EGIS '91. Brussels, Belgium, April 2–5, 1991.

Building a GIS on top of the open DBMS "Postgres"

Peter van Oosterom
TNO Physics and Electronics Laboratory,
P.O. Box 96864, 2509 JG The Hague, The Netherlands.
Email: oosterom@fel.tno.nl
and
Tom Vijlbrief
TNO Institute for Perception,
P.O. Box 23, 3769 ZG Soesterberg, The Netherlands.
Email: tom@izf.tno.nl

Many commercial Geographic Information Systems have a *dual architecture:* the thematic information is stored in a relational database management system and the spatial information is stored in a separate subsystem capable of dealing with spatial data and spatial queries. Besides not being elegant conceptually, this dual architecture also reduces the performance, because objects have to be retrieved and compiled from components that may be stored far apart in the two subsystems. We present a solution based on the extendable database management system "Postgres," in which thematic and spatial data are stored together.

1 Introduction

Many of today's information systems are built around a relational database (RDBMS). The geographic nature of the data in Geographic Information Systems (GISs) results in a number of problems for an RDBMS. It is impossible to state queries like: "Which cities with a population greater than 100,000 lie within 10 kilometers from the river Rhine" in the data manipulation language of an RDBMS. Even if it were possible to state this type of queries, then they could not be answered efficiently, because an RDBMS lacks the proper multidimensional index structures. Therefore, the storage system often consists of two major subsystems. We will call this the *dual architecture.* The first subsystem is based on an RDBMS and contains the alphanumeric thematic data. The second subsystem is a special purpose storage system, capable of dealing with spatial data. An object, that has both a thematic and a spatial component, has parts in both subsystems which are linked by *common identifier*. In order to retrieve one object, two subsystems have to be queried and the answer has to be composed. Many commercial GISs have chosen this inefficient and inelegant dual architecture.

We investigate a data model that does not have this dual architecture. We have chosen the relational data model as our starting point, because it is a powerful and well-known model. The required extension to the relational data model is based the incorporation of [30]: geographic data types (e.g., line, polygon), spatial operators (e.g., distance, overlap, nearest-neighbor), and spatial index structures (e.g., R-tree [12], KD2B-tree [29]). However, most DBMSs are *closed*, i.e., it is impossible to add new data types, operators, or index structures. An exception is *Postgres* [23], an experimental DBMS developed at the University of California, Berkeley. We developed a prototype GIS based on Postgres.

Section 2 states the functionality that a GIS demands from a DBMS. Besides the functional aspects, this paper also pays attention to performance aspects of a solution. Our prototype GIS, called the *GEO system*, is implemented on a Sun 3/60 workstation. Before describing this system in Section 4, a short introduction to the most important aspects of Postgres is given in Section 3. This paper is concluded in Section 5 with some final remarks and suggestions for future research and development.

2 Requirements of a DBMS demanded by a GIS

GISs require more capabilities from a DBMS than the traditional business data processing systems. This is also true for other spatial information system, such as CAD systems. The mathematically sound *relational data model* [4] is the starting point for the following discussion. This section enumerates the most important requirements of a DBMS that is used by a GIS. The DBMS must:

- *r1* Store all the data in *one storage system*, that is, both the spatial data and the non-spatial (or thematic) data. This avoids the drawbacks of a dual architecture.
- *r2* Support that tuples of the same relation may have *different sizes*. This is a very common situation in GISs as the number of points per polyline or polygon is not fixed.
- *r3* Represent *Complex objects*. Various geographic objects consist of multiple components. For example, the representation of a city may consist of a collection of roads, buildings, and various other terrain elements. These must be quickly accessible.
- *r4* Enable the *exchange of complete geographic data sets* (including the possible relationships between the different individual entities) between various GIS sites.
- *r5* Support (or be extendable to support) both *raster and vector data*. The vector data can be subdivided in the following three geometric data types: point, polyline, and polygon. There must be two and three dimensional variants of these types; see Subsection 2.1.
- *r6* Support the following three categories of *spatial operators*: geometric calculation operators, topological operators, and spatial comparison operators; see Subsection 2.2.
- *r7* Provide *spatial index structures*. As GISs usually deal with large collections of geographic data, a spatial index structure is indispensable; see Subsection 2.3.
- *r8* Provide *big tuple attributes*, because polylines, for example, may contain thousands of segments.
- *r9* Allow the implementation of an *advanced graphic user-interface* through a direct interface with the programming language C.

These requirements are not independent of each other. For example, the fact that all data types have to be stored in one storage system (*r1*), causes most of the other requirements. Requirements *r5, r6*, and *r7* are explained in more detail in the next three subsections.

