Paper submitted to the

International Journal of Geographical Information Systems,

December 1993.

THE DEVELOPMENT OF AN INTERACTIVE MULTI-SCALE GIS*

Peter van Oosterom and Vincent Schenkelaars†

TNO Physics and Electronics Laboratory,

P.O. Box 96864, 2509 JG The Hague, The Netherlands.

Email: {oosterom|schenkelaars}@fel.tno.nl

This paper presents the development of the first Geographic Information System that may be used to manipulate a single dataset at a *very large range of scales* (different detail levels). The design of this multi-scale GIS is fully integrated in the *open DBMS Postgres* and the *open GIS GEO++*. Besides the system design, this paper will also give details of the implementation in the Postgres DBMS environment of three generalization tools: 1. the *BLG-tree* for line and area simplification, 2. the *Reactive-tree* for selection based on importance and location, and 3. the *GAP-tree* for solving problems when using the other two structures for an area partitioning. Together with the geographic frontend, the DBMS forms the flexible basis for the realization of powerful GIS applications. The implementation has been successfully benchmarked with two large datasets: *World Databank II* and *DLMS DFAD*. The response times improve with one to two orders of magnitude.

*Parts of this paper are based on the paper *The Design and Implementation of a Multi-Scale GIS*, which has been awarded the first prize at the EGIS'93 conference, Genoa (van Oosterom & Schenkelaars, 1993).

†Vincent Schenkelaars is a PhD-student at the Erasmus University, Rotterdam.

1 Introduction

The advent of Geographic Information Systems (GISs) has changed the way people use maps or geographic data. The modern information systems enable browsing through geographic data sets by selecting (types of) objects and displaying them at different scales. However, just enlarging the objects when the user zooms in, will result in a poor map. Not only should the objects be enlarged, but they should also be displayed with more detail (because of the higher resolution), and also less significant objects should be displayed.

A simple solution is to store the map at different scales (or level of details). This would introduce redundancy with all related drawbacks: possible inconsistency and increased memory usage. Therefore, the geographic data should be stored in an integrated manner without redundancy, and if required, supported by a special data structure.

Detail levels are closely related to cartographic map generalization techniques. The concept of *on-the-fly* map generalization is very different from the implementation approaches described in the paper of Müller et al.: *batch* and *interactive* generalization (Muller, Weibel, Lagrange, & Salge, 1993). The term batch generalization is used for the process in which a computer gets an input data set and returns an output data set using algorithms, rules, or constraints (Lagrange, Ruas, & Bender, 1993) without the intervention of humans. This in contrast to interactive generalization ("amplified intelligence") in which the user interacts with the computer to produce a generalized map.

On-the-fly map generalization does not produce a second data set, as this would introduce *redundant* data. It tries to create a temporary generalization, e.g., to be displayed on the screen, from one detailed geographic database. The quick responses, required by the interactive users of a GIS, demand the application of specific data structures, because otherwise the generalization would be too slow for reasonable data sets. Besides being suited for map generalization, these data structures must also provide spatial properties; e.g., it must be possible to find all objects within a specified region efficiently. The name of these types of data structures is *reactive data structures* (van Oosterom, 1989, 1991, 1993).

The emphasis is put on the generalization techniques *simplification* and *selection*. The core of the reactive data structure is the "Reactive-tree" (van Oosterom, 1991), a spatial index structure, that also takes care of the selection part of the generalization. The simplification part of the generalization process is supported by the "Binary Line Generalization-tree" (van Oosterom & van den Bos, 1989). When using the Reactive-tree and the BLG-tree for the generalization of an area partitioning, some problems are encountered: gaps may be introduced by omitting small features and mismatches may occur as a result of independent simplification of common boundaries. These problems can be solved by additionally using the "GAP-tree".

The Postgres DBMS (Stonebraker, Rowe, & Hirohama, 1990) is extended with these reactive data structures. Postgres, the successor of Ingres, is a research project directed by Michael Stonebraker at the University of California, Berkeley. The characteristic new concepts in Postgres are: support for complex objects, inheritance, user extensibility (with new data types, operators and access methods), versions of relations, and support for rules. In particular, the extensibility of Postgres is used for the implementation of the reactive data structures. The Postgres reference manual (Postgres Research Group, 1991) contains all the information required to use the system.

Short descriptions of the principles of the BLG-tree and the Reactive-tree are given in Section 2. The GAP-tree is described in Section 3. Postgres related implementation aspects are described in Section 4. The implementation is tested and evaluated with two large data sets. The performance improvements achieved by using the new reactive data structures can be found in Section 5. In the next section, an enhanced version of the Reactive-tree is presented: the *self-adjusting Reactive-tree*. This new version is better suited to deal with all kinds of object distributions among the detail levels. Finally, conclusions are given in Section 7 along with some indications of future work.

2 Reactive Data Structures

The term reactive data structure was introduced in the IDECAP project (Projectgroep, 1982; van den Bos, van Naelten, & Teunissen, 1984). A reactive data structure is defined as a *geometric* data structure with *detail levels*, intended to support the sessions of a user working in an interactive mode. It enables the information system to *react* promptly to user actions. This section gives a short description of two reactive data structures: the *BLG-tree* and the *Reactive-tree*. More details can be found in (van Oosterom, 1991; van Oosterom & van den Bos, 1989). These structures are used for the cartographic generalization techniques simplification and selecting. There is still no support for the other generalization techniques: combination, symbolization, exaggeration, and displacement (Shea & McMaster, 1989).

2.1 BLG-tree

The Binary Line Generalization tree (BLG-tree) is a data structure used for line generalization. Only important objects (represented by polylines) need to be displayed on a small scale map, but without a generalization structure, these polylines are drawn with too much detail. A few well-known structures for supporting line generalization are the *Strip tree* (Ballard, 1981), the *Arc tree* (Günther, 1988), and the *Multi-Scale Line Tree* (Jones & Abraham, 1987). We use another data structure, the BLG-tree, because it is suited for polylines, continuous in detail level, and can be implemented with a simple binary tree.

The algorithm to create a BLG-tree is based on the Douglas-Peucker algorithm (Douglas & Peucker, 1973). The first approximation for a polyline is a line between the first point P_1 and the last point P_n. These points are always necessary, otherwise polygons which are composed of several polylines may not be closed when the end-points are omitted. To acquire the next approximation the point P_k with the greatest distance with respect to the line segment (P_1, P_n) is selected. When the original polyline is represented with these three points P_1, P_k, and P_n, a better approximation is normally found. The point P_k is used to split the polyline into two parts. The same procedure is recursively applied to the polyline parts from P_1 to P_k, and from P_k to P_n. Figure 1 illustrates the BLG-tree creation procedure and the resulting binary tree. This tree is stored for each polyline in order to avoid the expensive execution of the Douglas-Peucker algorithm, each time a line generalization is needed.

2.2 Reactive-tree

The Reactive-tree is based on the R-tree (Guttman, 1984), and has therefore similar properties. The main differences with the R-tree are that the internal nodes of a Reactive-tree can contain both *tree entries* and *object entries*, and the leaf nodes can occur at higher tree levels. An object entry looks like: *(MBR, imp-value, object-id)* where *MBR* is the minimal axes-parallel bounding rectangle, *imp-value* is a positive integer indicating the importance level, and *object-id* contains a reference to the object. The type of an object can be any geometric type; e.g. point, polyline, or polygon.

A basic notion is the *importance* of an object o, which is a function of its type (or role), thematic attributes, and geometric size: $I(o) = f(type, attributes, size)$. For example, the size of an area object could be measured by its area $A(o)$ or perimeter $P(o)$. The weight of the type of the object depends on the application. For example, a city area may be more important than a grassland area in a given application. This could be described with the following importance function (without using the thematic attributes): $I(o) = A(o) * WeightFactor(o)$.

A tree entry looks like *(MBR, imp-value, child-pointer)* where *child-pointer* contains a reference to a subtree, *MBR* is the minimal axes-parallel bounding rectangle of the whole subtree, while *imp-value* ($I(o)$) contains the importance of the child nodes decremented by one. The Reactive-tree has the following properties which must be maintained during insertion and deletion:

1. Each node is a physical disk page which contains between $M/2$ and M entries unless it has no siblings (a *pseudo-root*).
2. All nodes on the same level contain entries with the same importance value. More important entries are stored higher.
3. The root contains at least two entries unless it is also a leaf.

