field, over a period of several years. I doubt that it's smart to expect one person with a background in graphical applications, but with hardly any database experience at all, to replace this DBMS by a faster layer which should also be able to handle spatial data, in such a relatively short period of time.

Another reason why the system is slowed down, is that the program, and especially its ET++ function component, requires a lot of memory, so that despite the consents made to me at the start, even on the system I had available, the paging mechanism active under Unix mapped pages onto secondary storage a lot more frequently than expected. The most obvious solution for this problem is increasing the amount of real memory available on the system. Another solution, although definitely harder to achieve, would be to incorporate a more efficient kind of memory management and paging mechanism into the program itself, as was briefly mentioned in section "Implementing the database level" in chapter 4.

6.2 Future Development

Because of too little time, and because of the problems I encountered, some parts of the program are still not entirely finished and will have to wait for the next person working on this project (if ever). This includes the load and save routines; the program now loads a default set of files, and the user has no possibility to load another set. Also, the user does not have the possibility to save his changes back to disk. When you leave the program, all changes made to the database files are lost! Adding these routines to the program shouldn't be too much work, but it never had a very high priority during the project, which means that it was postponed until more important matters were finished... which didn't exactly work out as expected.

Also, the current layered data-structure has the severe disadvantage that it is used heavily throughout the program, and this data structure will not be easy to change. The fact that the data is stored as character strings at the lowest level - the `DbField` class - can easily be changed into any other type of storage mechanism, but other than that, changing this data structure means you will have to change large parts of the program. Initially I thought that such a layered structure would postpone the decision of the storage mechanism until the final level, but I did not realize that splitting the classes up into instances, and the instances into fields, already is a significant part of the storage mechanism, which severely limits the possible alternatives for data handling. For that reason, changing this data structure into something that is a lot more suited for spatial data handling will not be as easy as I intended it to be.

When looking at the order in which the subqueries are evaluated, it is sometimes possible to speed up the evaluation by making an estimate beforehand of the processing cost of (the order of) these subqueries. A different order of evaluation can sometimes reduce the processing time. However, this would require the maintenance of some kind of meta-database, in which cost estimates for each kind of operation are present. Apart from that aspect, it would also be better to first concentrate on a faster internal storage and management method, and only add this kind of optimizations lateron.

An important feature of the old GEO++ system based on Postgres, was that the user could easily extend or modify the system's functions and datatypes. In the new version of GEO++, based on the Spatial Query Evaluator, this dynamic linking had to be disabled, meaning that at the time being, this powerful feature is no longer available to the user.

There are lots of different aspects to a good GIS architecture, from spatial and aspatial indexing - static or dynamic -, file organization and access control, and even concurrency control, to extensive graphical management features, that constructing a good GIS will usually require a huge development effort, and that, as I already stated above, I think it is virtually impossible to have all this done by a single person without extensive experience in the field, in only a few months time, although I have learned a lot during this period. Still, for that reason it is not surprising that I had to make a lot of concessions to be able to show any results at all, and that the performance of these results is not very satisfying. However, this also shows that I severely underestimated the knowledge and experience required in implementing this system, when I accepted the assignment.

In the meantime, the POSTGRES system has also received significant changes and performance (speed) improvements, so that all in all, I'm not sure that it is even worth the enormous effort of creating such a replacement.

6.3 Conclusion

Given the current situation, seeing the problems and time involved in creating a good and especially fast alternative for the Postgres interface, and seeing that new versions of the Postgres package get faster every time and also come to include some spatial datatypes, I don't know if it will ever really be worthwhile to exchange the Postgres interface in GEO++ with a built-in database level. It will definitely require a lot of time, and several as yet unsolved problems will have to be taken care of. It may be possible though, to add an internal database level that is only used for implementing the Spatial Query Evaluator tasks and speeding them up, without removing the Postgres interface for general query evaluation.