2.1 Geographic Data Types

Nearly all geographic data processing [16, 25, 26] is performed with vector, raster, or a combination of these geometric data formats. The vector format has three subtypes: *point, polyline* and *polygon*. The emphasis is on the vector representation, because it allows more flexible object-oriented manipulations, though the raster representation also has advantages; see [27]. Although their representations might be complex, these data types must be regarded as atomic values in the data model.

Figure 1 demonstrates the use of geographic attributes in relations using the Postquel query language; see Subsection 3.2. The "2" behind point, polyline, and polygon indicates that these are two dimensional attributes. Similarly, it is possible to have three dimensional variants of these data types; a "3" behind the type.

2.2 Spatial Operators

This section describes the *basic* spatial operators. More complex operators exist, but do not have to be included in the basic GIS database system. For example, not included are: network calculations, advanced visualization techniques, simulation, and complex geometric calculations, such as the Voronoi diagram, convex hull, or smallest enclosing circle. The polygon-overlay takes two sets of polygons and calculates all intersections, which results in a third set of polygons. Although it is a complex operation, it is used in many GIS applications, and it could be added to the set of standard operators.

Many spatial operators or functions have been described by various authors [2, 11, 13, 14, 19]. We do not claim that our lists of operators are complete, but they should give a good impression of the basic spatial operators. We distinguish three fundamental classes of spatial operators in addition to the more standard operator classes (e.g., comparison, logical, statistical, and set operators) in an RDBMS:

1. *Geometric calculation operators* return a scalar value or a geometric value. Some of the most important operators are: `distance`, `length`, `perimeter`, `area`, `closest`, `intersection`, and `union`.

2. *Topological operators* return a geometric value. Some examples: neighbors, next link (in a polyline network), left and right polygons of a polyline, start and end nodes of polylines. We do not need an extra set of topological operators, because the topological model can be captured in a natural manner in the standard relational model; see [30].

3. *Spatial comparison operators* return a Boolean: `true` or `false`. Although the calculations are similar to the previous classes of operators, they form a separate group. Some examples are: `intersects`, `inside`, `larger_than`, `outside`, `north_of`, `on`, and `neighbor_of`. All comparison operators have two operands, which may be, in most cases, of any geometric type. Note that it is often possible to simulate these operators by combining the standard comparison operators with the geometric or topological operators.

As stated above, there can be other spatial operators that are useful for a specific application. But it is impossible to include them all in this paper. An "open" database is the solution for this problem. If an operator is not available in the set of basic spatial operators, it can be implemented by the user and, after it has been certified, added to the database system. In this way other users also benefit from the new capabilities. Note that organizational actions have to be taken. For example, someone must ensure the clear and unique naming of operators. It is possible to formulate queries using the basic spatial operators as will be demonstrated in Section 3.

2.3 Spatial Indexing Techniques

The B-tree [1], an indexing technique used in many DBMSs, combines several desirable properties. It is a dynamic, height balanced structure, i.e., insertions, deletions, and updates of entries may be alternated with searches. Because of the balanced nature, searches are efficient ($O(\log n)$ time). Also, the nodes in the B-tree are at least half filled. This results in a compact structure.

However, the B-tree is only suited for searching based on one dimensional attributes, such as numbers or strings. Multiple indices on more than one attribute of a relation are possible, but (with current implementations of RDBMSs) only one can be used for solving a query like:

```
retrieve (tower.all)
   where 5 < tower.location.x < 10
   and 12 < tower.location.y < 20.
```

These point queries can be solved efficiently

```
[a] create tower (name = char16, owner = char16, location = point2)
[b] create road (name = char16, shape = polyline2)
[c] create town (name = char16, #inhabitants = int4, shape = polygon2)
```

Figure 1: Relations with geographic attributes

by the *KDB-tree* [18] index structure. The KDB-tree is a KD-tree adapted for secondary storage and can handle point data in any dimension. The KD-tree cannot handle the polyline and polygon data types. In the literature there are several solutions for this problem, such as: the R-tree [5] (see Subsection 3.4), the Field-tree [6, 7], the Cell tree [10], the KD2B-tree, the Sphere-tree [29], and the Reactive-tree [28]. When fine-tuning the application, the proper indexing technique has to be selected.

3 Description of Postgres

This section gives an introduction to the open DBMS Postgres. A more detailed functional description can be found in [22] and several implementation decisions are discussed in [23]. The Postgres reference manual [33] contains all the information required to use the system. Postgres, the successor of Ingres, is a research project directed by Michael Stonebraker at the University of California, Berkeley. The novel characteristic new concepts in Postgres are: support for complex objects, inheritance, user extendability (with new data types, operators and access methods), versions of relations, and support for rules. The later may be used to implementation constraints.