In nodes with both tree entries and object entries, the importance level of the object entries is equal to the importance of the tree entries. The Insert and Delete algorithms which maintain the properties of the Reactive-tree are described in (van Oosterom, 1991). Figure 2 shows a scene with objects and MBRs. Figure 3 shows the resulting Reactive-tree. Tree entries are marked with a circle.

The further one zooms in, the more tree levels must be addressed. Roughly stated, during map generation based on a selection from the Reactive-tree, one should try to choose the required importance value such that a constant number of objects will be selected. This means that if the required region is large only the more important objects should be selected and if the required region is small, then the less important objects must also be selected. The recursive *Search* algorithm starts by accessing the root of the tree. All the object entries in the node are checked for overlap with the search area, if there is overlap, the object is returned. If the importance of the root is larger than the required importance, all the tree entries are checked for overlap. In case overlap exist, the corresponding subtree is descended. This recursive process is continued for all subtrees, until an importance level equal to the required level is reached. Since all objects of the same importance are on the same level, no further search is needed in this case.

3 The GAP-tree

This section first describes an area partitioning as a map basis, and the problems encountered when generalizing this type of map. In Subsection 3.2 the key to the solution is introduced:

the area partitioning hierarchy. The creation of the structure that supports this hierarchy, the GAP-tree, is outlined in Subsection 3.3. Operations on the GAP-tree are described in the last subsection.

3.1 Problems with an Area Partitioning

An area partitioning[1] is a very common structure used as a basis for many maps; e.g. choropleths. Two problems occur when using the techniques described in the previous section:

1. *Simplification:* Applying line generalization to the boundaries of the area features might result in ugly maps because two neighboring area features may now have overlaps and/or gaps. A solution for this problem is to use a topological data structure and apply line generalization to the common edges.
2. *Selection:* Leaving out an area will produce a map with a hole which is of course unacceptable. No obvious solution exists for this problem.

A solution for the second (and also for the first) problem is presented. It is based on a *novel* generalization approach called the *Area Partitioning Hierarchy*, which can be implemented efficiently in a tree structure.

3.2 Area Partitioning Hierarchy

The gap introduced by leaving out one area feature must be filled again. The best results will be obtained by filling the gap with neighboring features. This can be easy in case of so called "islands": the gap will be filled with the surrounding area, but it may be more difficult in other situations.

By taking both the spatial relationships and the importance (see Subsection 2.2) of an area feature into account, an *area partitioning hierarchy* is created. This hierarchy is used to decide which area is removed and also which other area will fill the gap of the removed feature.

3.3 Building the GAP-tree

The polygonal area partitioning is usually stored in a topological data structure with nodes, edges, and faces. The process, described below, for producing the area partitioning hierarchy

[1] In an area partitioning each point in the 2D domain belongs to exactly one of the areas (polygons). That is, there are no overlaps or gaps.

assumes such a topological data structure (Boudriault, 1987; DGIWG, 1992; Molenaar, 1989; Peucker & Chrisman, 1975); see Figure 4. The topological elements have the following attributes and relationships:

- a *node* (or 0-cell) contains its point and a list of references to edges sorted on the angle;
- an *edge* (or 1-cell) contains its polyline, length and references to the left and to the right face;
- a *face* (or 2-cell) contains its *WeightFactor*, area, and a list of sorted and signed references to edges forming the outer boundary and possibly inner boundaries;

Note that this topological data structure contains some redundant information because this enables more efficient processing later on. After the topological data structure has been created, the following steps will produce a structure, called the *Generalized Area Partitioning* (GAP)-tree, which stores the required hierarchy:

1. For each face in the topological data structure an "unconnected empty node in the GAP-tree" is created;
2. Remove the least important area feature a, i.e., with the lowest importance $I(a)$, from the topological data structure;
3. Use a topological data structure to find the neighbors of a and and determine for every neighbor b the length of the common boundary $L(a, b)$
4. Fill the gap by selecting the neighbor b with the highest value of the collapse function: $\text{Collapse}(a, b) = f(L(a, b), \text{CompatibleTypes}(a, b), \text{WeightFactor}(b))$. The function $\text{CompatibleTypes}(a, b)$ determines how close the two feature types of a and b are in the feature classification hierarchy associated with the data set (AdV, 1988; DGIWG, 1992; DMA, 1986). For example, the feature types "tundra" and "trees" are closer together than feature types "tundra" and "industry" in DLMS DFAD (DGIWG, 1992);
5. Store the polygon and other attributes of face a in its node in the GAP-tree and make a link in the tree from parent b to child a;
6. Adjust the topological data structure, importance value $I(b)$, and the length of common boundaries $L(b, c)$ for every neighbor c of the adjusted face b to the new collapsed situation.

Repeat the described steps 2–6 until all features in the topological data structure are at the required importance level (for a certain display operation). This procedure is quite expensive and probably too slow for large data sets to be performed on-the-fly. Therefore, the hierarchy is pre-computed and stored in the GAP-tree. The 2–6 steps are now repeated until only one huge area feature is left, because we can not know what the required importance level will be during the interactive use in a GIS. The last area feature will form the root of the GAP-tree. Further, a priority queue may be used to find out efficiently which face a has the lowest importance value $I(a)$ in step 2 of the procedure.

Figure 5.a shows a scene with a land-use map in the form of an area partitioning. In Figure 5.b the GAP-tree, as computed by the procedure described above, is displayed. Note that a few attributes are shown in Figure 5.b: polygon, area, and perimeter of the final feature in the GAP-tree. The polygon is a real self-contained polygon with coordinates, and it is not a list of references. It is important to realize that this data structure is not redundant with respect to storing the common boundaries between area features. The only exception to this is the situation where a child has a common edge with another child or its parent; see the thick edges in Figure 5.a.

More statistical information has to be obtained on how frequent this situation occurs in real data sets. Islands may be relatively uncommon for many area partitionings, particularly statistical reporting zones and political boundaries. A solution for the "common edges" problem is to store references to these edges, similar to a topological data structure. Care must be taken in order to achieve that these edges are also in the right place in the GAP-tree. Because the tree is traversed (and displayed) in a breadth-first order, the best location for a shared edge is together with the area feature that is on the highest level in the GAP-tree. The child that shares this edge, contains a reference to it. The objects retrieved from the database are buffered (in main memory) and displayed, so references can be traced efficiently. In the case of a common edge between two objects at the same level, the edge is stored in the first one and the second one contains the reference to this edge. A final problem is illustrated in Figure 6 where the neighbor of feature F2 is feature F1. Feature F1, which contains edge E12, falls outside the query region. Therefore, the common edge E12 is not in the buffer. However, the missing edge is not really necessary as it is also completely outside the query region. The area feature F2 can be clipped against the query region.

As can be seen in Figure 5.b, the GAP-tree is a multi-way tree and not a binary tree. Some visual results of the on-the-fly generalization techniques with real data are displayed in Section 5. The additional operations using the GAP-tree, not necessarily related to visualization, are described in the next subsection.

3.4 Operations on the GAP-tree

As a feature in the GAP-tree contains the total generalized area, the actual area A of the polygon has to be corrected for the area of its children with the following formula:

$$A(actual) = A(parent) - \sum_{child \in childeren} A(child)$$

It is important to realize that only one level down the tree has to be visited for this operation and not the whole subtree below the parent node. In a similar way the perimeter P of a polygon can be computed, with the only difference that the perimeter of the children have to be added to the perimeter of the parent. This results in the formula:

$$P(actual) = P(parent) + \sum_{child \in childeren} P(child)$$

This formula for perimeter only works if the children have no edges in common with each other or with the parent. Often the boundaries of the areas in the GAP-tree are indeed non-redundant. This also enables the use of the BLG-tree for simplification of important area features at small-scale maps without producing overlaps or gaps between features. The use of the BLG-tree has a very positive effect on the response times of small-scale maps; see Section 5.