Bibliography

[1] Vijlbrief, T. & van Oosterom, P.
The GEO++ System: An Extensible GIS
Paper

[2] van Oosterom, P. & Vijlbrief, T.
Building a GIS on top of the open DBMS "POSTGRES"
Proceedings EGIS '91, Second European Conference on Geographical Information Systems, pp. 775-787

[3] Frei, C. & Schaudt, H.
ET++ Tutorial - Eine Einführung in das Application Framework
Paper, Bern, 1991

[4] Stonebraker, M., Rowe, L. A. & Hirohama, M.
The Implementation of POSTGRES
IEEE Transactions on Knowledge and Data Engineering, 2(1), 125-142
1990

[5] Beng Chin Ooi
Efficient Query Processing in Geographic Information Systems
Edited by G. Goos and J. Hartmanis
Springer Verlag
Berlin Heidelberg 1990
ISBN 0-387-53474-1

[6] Stroustrup, B.
The C++ Programming Language, Second Edition
Addison Wesley
Massachussets, 1992

[7] Weinand, A., Gamma, E. & Marty, R.
ET++ - An Object Oriented Application Framework in C++
OOPSLA '88, pp. 46-57

[8] Weinand, A., Gamma, E. & Marty, R.
Design and Implementation of ET++, a Seamless Object-Oriented Application Framework
Structured Programming, 10(2), pp. 63-87

[9] Kuijf, S.
GEO & Grass, Geographic Information Systems
Paper

[10] Vijlbrief, T.
The GEO System, Version 1.33
Paper, '92

[11] van Oosterom, P.
The Reactive-tree - A Storage Structure for a Seamless, Scaleless Geographic Database
In Auto-Carte 10, 1991, pp. 393-407

[12] Unisys Corporation
ARGIS 4GE, 1989.

[13] ESRI
ARC/INFO - The Georelational Model Revisited
Arc News Winter, 11(1), 1989, pp. 9.

[14] Morehouse, S.
The Architecture of Arc/Info
Auto-Carto 9, 1989, pp. 266-277.

[15] Intergraph
MGE - The Modular GIS Environment
Brochure, 1990.

[16] Abel, D.J.
SIRO-DBMS: a database tool-kit for Geographical Information Systems
International Journal of Geographical Information Systems, 3(2), 1989, pp. 103-116.

[17] Waugh, T.C. & Healey, R.G.
The GEOVIEW design
A relational data base approach to geographical data handling.
International Journal of Geographical Information Systems, 1(2), 1987, pp. 101-118.

[18] Pedersen, L.-O. & Spooner, R.
Data Organisation in System 9.
WILD Heerbrugg, 1988.

[19] Herring, J.R.
TIGRIS: Topologically Integrated Geographical Information System.
Auto-Carto 8, 1987, pp. 282-291.

[20] Bennis, K., David, B., Quilio, I. & Viémont, Y.
GéoTropics Database Support Alternatives for Geographic Applications.
4th International Symposium on Spatial Data Handling
Zürich, Switzerland, 1990, pp. 599-610.

[21] Finkley, R.A., Bentley, J.L.
Quad Trees: A data structure for retrieval on composite keys.
Acta informatica 4, 1974, pp 1-9.

[22] Knuth, D.E.
Fundamental Algorithms, 2nd Edition
The art of computer programming, vol. 1
Reading, MA: Addison-Wesley, 1973.

[23] Fussell, D., Kedem, Z.M., Silberschatz, A.
Deadlock removal using partial rollback in database systems.
Proc. ACM SIGMOD International Conf. on management of data
Ann Harbor, Michigan, 1981, pp. 65-73.

[24] Samet, H.
The quadtree and related hierarchical data structures.
ACM Comp. Surveys 16, 1984, pp. 187-260.

[25] Shaffer, C.A.
Application of alternative quadtree representation.
MCS-83-02118, Ph.D. Thesis, Center for Automation Research, Uni. of Maryland, College Park, 1986.

[26] Bentley, J.L.
Multidimensional binary search trees used for associative searching.
Comm. ACM 18, 9, 1975, pp. 509-517.

[27] Guttman, A.
R-trees: A dynamic index structure for spatial searching.
Proc. ACM SIGMOD Int. Conf. on Management of Data
Boston, MA, 1984, pp. 47-57.

[28] Bayer, R., McCreight, E.
Organization and maintenance of large ordered indices.
Acta Informatica 1, 3, 1972, pp. 173-189.

[29] Comer, D.
The ubiquitous B-Tree.
ACM Comp. Surveys 11, 2, 1979, pp. 121-137.

[30] Sellis, T., Roussopoulos, N., Faloutsos, C.
The R^+-tree: A dynamic index for multi-dimensional objects.
Proc. 13th Int. Conf. Very Large Data Bases, Brighton, England, 1987, pp. 507-518.

[31] Matsuyama, T., Hao, L.V. & Nagao, M.
A file organization for geographic information systems based on spatial proximity.
Int. Journal Comp. Vision, Graphics, and Image Processing 26, 3, 1984, pp. 303-318.