In the next subsections we explain the features that are of interest to GISs. We will illustrate these features with some GIS examples based on the current version of Postgres (version 2.0.3). First, Subsection 3.1 describes the global architecture of Postgres. The next subsection describes the query language Postquel and Subsection 3.3 gives an example with a user defined type. In Subsection 3.4 the spatial access method that is available within Postgres, the R-tree, is described.

3.1 The Architecture of Postgres

Postgres can be viewed as a collection of files and processes that operate on these files. The files contain the relations and data required for the access methods, that is, the B-tree or the R-tree itself. A daemon process *postmaster* handles the communication between the *backend* (the process that does the real DBMS work and is therefore called *Postgres*) and the *frontend* or application. The postmaster starts a backend process for each application that requests the services of Postgres. A standard Postgres application is the *monitor*, an alphanumeric user-interface for Postgres. The user may state Postquel queries and the answers are displayed in a tabular format. New applications can be developed based on Postgres by using the C library functions of *libpq*. This library contains functions to pass the queries to the backend and to interpret the buffers, called *portals*, which are used to return the results. Another way of interacting with Postgres is by using the *fast path*. The fast path makes it possible to call Postgres system functions directly. In this way the query language is bypassed and best performance is achieved by calling the access methods.

3.2 The Query Language Postquel

The query language Postquel is based on three concepts:

- There are three kinds of *data types*: base types (built-in, system, and user), array types (fixed and variable length) and composite types (tuple, set of tuples, and relation).
- The following kinds of *functions* are available: normal functions (C or Postquel), operators (binding of a symbol to a function), aggregate functions (count, sum,

average, min, max, etc.), and inheritable functions.

- *Rules* have the form: "on condition then do action" and they are used to trigger DBMS actions. Subsection 4.4 gives an example.

User defined types, with their own functions and operators, are of particular interest, because these may be used to define the geographic data types. Subsection 3.3 describes the user types in more detail. The database administrator may "upgrade" user types to system types, making them available to each data base created on the system.

The current distribution of Postgres already contains a system type example that approximates a two dimensional variant of the extension with geographic data types we proposed. It consists of the four types: `point, lseg, path`, and `box`. The type `lseg` implements a single line segment. Polylines and polygons may be represented by `path`, which is a variable length array of `lseg`. The special case of a two dimensional axes-parallel rectangle is represented by the type `box`. Some useful functions and operators are provided (test for overlapping boxes, test if a point lies inside a box, the distance between two points), but more are required for a really good geographic extension of the DBMS; see Subsection 2.2.

Some practical GIS example queries show how geographic functions might be used in Postquel; see Figure 2. Query [a] is a "minimum distance" query. This will not work in Postgres version 2, because the aggregate function `min` is not yet implemented. The next query [b] does the same thing, but is formulated more efficiently by using the function `closest`. The last example, query [c], uses the `inside` function in order to retrieve all the names of the towers in Amsterdam.

3.3 Defining a User Type

The example in this subsection defines the new user type *circle*. This example is based on the tutorial distributed with Postgres. It is important to realize that, in Postgres, there is a difference between the *internal* and the *external* representation of a type. The external representation is a character string for user input and output as used in the monitor. In the case of a circle this could be: (center_x, center_y, radius), for example, (0, 0, 1): the unit circle. The internal representation determines how the type is organized in memory, just as in the programming language C. Figure 3 shows the C code that defines the internal representation of the new type circle and some C functions for it. Assume that this is stored in the file `circle.c`.

The functions `circle_in` and `circle_out` perform the conversions between external and internal representations. There is also one *operator* function for this type: `circle_area_greater`, which determines whether the area of the first circle is greater than the area of the second one. After compiling, which produces the object file `circle.o`, Postgres must be informed about the existence of the new type and its functions: first the conversion functions are defined (see Figure 4 queries [a,b]), then the new type [c] is defined, and finally the operator function [d] and it's symbolic representation [e] are defined; an ">" sign. Now it is possible to create relations with circles in them [f], append records to them [g,h,i], and retrieve the circles which have an area greater than the circle (0,0,8) [j]. The result of the last query [j] is of course the circle (5,1,9).