4 Postgres Implementation

In this section we indicate how the implementations of the reactive data structures are linked to the Postgres DBMS. There is a big implementation difference between the BLG-tree and the Reactive-tree, because the BLG-tree is implemented as a part of an abstract data type, while the Reactive-tree is a new access method. The latter is far more difficult to add, because an access method interacts with many (undocumented) parts of the Postgres DBMS. The reader will be spared from the C-code details of the implementation, which can be found in (Schenkelaars, 1992). The implementations of the BLG-tree, Reactive-tree, and GAP-tree are described in Subsections 4.1, 4.2, and 4.3 respectively. An example using these structures is given in Subsection 4.4.

4.1 BLG-tree

The BLG-tree is integrated into the `POLYLINE2` data type (Vijlbrief & van Oosterom, 1992). In Postgres, each data type should at least be provided with two functions: an input function which converts an external ASCII representation into an internal representation, and an output function which translates the internal representation into an ASCII string. The `POLYLINE2` input function is modified to create a BLG-tree and to store this tree along with the defining points of the original line.

The process of retrieving a generalized polyline from a BLG-tree is less complex than the creation process. The output function `Blg2Pln` is provided to retrieve a generalized polyline. It is called with two parameters: the original polyline, and the maximum distance (error) between the original polyline and the generalized polyline. Note that in this manner, the database does not return unnecessary points to the application, which can save a substantial amount of data transfer time. For polygons a similar implementation is created: the `POLYGON2` data type with the output function `Blg2Pgn`.

4.2 Reactive-tree

In order to use the Reactive-tree as a new access method, some meta information must be inserted in the following Postgres system tables: `pg_am`, `pg_proc`, `pg_operator`, `pg_opclass`, `pg_amop`, and `pg_amproc`. An access method has to be registered in the table `pg_am`. This is done with the following query:

```
append pg_am(
  amname = "ReactiveTree",          amowner = "6",
  amstrategies = 8,                 amsupport = 6,
  amgettuple = "reactgettuple",     aminsert = "reactinsert",
  amdelete = "reactdelete",         ambeginscan = "reactbeginscan",
  amrescan = "reactrescan",         amendscan = "reactendscan",
  ammarkpos = "reactmarkpos",       amrestrpos = "reactrestrpos",
  ambuild = "reactbuild")
```

The first attribute contains the name of the new access method (`ReactiveTree`). The second attribute indicates the user-id of the owner of the access method; in this case user-id number 6. In Postgres an access method is based on:

1. A set of (boolean) *access method user functions* to indicate the relative position of two objects; in case of a 2D spatial index this set contains 8 (`amstrategies`) elements. For the Reactive-tree these are called: `Left2ReaRea`, `Overleft2ReaRea`, `Overlap2ReaRea`, `Right2ReaRea`, `Overrgt2ReaRea`, `Equal2ReaRea`, `Contain2ReaRea`, and `ContBy2ReaRea`. For each of these functions an operator symbol has to be defined (in the `pg_operator`

table), which can be manipulated by the query optimizer. These will be called the *access method user operators* and are numbered according to the following sequence `<<`, `&<`, `&&`, `&>`, `>>`, `~=`, `~`, and `@`.

2. A set of *access method procedures*; in case of the Reactive-tree 6 (`amsupport`) functions are needed and numbered in the order: `Union2ReaRea`, `Inter2ReaRea`, `Size2Rea`, `IsObject2Rea`, `MakeTreeObj2Rea`, `React2Imp`.

3. A fixed set of *access method manipulation functions*:

   - `amgettuple` contains the name of the function which performs an index search: `reactgettuple`;
   - `aminsert` contains the function which inserts a tuple in the index: `reactinsert`;
   - `amdelete` contains a function which deletes a tuple from the index: `reactdelete`;
   - `ambeginscan` contains a function which initializes an index scan: `reactbeginscan`;
   - `amrescan` contains a function which rescans a previous started scan: `reactrescan`;
   - `amendscan` contains a function which ends an index scan: `reactendscan`;
   - `ammarkpos` contains a function which marks a position in a scan: `reactmarkpos`;
   - `amrestrpos` contains a function which releases a marked position in the scan: `react-restrpos`;
   - `ambuild` contains a function which initializes the index and add all tuples that are in the relation: `reactbuild`.

All these functions (access method user functions, access method procedures, and access method manipulation functions) must be registered in the `pg_proc` table. This is done, for example for the `reactinsert`, as follows:[2]

```
define function reactbuild (language = "c", returntype = int4)
  as "$REACTHOME/reactree.o"
```

For each data type the user wants to access with the defined access method, an "operator class" has to be defined (in the `pg_opclass` table). An operator class collects both access method procedures (registered in the `pg_amproc` table) and access method operators (registered

[2] One problem arises: the functions often need arguments of types that do not exist as standard Postgres types. For example `reactinsert` needs a `Relation` type. Therefore no arguments, and an int4 as return value, are defined in the function definition. Postgres does no type checking in this case. Only the number of arguments the function expects, has to be provided:

```
define function reactinsert (language = "c", returntype = int4) as "$REACTHOME/reacttree.o"
replace pg_proc (pronargs = 2) where pg_proc.proname = "reactinsert"
```

in pg_amop table).

In order to make sensible use of the Reactive-tree, a data type is required that possesses both a geometric attribute and an importance level. For this purpose a new type was created: REACTIVE2, which contains a bounding box and an importance level.

The following sequence of Postquel queries shows the registration of the operator class Reactive2_op in the pg_opclass table, and examples of adding a operator respectively a procedure to the pg_amop table respectively to the pg_amproc table.

```
append pg_opclass (opcname = "Reactive2_ops")

append pg_amop(
  amopid = am.oid, amopclaid = opc.oid,
  amopopr = opr.oid, amopstrategy = "1"::int2, ...)
  from opc in pg_opclass, opr in pg_operator,
    opt in pg_type, am in pg_am
  where opr.oprname = "<<" and opt.typname = "REACTIVE2" and
    opt.oid = opr.oprright and opt.oid = opr.oprleft and
    am.amname = "ReactiveTree" and opc.opcname = "Reactive2_ops"

append pg_amproc(
  amid = am.oid, amopclaid = opc.oid,
  amproc = proc.oid, amprocnum = "1"::int2)
  from opc in pg_opclass, proc in pg_proc,
    am in pg_am
  where am.amname = "ReactiveTree" and
    opc.opcname = "Reactive2_ops" and proc.proname = "Union2ReaRea"
```

4.3 GAP-tree

A possible implementation difficulty of the GAP-tree is the fact that it is a multi-way tree and not a binary tree. Therefore, a simple linear version has been derived from the GAP-tree by putting the features in a list based on their level in the tree. The top level feature in the tree will be the first element of this list, the second level features will follow, and so on. For example, the linear list for the scene in Figure 5 is: GRASS, FOREST, CORNFIELD, TOWN, LAKE, CENTER, PARK, INDUSTRY, ISLAND, POND. When the polygons are displayed in this order, a good map can be produced without the GAP-tree. However, it is very difficult to compute the actual area without the GAP-tree.

As mentioned before, the on-the-fly generalization has been developed within the Postgres DBMS environment. Postgres is an extensible relational system. A relation does not guarantee any order among its elements. A good display can be obtained by sorting on the sequence number in the list or on the area of the feature.

4.4 Example

In this subsection an example of the use of the Reactive-tree, BLG-tree, and GAP-tree will be given. The Reactive-tree access method can be defined before any tuples are added to the relation, or one can first insert all tuples and decide later that a Reactive-tree is needed. The following two Postquel queries show the definition of the user table `AreaFeature` and the definition of a Reactive-tree index on this table using the `Reactive2_ops` operator class.

```
create AreaFeature (Height=int2, Idcode=int2, Tree=int2,
   Roof=int2, shape=POLYGON2, reactive = REACTIVE2)\g

define index af_index on AreaFeature
   using ReactiveTree (reactive Reactive2_ops)\g
```

Figures 7 and 8 show the DLMS DFAD data set in the "interactive use" of the Reactive-tree, the BLG-tree, and the GAP-tree in the Postgres GIS frontend GEO++ (Vijlbrief & van Oosterom, 1992). Note that no visual loss of information occurs at the top map which is created from the same geographic data set. The GEO++ system automatically generates Postquel queries with the proper values for the BLG-tree and the Reactive-tree depending on the current scale. For the map in Figure 7 the following query is generated:

```
retrieve (blg_pgn2= Blg2Pgn(AreaFeature.shape,"0.01"::float4))
  where AreaFeature.reactive && "(13,40,23,47,2)"::REACTIVE2
  sort by AreaFeature.oid
```

The Postgres query optimizer automatically selects the Reactive-tree access method when evaluating this query. The BLG-tree is used by specifying the function `Blg2Pgn` in the target list of the query. The linearized GAP-tree is reflected by the *sort by* clause of the query.