[32] Hinrichs, K., & Nievergelt, J.
The grid file: A data structure designed to support proximity queries on spatial objects.
Proc. International Workshop on Graphtheoretic Concepts in Comp. Science.
Trauner-Verlag, 1983, pp. 100-113.

[33] Nievergelt, J., Hinterberger, H. & Sevcik, K.C.
The grid file: An adaptable, sysmetric multikey file structure.
ACM Trans. on Database Syss. 9, 1, 1984, pp. 38-71.

[34] Tamminen, M.
Efficient spatial access to a data base.
Proc. SIGMOD International Conf. on the Management for Data, ACM, 1982, pp. 200-206.

[35] Tamminen, M.
The extendible cell method for closest points problems.
BIT 22, 1982, pp. 27-41.

[36] Sacks-Davis, R., McDonnell, K.J. & Ooi, B.C.
GEOQL - A query language for geographic information systems.
Australian and New Zealand Association for the Advancement of Science Congress, Townsville Australia, 1987 (also appears as Tech. Rep. 87/2, Dept. of Computer Sc. R.M.I.T. Melbourne).

[37] Tamminen, M. & Sulonen, R.
The EXCELL method for efficient geometric access to data.
Proc. ACM IEEE 19th Design Automation Conf., Las Vegas, Nevada, 1982, pp. 345-351.

[38] Oracle overview and introduction to SQL.
Oracle Corporation, U.S.A., 1985.

[39] Held, G.D., Stonebraker, M.R. & Wong, E.
INGRES - A relational data base management system.
Proc. AFIPS National Comp. Conf. 44, 1975, pp. 409-416.

[40] Stonebraker, M. et al.
The Design and Implementation of INGRES
ACM-TODS, September 1976.

[41] Sordi, J.
IBM Database 2: The Query Management Facility
IBM Systems Journal, February 1984.

[42] Robinson. J.
The K-D-B Tree: A Search Structure for Large Multidimensional Indexes.
Proc. ACM-SIGMOD Conference on Management of Data, Ann Arbor, Michigan, 1981.

[43] Schmucker, K.J.
Object Oriented Programming for the Macintosh.
Hayden, Hasbrouch Heights, New Jersey, 1986.

Appendix A

File System

This appendix presents a listing of the programs written for this project with a short description.

A.1 Source Files

The largest set of source files is those for the new database layer. They are: **DataColl.C, DataManager.C, ClassColl.C, DbField.C, Functions.C, Instance.C, ObjPoint.C, ObjLine.C, OperTypes.h** and **SKDTree.C**. A detailed description of the functions of these files can be found in chapter 4.

The remaining files deal with the Spatial Query Evaluator on interface level. These files are:
QEController.C, QEDialog.C, QueryEdit.C and **ValSlider.C**.
The functions of these files are described in more detail in chapter 5.

A.2 Data Files

In the 'geodata' directory, a set of data files is stored. These are:

Defs contains the name of the database on the first line, followed by a series of lines that contain the classname (eg. "**cities**") and a list of attributes with their associating types. The program expects a series of .dat files in the same directory - one for each class. These files have the classname as filename, with the .dat extension appended to them. The .dat files contain the actual data stored in the classes. Each line contains one record with values for all the types available in that class. The values are separated by tabs or a space character. By default, the following .dat files are supplied:

- **cities.dat** contains all the cities across the world.

- **cities1.dat** contains only those cities within the Netherlands. This is a subset of the data from **cities.dat**
- **col1.dat** contains a list of colour settings.
- **dynamic.dat** contains some data for dynamic loading. This is currently not used.
- **europe.dat** contains a list of all the cities in Europe. This is a subset of the data from **cities.dat**
- **geo_active_apr.dat** is currently empty.
- **geo_ap.dat** is currently empty.
- **geo_colors.dat** is currently empty.
- **geo_dyninfo.dat** is currently empty.
- **geo_icons.dat** is currently empty.
- **geo_menu.dat** is currently empty.
- **geo_monitors.dat** is currently empty.
- **geo_predefs.dat** is currently empty.
- **geo_regexp.dat** is currently empty.
- **heights.dat** is currently empty.
- **icon1.dat** is currently empty.
- **o2.dat** is currently empty.
- **o3.dat** is currently empty.
- **o4.dat** is currently empty.
- **tables.dat** is currently empty.
- **width.dat** is currently empty.