3.4 The R-tree

The *R-tree* was defined by Guttman [12] in 1984. The leaf nodes of the R-tree contain entries of the form: *(I,object-identifier)*, where *object-identifier* is a pointer to a data object and *I* is a bounding box (or Minimal Bounding Rectangle, *MBR*). The internal nodes contain entries of the form: *(I,child-pointer)*, where *child-pointer* is a pointer to a child node and *I* is the *MBR* of that child. The maximum number of entries in each node is called the *branching factor M* and is chosen to suit paging and disk I/O buffering. The *insert* and *delete* algorithms of Guttman assure that the tree is balanced (all leaf nodes are on the same

```
[a] retrieve min(distance(tower.location, "(10,15)"::point2))
[b] retrieve closest(tower.location, "(10,15)"::point2))
[c] retrieve (tower.name)
      where inside((retrieve (town.polygon)
                    where town.name = "Amsterdam"), tower.location)
```

Figure 2: Postquel queries using geographic functions

```
#include <stdio.h>
typedef struct { double x, y; } POINT;
typedef struct { POINT center; double radius; } CIRCLE; /* Internal repr */

CIRCLE *circle_in(str) /* Convert from external to internal representation */
char *str;
{/* Allocate new CIRCLE, parse string, return result. */}

char *circle_out(circle) /* Convert from internal to external representation */
CIRCLE *circle;
{
   char *result;
   if (circle == NULL) return(NULL);

   result = (char *) palloc(60);
   (void) sprintf(result, "(%g,%g,%g)",
      circle->center.x, circle->center.y, circle->radius);
   return(result);
}

char circle_area_greater(circle1, circle2)
CIRCLE *circle1, *circle2;
{ return(circle1->radius > circle2->radius); }
```

Figure 3: A part of the C source code defining the new type `circle`

```
[a] define C function circle_in (file = "circle.o",
      returntype = circle) arg is (char16)
[b] define C function circle_out (file = "circle.o",
      returntype = char16) arg is (circle)
[c] define type circle (internallength = 24, input = circle_in,
      output = circle_out)
[d] define C function circle_area_greater (file = "circle.o",
      returntype = bool) arg is (circle, circle)
[e] define operator > (arg1 = circle, arg2 = circle,
      procedure = circle_area_greater)
[f] create tutorial (a = circle)
[g] append tutorial (a = "(5,1,9)"::circle)
[h] append tutorial (a = "(2,2,5)"::circle)
[i] append tutorial (a = "(0,1,7)"::circle)
[j] retrieve (tutorial.all) where tutorial.a > "(0,0,8)"::circle
```

Figure 4: The Postquel part of defining the new type `circle`

level) and that the number of entries in each node lies between m and M, where $m \leq M/2$ is the minimum number of entries per node. Figure 5 shows an R-tree with two levels and *M=4*. The lowest level contains three leaf nodes and the highest level contains one node with pointers and *MBRs* of the leaf nodes.

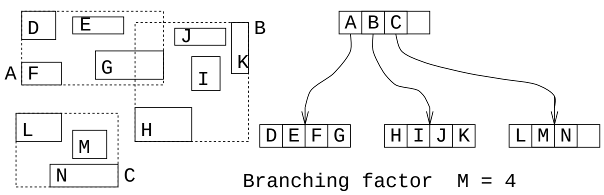

Figure 5: The R-tree

The information in the remainder of this section is based on the *beta* version of the R-tree in Postgres and is provided by Mike Olson[17]. The R-tree will be included in the next public release of Postgres (version 2.1). The performance tests with the R-tree of Postgres are done on DECstation 5000/200 under Ultrix 4.0. The use of the R-tree is similar to the use of the B-tree in Postgres. That is, one can use the Postquel construct `define index` to define an index on the `region` attribute of the relation `testrel`; see Figure 6 [a,d]. The R-tree can be used with system type box and the following operators:

operator	**meaning**
`a << b`	box `a` is strictly left of box `b`
`a &< b`	`a` is left of `b`, or overlaps `b`, but
	does not extend to the right of `b`
`a && b`	`a` overlaps `b`
`a &> b`	`a` is right of `b`, or overlaps `b`, but
	does not extend to the left of `b`
`a >> b`	`a` is strictly right of `b`
`a @ b`	`a` is contained by `b`
`a ~ b`	`a` contains `b`
`a ~= b`	`a` and `b` are the same box

The relation `testrel` is populated (Figure 6 [b]) with 30,000 rectangles, sides random between 0 and 1000, and origins random distributed in three regions 10,000 in (0, 0, 10000, 10000), 10,000 in (30000, 10000, 50000, 30000), and 10,000 in (0, 0, 50000, 50000). This data set is chosen, because it is representative for map data: objects of different sizes and a population density that is not constant over the whole region. Figure 6 [c] shows a rectangle *overlap* query. This is an important type of query, because it is used to generate maps on the rectangular screen. A *point* query, used for implementing a "pick" operation, can be formulated by taking a box with equal diagonal corner points.