5 Performance Results

In this section the benchmarks of the reactive data structures are presented. In subsection 5.1 is The World Databank II (WDB II; see Figure 9) (Gorny & Carter, 1987) test described. In subsection 5.2 are the results shown of the tests with the DLMS DFAD[3] data (DMA, 1986) of the former Republic of Yugoslavia presented.

The tests were performed on a Sun SPARCstation II (32Mb main memory) under SunOs 4.1.2. The data was retrieved over a network file system by Postgres. The special test program is a simple Postgres frontend application. The response time was measured with the Unix `time` command. The test program needs a parameter which indicates the size of the search area. This size is used to calculate which importance values are retrieved: thus less important features are retrieved when the area is smaller. Of course, the user could overule the level of importance, as generated by the frontend, in a specific query.

The R-tree has no mechanism to select on importance level, that is why the R-tree retrieves much more objects in the larger areas. The following cases were tested: no index structure, an R-tree index, an R-tree index and a BLG-tree, a Reactive-tree index and a BLG-tree. The test on the DLMS DFAD data set uses the GAP-tree too in the last case. In all test, ten random area queries are executed. The presented response time is in seconds and is the average time over the ten randomly generated queries. The Reactive-tree uses importance levels to reduce the number of selected objects at the more coarse level. Actually, the information density (i.e., the number of displayed features) should remain equal under the varying scales.

5.1 The World Data Bank II Test

The WDB II set is not ideal for the Reactive-tree because the number of objects does not increase enough when a lower importance level is reached. But because of the very large polylines in the dataset, the BLG-tree resulted in a substantial performance improvement. The total number of features in this data set is 38.096.

First the raw data needed to be translated into Postgres tuples. This resulted in a database of approximate 120Mb (116Mb user data, 3Mb index, 1Mb Postgres meta-data).

Queries which selected tuples at four different area sizes were executed. The area sizes varied from 32×32 degrees to 4×4 degrees latitude/longitude. These area sizes were selected because

[3]DLMS (Digital LandMass) DFAD (Digital Feature Analyses Data) Level 1 has a data density which can be compared to 1:200,000 scale map

in each area decrement, one level of detail more is shown. The Reactive-tree in the WDB II test used importance levels form 1 to 4.

5.2 The DLMS DFAD Test

In this section the results of the performance tests of the combined use of the GAP-tree, the Reactive-tree, and the BLG-tree are presented. The DLMS DFAD data of the former Republic of Yugoslavia is used. Only the area features are used in order to evaluate the effectiveness of the GAP-tree.

The visual results of the on-the-fly generalization techniques can be seen in Figures 7 and 8. All maps, including the overview in the upper right, are generated by the same query with only different sized retrieved regions. DLMS DFAD can be regarded as land-use data with over 100 different area classifications, such as: lake, water, trees, sand, swamp, tundra, snow/ice, industry, commercial, recreational, residential, etc. A few notes with respect to the visualization:

1. Many colors on the screen are lost in the gray scales of the printer.
2. As the emphasis is on the GAP-tree, the line and points features have been omitted, resulting in an incomplete map.
3. In the upper right corner of each figure, an overview of the region is shown without the mainland of Italy.
4. Though the DLMS DFAD data is stored in a seamless database, it has been digitized on a map sheet base. During this process similar features have been classified differently; see Figure 7 near the 44 degrees meridian.

The DLMS DFAD test database contains 70.272 area features (requiring about 60Mb) which are given an importance value ranging from 1 to 5, at each level the more detailed level contains about one order of magnitude more features. Queries which selected tuples at five different area sizes were executed. The area sizes varied from 1.6×1.6 degrees to 0.1×0.1 degrees latitude/longitude. The use of the GAP-tree will make sure that the map does not contain gaps when omitting the less important area features.

5.3 Results Analysis

The difference in response time between the Reactive-tree and the R-tree is decreasing when moving to smaller areas, since the difference in the number of returned objects is narrowing.

Tables 1 and 3 show the average number of returned objects for each method. In a similar way, the BLG-tree is more effective when used in a large area search. In that case, a lot of polyline or polygon points can be omitted.

All in all, the Reactive-tree in combination with the BLG-tree gives very good improvements. The process of selection and generalization results in a better map without too much detail. The selection could be done with the R-tree, but in that case a user should change the queries on every scale, in order to select only the desired objects. Also, the important objects would then be located at the leaf level in the tree just as the other objects, which would slow down the small-scale queries. Figures 10 and 11 show the response times of the different methods, while Tables 2 and 4 show the same information in tabular form.

Using no index at all, results in extremely long response times for data sets of the presented size even for relatively small query regions: 205 seconds for WDB II (16 square degrees) and 113 seconds for DLMS DFAD (0.01 square degrees); see Tables 2 and 4. This is due to the fact that a sequential scan over the whole data set has to be performed. Using a spatial index structure enhances the performance for small query regions dramatically as can be seen in Figures 10 and 11.

For very small query regions, using the R-tree and BLG-tree, is even slightly more efficient than using the Reactive-tree and the BLG-tree. The reason for this is the small overhead in using a user-defined access method (Reactive-tree) instead of a Postgres build-in access method (R-tree). The differences are very small. However, the R-tree/BLG-tree response times increase a lot when the query size regions increases. This is caused by the large number of objects which do not need to be retrieved when using the Reactive-tree/BLG-tree. This makes it possible to achieve more or less constant response times irrespective of the size of the query region: always less than 5 seconds for WDB II and always less than 10 seconds for the DLMS DFAD.

If geometrically close objects are guaranteed stored close together on disk, some extra speed may be gained. The Reactive-tree does not give such a guarantee, it depends on the inserting order, whether two geometrically close objects are also stored close on disk. *Clustering* can be used to improve this storage aspect. In Postgres, a simple kind of clustering can be easily simulated by retrieving the objects using a rectangle that contains all the objects. In fact, an access method scan is performed. When the objects are stored in a new relation in the retrieved order, geometrically close objects are stored close together on disk. This reduces the number of disk pages to be fetched for spatial queries and therfore the results will be returned faster.

6 Self-Adjusting Reactive-tree

In this section another type of Reactive-tree is presented: the *self-adjusting Reactive-tree*. A drawback of the normal Reactive-tree, as presented in 2.2, is that there is a one-to-one correspondence between the levels in the tree and the importance values. This is no problem if there is a hierarchical distribution of the data with respect to the importance values and when these values are properly numbered. However, in case the distribution is different, one would still want the Reactive-tree to behave in an efficient manner. The same is true if the user would decide only to use "strangely" numbered importance values; e.g. 1, 10, 15, 50, and 900. The pseudo-roots may cause the tree to be underfull and badly balanced.

The one-to-one correspondence between tree levels and importance values probably will have to be abandoned. It is then possible that objects of different importance are stored in the same node (at the *same* tree level), as long as less important objects are never stored in nodes *above* more important objects. It might also be convenient to store objects of the same importance at different tree levels if this helps to get a well-filled and balanced tree. There are two ways in which this new object ordering may be carried out:

- In a *global* manner: nowhere in the tree is it allowed that a less important object is stored on a higher tree level than a more important object.
- In a *local* manner: if a node contains an object of a certain importance than the sub-tree below this node may not contain more important objects.

It is not difficult to see that if the global ordering has to be satisfied, it is then impossible to always get a well-filled (and balanced) tree. For example, in the case that there is a strange data distribution, such as: a large number of important objects to the left-hand side of the scene and a number of less important objects located to the far right-hand side. Objects which are spatially far apart should not be stored in the same sub-tree. However, if they are stored in different subtrees, then the less important objects would be stored too high unless there is a path of (nearly empty) pseudo-roots.

Therefore, we abandoned the global ordering and tried to design a Reactive-tree which satisfies the less strict local object ordering. In subsequent research we will carry out practical testing with both the normal Reactive-tree and the self-adjusting Reactive-tree for different data sets; both well and badly distributed.

The self-adjusting Reactive-tree satisfies the following defining properties:

1. For each object entry *(MBR, imp-value, object-id)*, *MBR* is the smallest axes-parallel rectangle that geometrically contains the represented object of importance *imp-value*. The most important objects have the lowest importance values.

2. For each tree entry *(MBR, imp-value, child-pointer)*, *MBR* is the smallest axes-parallel rectangle that geometrically contains all rectangles in the child node and *imp-value* is the importance of the child-node adequately decremented.

3. All the nodes on the same level are in the same importance value range. This implies that it is not possible that a subtree contains object entries that are more important (lower importance value) than the object entries in the parent of this subtree.

4. Every node contains between $m(\leq M/2)$ and M object entries and/or tree entries.

5. The importance of the tree entries is less or equal to the importance value of the least important (highest importance value) object entry in every node.

6. The root contains at least 2 entries unless it is a leaf.

The fact that the empty tree satisfies these properties and that the Insert and Delete algorithms (Schenkelaars, 1992) do not destroy them, guarantees that a self-adjusting Reactive-tree always exists.

7 Conclusion

After the Reactive-tree and BLG-tree, the GAP-tree forms a new important step towards the realization of an interactive multi-scale GIS. It is now possible to interactively browse through large geographic data sets. In both the DLMS DFAD database (70.000 features, 60 Mb) and the WDB II database (38.000, 120 Mb) it is now possible to get map displays at any required scale in about five seconds. In the near feature, more tests with datasets are planned; e.g., with the Topographic base map of The Netherlands. An interesting open question is how to assign importance values automatically to features when building the Reactive-tree for a new data set.

A new reactive data structure is presented: the self-adjusting Reactive-tree, which should be even more effective. Future implementation and testing will have to confirm this expectation. Another question which still has to be answered is: When is it better to store the BLG-tree and when is computing a generalized line on the fly preferred? This is a matter of balancing CPU and disk speed.

Further research is also required to determine how the GAP-tree can be maintained efficiently under edit operations. Additional further research topics are: the use of (dynamic) clustering techniques, and the design and implementation of other generalization techniques to support: combination, symbolization, and displacement.

Acknowledgments

We would like to thank the Postgres Research Group (University of California at Berkeley) for making their system available. Special thanks to Tom Vijlbrief for helping us with the GEO++ issues. Many valuable comments and suggestions on a preliminary version of this paper were made by Marcel van Hekken and Paul Strooper.

References

AdV (1988). Amtliches Topographic-Kartographisches Informationssystem (ATKIS). Arbeitsgemeinschaft der Vermessungsverwaltungen der Länder der Bundesrepublik Deutschland (AdV). (in German).

Ballard, D. H. (1981). Strip Trees: A Hierarchical Representation for Curves. *Communications of the ACM*, *24(5)*, 310–321.

Boudriault, G. (1987). Topology in the TIGER File. in *Auto-Carto 8*, pp. 258–269.

DGIWG (1992). DIGEST – Digital Geographic Information – Exchange Standards – Edition 1.1. Defence Mapping Agency, USA, Digital Geographic Information Working Group.

DMA (1986). Product Specifications for Digital Feature Analysis Data (DFAD): Level 1 and Level 2. Defense Mapping Agency, Aerospace Center, St Louis, Mo.

Douglas, D. H., & Peucker, T. K. (1973). Algorithms for the Reduction of Points Required to Represent a Digitized Line or its Caricature. *Canadian Cartographer*, *10*, 112–122.

Gorny, A. J., & Carter, R. (1987). World Data Bank II: General Users Guide. US Central Intelligence Agency.

Günther, O. (1988). *Efficient Structures for Geometric Data Management*. No. 337 in Lecture Notes in Computer Science. Springer-Verlag, Berlin.

Guttman, A. (1984). R-Trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD*, *13*, 47–57.

Jones, C. B., & Abraham, I. M. (1987). Line Generalization in a Global Cartographic Database. *Cartographica*, *24(3)*, 32–45.

Lagrange, J. P., Ruas, A., & Bender, L. (1993). Survey on Generalization. IGN.

Molenaar, M. (1989). Single Valued Vector Maps: A Concept in Geographic Information Systems. *Geo-Informationssysteme*, *2(1)*, 18–26.

Muller, J. C., Weibel, R., Lagrange, J. P., & Salge, F. (1993). Generalization: state of the art and issues. European Science Foundation, GISDATA Task Force on Generalization.

Peucker, T. K., & Chrisman, N. (1975). Cartographic Data Structures. *American Cartographer*, *2(1)*, 55–69.

Postgres Research Group (1991). The Postgres Reference Manual, Version 3.1. Tech. rep. Memorandum, Electronics Research Laboratory, College of Engineering.

Projectgroep (1982). IDECAP Interactief Pictorieel Informatiesysteem voor Demografische en Planologische Toepasingen: Een verkennend en vergelijkend onderzoek. Tech. rep. Publicatiereeks 1982/2, Stichting Studiecentrum voor Vastgoedinformatie te Delft.

Schenkelaars, V. F. (1992). Master's thesis: Implementation of Reactive data structures for Postgres. Tech. rep. FEL-92-S343, FEL-TNO Divisie 2.

Shea, K. S., & McMaster, R. B. (1989). Cartographic Generalization in a Digital Environment: When and How to Generalize. in *Auto-Carto 9*, pp. 56–67.

Stonebraker, M., Rowe, L. A., & Hirohama, M. (1990). The Implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, *2(1)*, 125–142.

van den Bos, J., van Naelten, M., & Teunissen, W. (1984). IDECAP Interactive Pictorial Information System for Demographic and Environmental Planning Applications. *Computer Graphics Forum*, *3*, 91–102.

van Oosterom, P., & Schenkelaars, V. (1993). Design and Implementation of a Multi-Scale GIS. in *Proceedings EGIS'93: Fourth European Conference on Geographical Information Systems*, pp. 712–722. EGIS Foundation.

van Oosterom, P., & van den Bos, J. (1989). An Object-Oriented Approach to the Design of Geographic Information Systems. *Computers & Graphics*, *13(4)*, 409–418.

van Oosterom, P. (1989). A Reactive Data Structure for Geographic Information Systems. in *Auto-Carto 9*, pp. 665–674.

van Oosterom, P. (1991). The Reactive-Tree: A Storage Structure for a Seamless, Scaleless Geographic Database. in *Auto-Carto 10*, pp. 393–407.

van Oosterom, P. (1993). *Reactive Data Structures for Geographic Information Systems*. Oxford University Press, Oxford.

Vijlbrief, T., & van Oosterom, P. (1992). The GEO System: An Extensible GIS. in *Proceedings of the 5th International Symposium on Spatial Data Handling, Charleston, South Carolina*, pp. 40–50 Columbus, OH. International Geographical Union IGU.