A restrictive spatial query, that retrieves up to 100 objects, without the R-tree takes about 85 seconds using a *sequential scan*. Building an R-tree index on the test relation with 30,000 objects takes about 35 minutes. However, now the same spatial queries run typically a *few hundred* times faster using the *index scan*. As the size of the relation grows, the gain of the index-scan will become larger and larger compared to the sequential scan.

The size of the file that contains the `testrel` is 3.0 Mb. The size of the file that contains the index `testind` is 6.7 Mb. This may seem a lot in comparison the relation `testrel`, but in the case of GIS-application with tuples that have polygon attributes varying from 10 to 1,000 points, the overhead of the R-tree is quite acceptable.

4 Implementation of a GIS on top of Postgres

We have built the *GEO system*, a general purpose GIS frontend for Postgres. The system has a "direct manipulation user-interface," allows us to implement real world GIS systems, and allows us to experiment with the user-interface and various data structures and storage techniques. Some of the expected applications are: electronic sea-maps and various Command and Control (C2) systems. The current prototype system is written in C++ [24] and uses the ET++ [31, 32] class library.

4.1 ET++

ET++ is a C++ class library, written by Andre Weinand, Erich Gamma and Rudolf Marty of the University of Zürich. The library consists of a Smalltalk-like collection of classes, just as Keith Gorlen's NIH[1] [9] class

[1] The original name of NIH was OOPS.

```
[a] create testrel (region = box)
[b] append testrel /* append lots of tuples */
[c] retrieve (testrel.all) where testrel.region && "(98,20,9,10)"::box
[d] define index testind on testrel using rtree (region box_ops)
```

Figure 6: Defining an R-tree index for the relation `testrel`

library. The library further contains graphic user-interface building blocks in a manner similar to Interviews. In contrast to NIH and Interviews, ET++ contains both in an integrated design.

An executable ET++ program attaches automatically to the actually used Window System: X11 [8, 20], NeWS, or SunView. The visual appearance of the program is the same in each of the window systems. The running program also behaves the same. This is possible because ET++ defers calling of the actual underlying window system till the lowest level of drawing lines and pixels. The fonts used and the layout and interaction of the user-interface building blocks give it a Macintosh appearance. This is no coincidence, since the authors based many of their ideas on the MacApp framework [21]. In the context of user-interface design, ET++ has the advantage over most other graphic user-interface toolkits (SunView, XView, Xt-widgets) that it enables the designer to change every aspect of the visual appearance and "look and feel" by overruling the appropriate methods in the C++ class inheritance framework. This allows us to implement our own ideas of the ideal "look and feel" without rewriting most of the toolkit. Other features which can be built with with minimal programming effort using ET++ , are:

- A Smalltalk-like class hierarchy and source *browser*.
- *Dynamic* loading.
- *Generic object I/O*.
- *Cut and Paste* between applications.
- Multiple *panes*: a window can be split into two or four sections which show different portions of the image in the window. Each pane has individual scrollbars.
- Generation of a PostScript file, containing the contents of a window. The output does not result in a bitmap image, but uses the full resolution of the PostScript device.

4.2 The GEO system

The GEO system has the following features:

1. A special purpose *data structure layer* (currently our own R-tree implementation). The layer allows the display of geographic data not stored in Postgres because query asking functionality is not required. This special purpose database is fast and can be used for map background data, such as landmass contour data, rivers, etc. When the R-tree becomes available within Postgres and proves to be efficient, then the special purpose data structure layer will be removed.

2. A *Postgres layer* which enables the end-user to formulate queries by using a "direct manipulation user-interface," in contrast to typing postquel queries. In case of geographic data, the results of these queries are displayed as labeled points or polylines on the map. It is also possible to present the results in a tabular format. Further, this module allows the user to modify Postgres data with a "direct manipulation user-interface." The Postgres layer is discussed in more detail in Subsection 4.3.

3. An *annotation layer*. This layer has drawing capabilities as found in many drawing programs (text, polylines with arrows, polygons, etc.) and allows the exchange of information between the users of different networked workstations. Users can view and edit different annota-