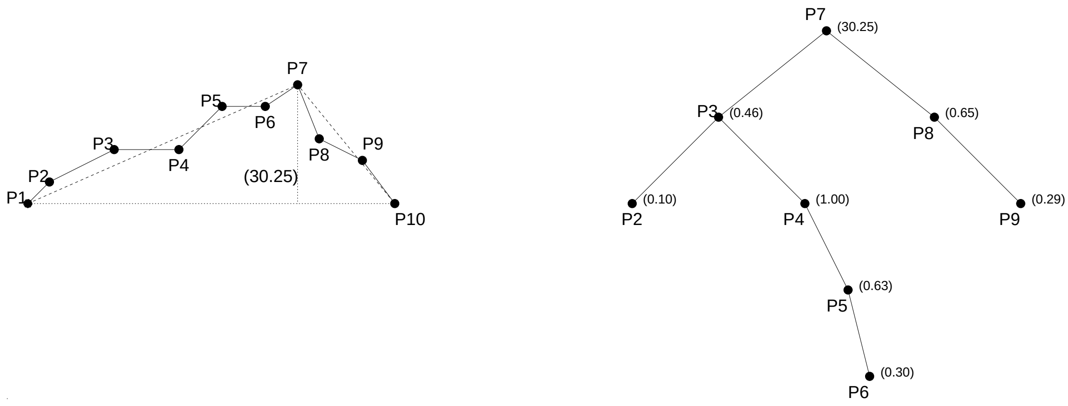

Figure 1: Binary Line Generalization tree

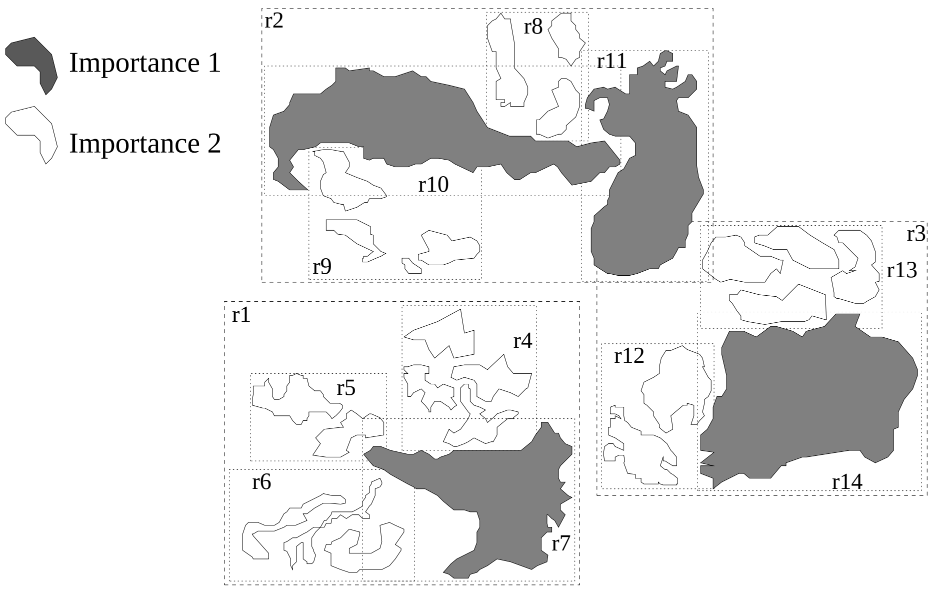

Figure 2: An example of Reactive-tree rectangles

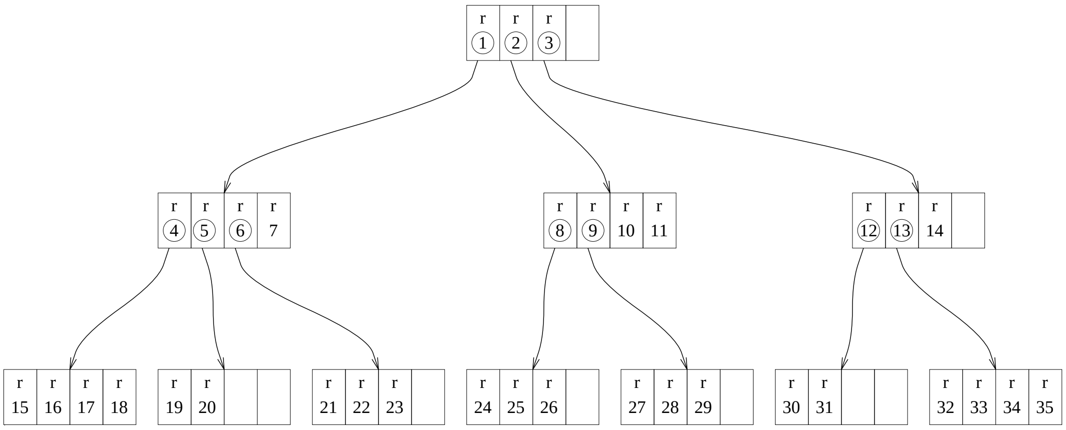

Figure 3: The Reactive-tree

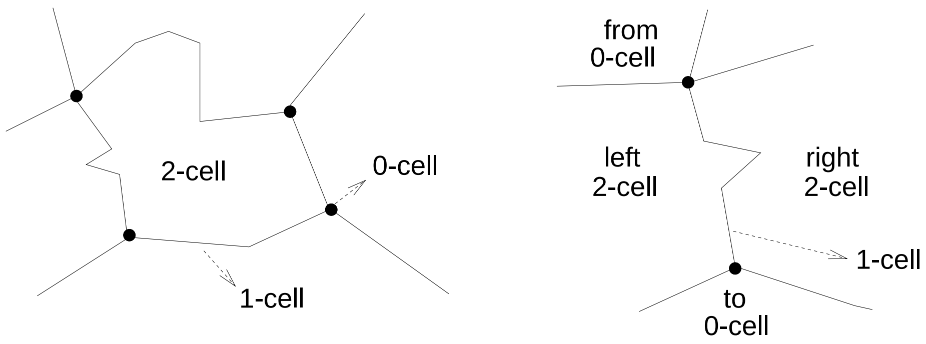

Figure 4: The topological data structure

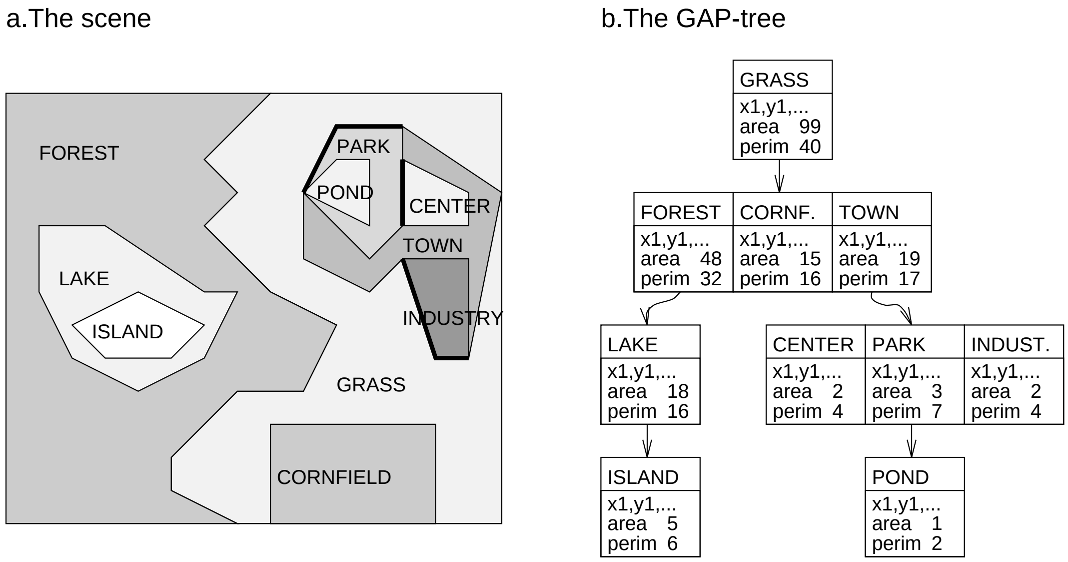

Figure 5: The scene and the associated GAP-tree

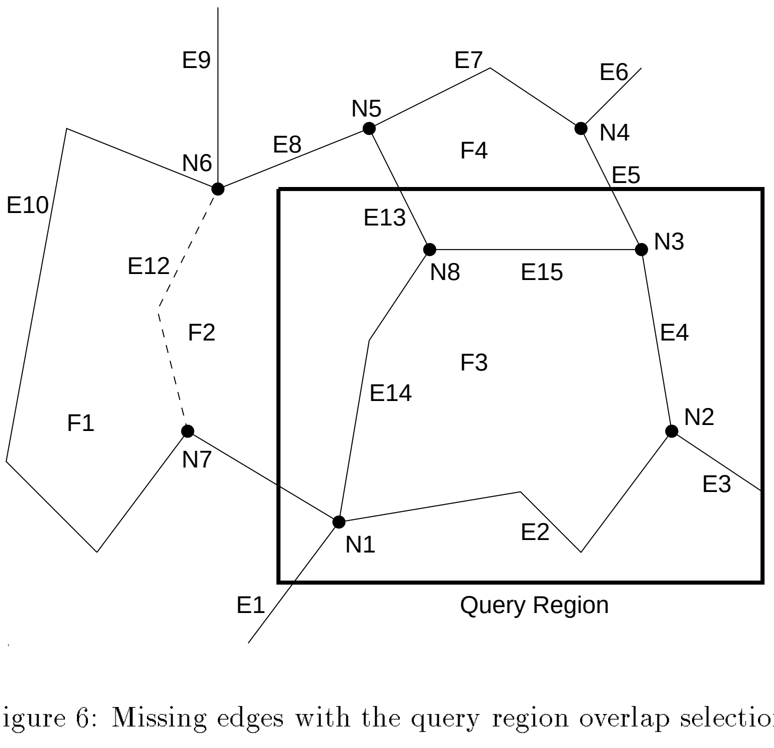

Figure 6: Missing edges with the query region overlap selection

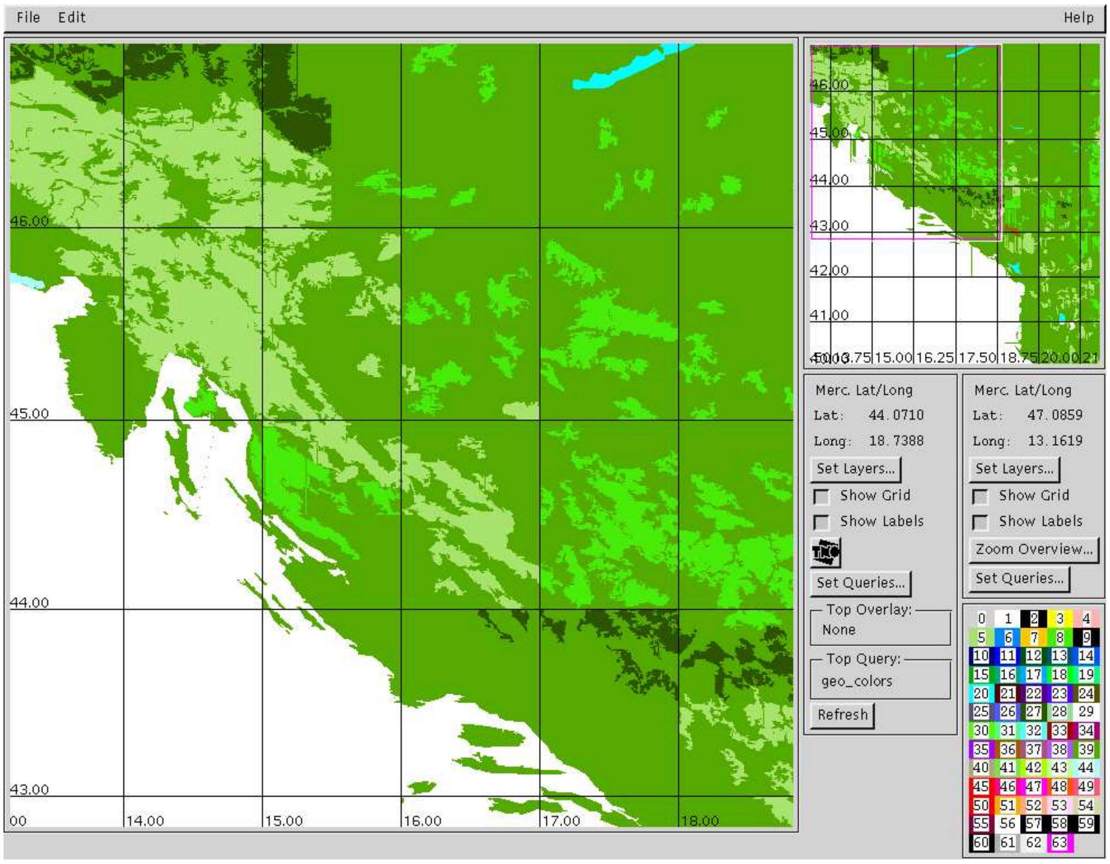

Figure 7: Using the GAP/Reactive-tree and BLG-tree (DLMS DFAD; coarse map)

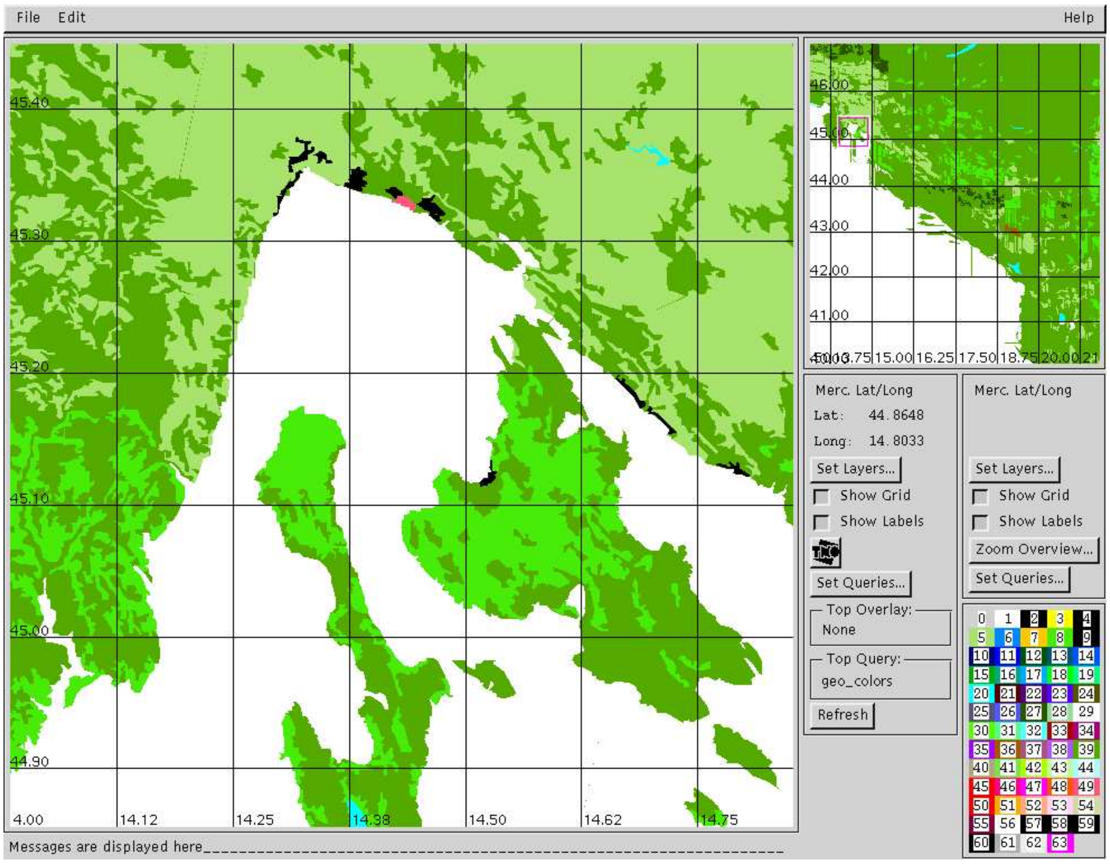

Figure 8: Using the GAP/Reactive-tree and BLG-tree (DLMS DFAD; detail map)