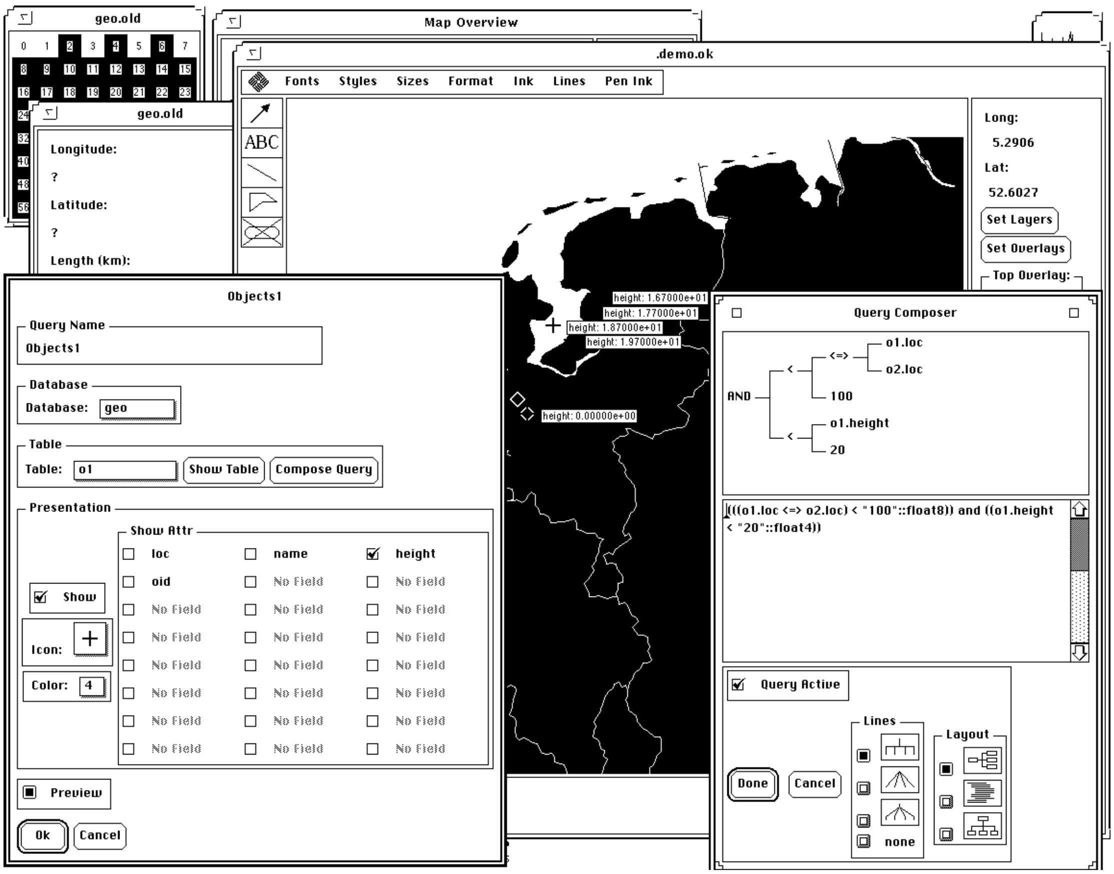

Figure 7: The user-interface of the GEO system, showing a composed query

tion overlays and look at overlays created by others: a briefing.

4. The capability of making changes in Postgres data visible without specific user actions makes it possible to create *dynamic displays* with moving and/or changing objects; see Subsection 4.4.

5. The possibility to *customize the GEO system* by hiding features and options of the system which are not needed for a specific application or adding special functions or icons by editing metadata in the Postgres database.

In the current system only the features 1–3 have actually been implemented, while features 4 and 5 are under development.

4.3 The Postgres Layer

In the *Postgres layer*, one first specifies the database table and the attributes, which should be retrieved. Then, the selection criteria can be specified by building a tree which represents the "where" clause of the query. The resulting tree is the graphic representation of the parse tree of a "where" clause in the Postquel query language. The operations that the user can apply to the nodes in this tree are the productions in the context free grammar describing a Postquel "where" clause, although the end-user is probably not (and should not be) aware of this underlying principle. This guarantees that all possible "where" clauses can be specified. This graphic tree building has two advantages:

1. The graphic tree representation makes inherently complex boolean queries easier to understand (parse) for the end-user.

2. It is not possible to formulate queries that result in a syntax error. The system checks the parameter types of functions and operators and guides the user by the selection of the actual parameters. For example, if the user chooses a function like `distance`, then he can only select ta-

ble attributes that are of the correct type (`point2`).

Most errors made by the users, however, are semantic and not syntactic. This cognitive aspect deserves future research. The productions (rewriting rules) are:

- Choosing a function or operator, which is used for implementing the restriction.
- Choosing a table attribute, which is an operand for a selected function or operator. This may be an attribute of another table (implementing joins) or of a previously composed query.
- Choosing a constant from a range of types (`bool`, `int4`, `point2`, `polyline2`, `polygon2`, `text`, etc.) as operand. The available types are retrieved from the Postgres system tables.
- Choosing a boolean operator: `and, or`, and `not`. This enables the user to create more complex queries.