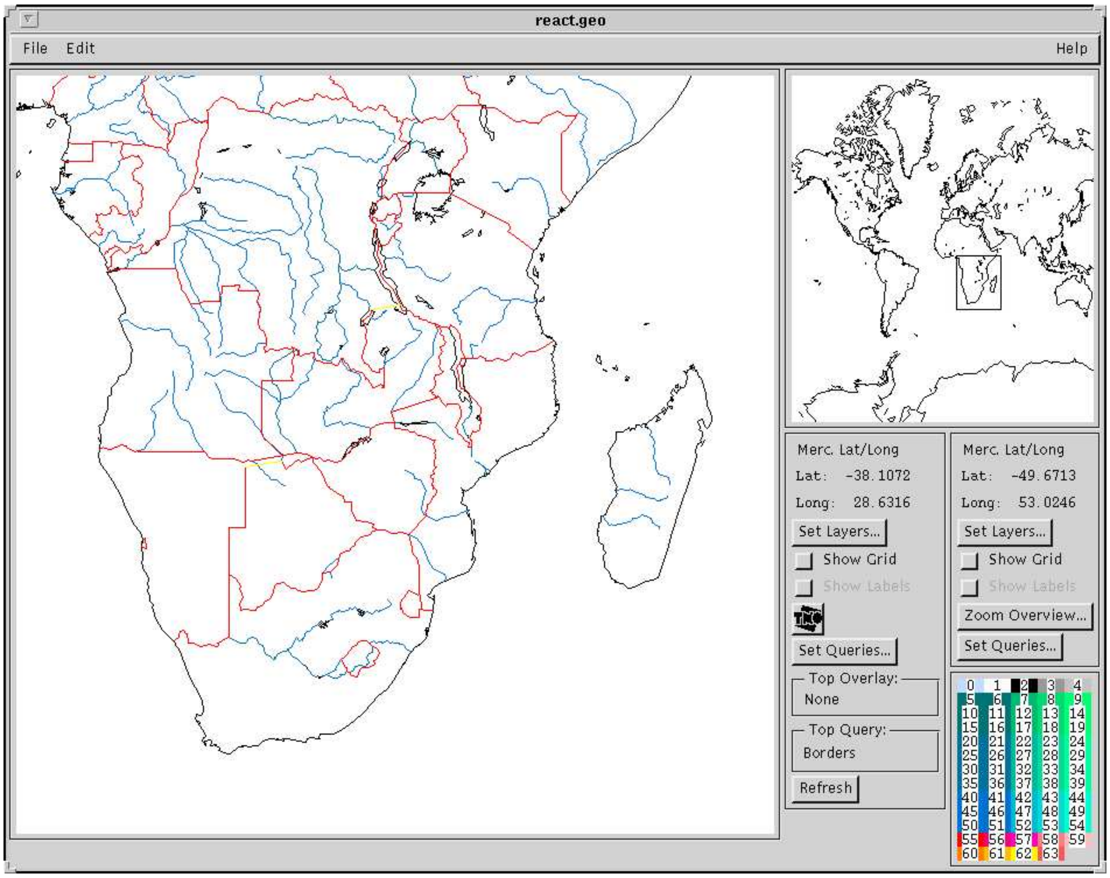

Figure 9: Using the Reactive-tree and BLG-tree (WDB II)

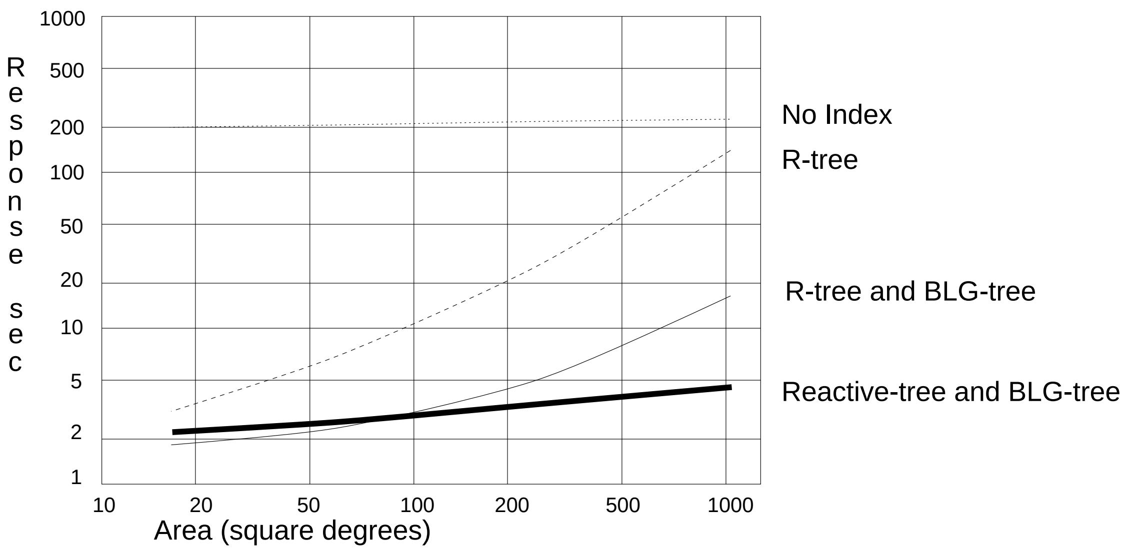

Figure 10: The average response time of the index structures in WDB II

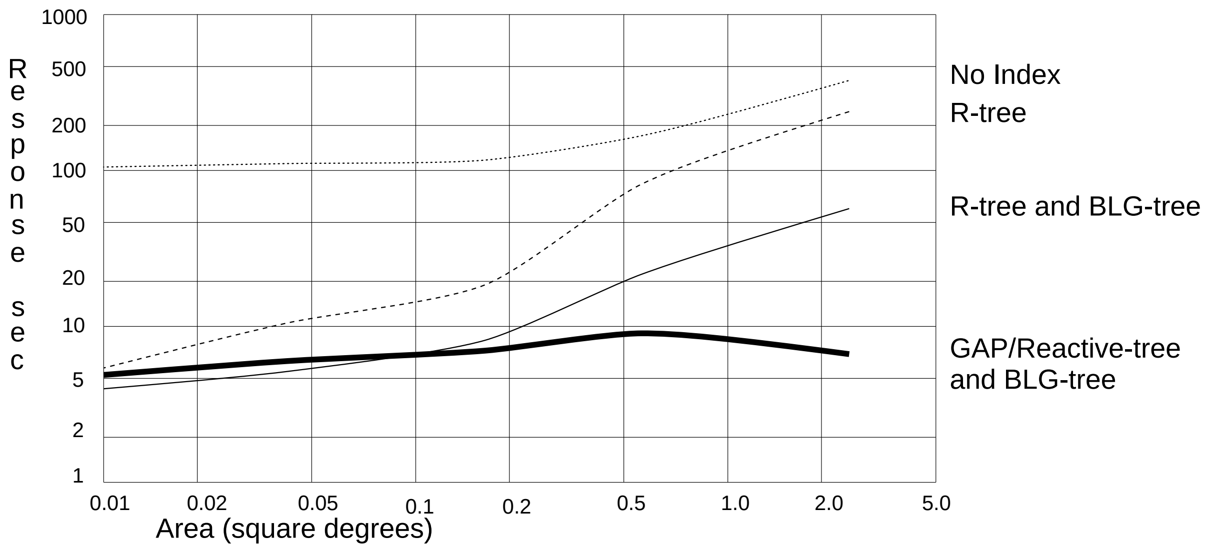

Figure 11: The average response time of the index structures in DLMS DFAD

Table 1: Area sizes and the average number of returned objects in WDB II

Area (in square degrees)	16	64	256	1024
No index	16.2	45.9	244.2	1380.4
R-tree	16.2	45.9	244.2	1380.4
R-tree & BLG-tree	16.2	45.9	244.2	1380.4
Reactive-tree & BLG-tree	10.7	23.8	67.0	257.6

Table 2: Area sizes and the average response time in WDB II

Area (in square degrees)	16	64	256	1024
No index	205.5	211.2	230.8	348.8
R-tree	3.4	7.0	27.0	143.5
R-tree & BLG-tree	1.8	2.2	5.0	15.3
Reactive-tree & BLG-tree	2.3	2.6	3.2	4.9

Table 3: Area sizes and the average number of returned objects in DLMS DFAD

Area (in square degrees)	0.01	0.04	0.16	0.64	2.56
No index	25.9	64.7	198.0	723.6	2262.3
R-tree	25.9	64.7	198.0	723.6	2262.3
R-tree & BLG-tree	25.9	64.7	198.0	723.6	2262.3
GAP/Reactive-tree & BLG-tree	25.9	52.8	43.6	55.8	48.7

Table 4: Area sizes and the average response time in DLMS DFAD

Area (in square degrees)	0.01	0.04	0.16	0.64	2.56
No index	113.8	117.7	127.1	183.4	378.7
R-tree	6.6	10.2	19.0	80.5	285.8
R-tree & BLG-tree	4.3	5.7	8.1	21.2	61.2
GAP/Reactive-tree & BLG-tree	5.4	6.9	7.5	9.7	7.3

List of Figures