The query in Figure 7, which retrieves (and displays) all objects of relation `o1` which have a height less than 20 and whose distance from at least one of the objects in relation `o2` is less than 100 kilometers, could be composed by the following sequence:

1. Choose `operator < (float4,float4)`
2. Choose `attribute o1.height`
3. Choose `constant` and enter 20.
4. Choose `boolean AND`
5. Choose `operator < (float8,float8)`
6. Choose `operator <=> (point,point)`
7. Choose `attribute o1.loc`
8. Choose `attribute o2.loc`
9. Choose `constant` and enter 100.

Note that constants are automatically cast to the correct type, that is, `float4` in line 3 and `float8` in line 9. However, attributes can only be selected if they are of the correct type, that is, `float4` in line 2 and `point2` in line 7 and 8.

Of course, the same query could be composed by applying the productions in another order. For example, start with `operator AND`, choose `operator < (float8,float8)`, etc. The only restriction in the current version is that the operator has to be selected before the operands can be chosen. So one has to know the type of the operands. The next version will apply automatic typecasting (e.g., the system chooses between `< (int4,int4)` and `< (float8,float8)`) and will allow the user to select an operand first and insert an operator later.

4.4 Dynamic Display

The dynamic display capability can be implemented by means of Postgres asynchronous portals which notify applications when a rule has fired. The GEO system could define rules on the relations of interest (relations that should continuously be updated on the display) and will be notified by asynchronous portals when the rules fire.

An alternative is to have a Postgres relation containing the *changes* made to the relations of interest, and user defined Postgres *rules* that append data to this relation; see Figure 8 [a,b]. The advantage of this approach is that the user has more control over the type of updates, which should cause a update of the display. This is due to the fact that he (and not the GEO system) defines the rules with the appropriate restrictions. The GEO system just has to monitor the `changes` relation by means of a defining a single rule on this relation. Examples of three user defined triggers that notify GEO system of changes to relation `o1` are given in Figure 8 [c,d,e]. Rules 8 [c] and 8 [d] cause new and deleted tuples to be (un)displayed. Rule 8 [e] causes objects whose name is changed, to be redisplayed only when they are taller than 20 meters.

The GEO system should remove tuples in `changes` which are older than, for example, 1 minute. The tuples cannot be removed immediately after being processed because more than one GEO system could be monitoring the `changes` relation.

```
[a] create o1 (loc = point2, name = char16, height = float4)
[b] create changes (relname = char16, changedoid = oid)
[c] define rewrite rule o1append is
       on append to o1
       do append changes(relname = "o1", changedoid = current.oid)
[d] define rewrite rule o1delete is
       on delete to o1
       do append changes(relname = "o1", changedoid = current.oid)
[e] define rewrite rule o1change is
       on replace to o1.name
       where current.height > 20
       do append changes(relname = "o1", changedoid = current.oid)
```

Figure 8: The relation `changes` and three triggers on changes to relation `o1`

5 Conclusion

Postgres offers several mechanisms for developing advanced GISs that have not been exploited in GEO system yet. For example, Postgres offers historic data and versions of relations. There are types of GISs in which this plays an important role: C2 systems, GISs monitoring of environment, or GISs visualizing census data. It is obvious that these kind of applications will benefit from the automatic storage of historical data. For example, there is no extra coding required (in the application) to solve the query:

```
retrieve (t.name, t.#inhabitants)
from t in town["1 January 1980"].
```

Applications that require geographic data at multiple scales are another example where the novel mechanisms of Postgres, might offer solutions. We are developing a system that avoids storing redundant data, i.e., do not simply store a separate map for each scale. This system might benefit from a combination of techniques:

- Our intended implementation of the Reactive-tree [28] within Postgres.
- The use of rules to derive small scale maps of large scale databases [15].
- Functions within Postgres are useful for the implementation of procedural map generalization techniques. For example, associated with a polyline or polygon is a line generalization algorithm to reduce the number of points used, when working with small scale maps.
- Composite type attributes (relation, (set of) tuple) can be used for multi-scale representation of a single object. These composite type attributes allow references to other tables, which describe the refinement of objects at a larger scale map [3].

Another important research area deals with the cognitive (user-interface) aspects of a GIS. Tests with real users are necessary to determine what a "good" graphic interface to GISs should look like. It is clear that the direct use of Postquel by end-users is not optimal.

Acknowledgments

Many valuable comments and suggestions on an preliminary version of this paper were made by the following persons: Hans Jense, Carol McCann, Hans Schipper, Paul Strooper, Marcel van Veelen, and Marco Woestenburg. We would also like to thank the Postgres Research Group (University of California at Berkeley) and the developers of ET++ for making their systems available. Special thanks to Mike Olsen (of the Postgres Research Group) for his assistance and testing the performance of the beta-version of the R-tree within Postgres.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1973.

[2] N.S. Chang and K.S. Fu. A relational database system for images. In *Pictorial Information Systems*, volume 80 of *Lecture Notes in Computer Science*, pages 288–321. Springer-Verlag, 1980.

[3] Shi-Kuo Chang and Tosiyasu L. Kunii. Pictorial data-base systems. *Computer (U.S.A.)*, 14(11):13–21, November 1981.

[4] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[5] Christos Faloutsos, Timos Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. *ACM SIGMOD*, 16(3):426–439, December 1987.

[6] André Frank. Storage methods for space related data: The Field-tree. Technical Report Bericht Nr. 71, Eidgenössische Technische Hochschule Zürich, June 1983.

[7] Andrew U. Frank and Renato Barrera. The Field-tree: A data structure for Geographic Information System. In *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, pages 29–44. Lecture Notes in Computer Science 409, Springer Verlag, July 1989.

[8] Jim Gettys, Robert W. Scheifler, and Ron Newman. Xlib – C Language X Interface, X Window System, X Version 11, Release 4. Technical report, Digital Equipment Corporation/ Massachusetts Institute of Technology, 1989.

[9] Keith E. Gorlen. An object-oriented class library for C++ programs. *Software – Practice and Experience*, 17(12):899–922, December 1987.

[10] Oliver Günther. *Efficient Structures for Geometric Data Management*. Number 337 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1988.

[11] R.H. Güting. Geo-relational algebra: A model and query language for geometric database systems. In *Advances in Database Technology – EDBT'88*, pages 506–527, March 1988.

[12] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD*, 13:47–57, 1984.

[13] Thomas Joseph and Alfonso F. Cardenas. PICQUERY: A high level query language for pictorial database management. *IEEE Transactions on Software Engineering*, 14(5):630–638, May 1988.

[14] Sudhakar Menon and Terence R. Smith. A declarative spatial query processor for Geographic Information Systems. *Photogrammetric Engineering and Remote Sensing*, 55(11):1593–1600, November 1989.

[15] Jean-Claude Müller. Rule based generalization: Potentials and impediments. In *4th International Symposium on Spatial Data Handling, Zürich, Switzerland*, pages 317–334, July 1990.

[16] George Nagy and Sharad Wagle. Geographic data processing. *Computer Surveys*, 11(2):139–181, June 1979.

[17] Mike Olson. Postgres Research Group, University of California at Berkeley. Personal Communication, February 1991.

[18] John T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. *ACM SIGMOD*, 10:10–18, 1981.

[19] Nick Roussopoulos, Christos Faloutsos, and Timos Sellis. An efficient pictorial database system for PSQL. *IEEE Transactions on Software Engineering*, 14(5):639–650, May 1988.

[20] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on*

Graphics, 5(2):79–109, April 1986.

[21] K.J. Schmucker. *Object Oriented Programming for the Macintosh.* Hayden, Hasbrouck Heights, New Jersey, 1986.

[22] Michael Stonebraker and Lawrence A. Rowe. The design of Postgres. *ACM SIGMOD*, 15(2):340–355, 1986.

[23] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[24] B. Stroustrup. *The C++ Programming language.* Addison–Wesley, Reading, Mass., 1986.

[25] Peter van Oosterom. Spatial data structures in Geographic Information Systems. In *NCGA's Mapping and Geographic Information Systems, Orlando, Florida*, pages 104–118, September 1988.

[26] Peter van Oosterom. Spatial data structures in Geographic Information Systems. In *Computing Science in The Netherlands*, pages 463–477, 1988.

[27] Peter van Oosterom. *Reactive Data Structures for Geographic Information Systems.* PhD thesis, Department of Computer Science, Leiden University, December 1990.

[28] Peter van Oosterom. The Reactive-tree – A storage structure for a seamless, scaleless geographic database. In *Auto-Carto 10, Baltimore*, March 1991.

[29] Peter van Oosterom and Eric Claassen. Orientation insensitive indexing methods for geometric objects. In *4th International Symposium on Spatial Data Handling, Zürich, Switzerland*, pages 1016–1029, July 1990.

[30] Peter van Oosterom, Marcel van Hekken, and Marco Woestenburg. A geographic extension to the relational data model. In *Geo '89 Symposium, The Hague*, pages 319–333, October 1989.

[31] André Weinand, Erich Gamma, and Rudolf Marty. ET++ – An object oriented application framework in C++. In *OOPSLA'88*, pages 46–57, September 1988.

[32] André Weinand, Erich Gamma, and Rudolf Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[33] Sharon Wensel (editor). The Postgres reference manual. Technical Report Memorandum No. UCB/ERL M88/20 (revised), Electronics Research Laboratory, College of Engineering, April 1989.