

A Family of Storage Methods for Geographic Data

Maurício R. Mediano^{1,2}, Marco A. Casanova², Marcelo Dreux^{1,2}

109

¹Pontifícia Universidade Católica - Rio R. Marquês de São Vicente, 225 22453 Rio de Janeiro RJ - Brasil
²Centro Científico Rio - IBM Brasil Av. Presidente Vargas, 824/844 20071 Rio de Janeiro RJ - Brasil

Abstract

This paper first introduces a new data structure, called *V-trees*, designed to store long sequences of points in 2D space. They operate in much the same way as positional B-Trees do in the context of long fields, but they are better understood as a variant of R-trees. Then, the paper introduces a second data structure, called *VR-trees*, that extends *V-trees* to store sets of sequences of points. Finally, the paper discusses how to use *VR-trees* to store thematic maps, that is, sets of non-overlapping polygons.

The design of these structures was motivated by the problem of storing and retrieving geographic objects that are fairly long, such as river margins or political boundaries, and the fact that geographic queries typically access just fragments of such objects, frequently using a smaller scale.

1 Introduction

Geographic databases typically store objects that fall into two broad classes [9], geographic objects and fields. *Geographic objects* have an identification with real-world elements, such as parcels in a cadastral map and poles in an electrical network. These objects can be associated with one or more geographic representation in a given geo-referenced coordinate space. By contrast, *fields* represent spatially distributed variables taking values from some domain. They may be associated with digital terrain models, thematic maps or reflectance to incident radiation (as in satellite or aerial images).

This paper is primarily concerned with the problem of storing and retrieving the representation of geographic objects that are fairly long, such as river margins or political boundaries. In more general terms, we concentrate on the storage and retrieval of long sequences of points in 2D space, in a given geo-referenced coordinate space. From the onset, it should be stressed that geographic queries [6, 7] typically access just fragments of the representation of a (long) geographic object, not the whole representation, and they may involve changes to smaller scales (i.e, from 1:250,000 to 1:1,000,000), which means that many points of the object representation will map into a single point, as far as the query is concerned.

The solution we propose is based on a new data structure, called *V-trees*, basically designed to store long sequence of points and yet allow efficient access to their fragments. They also greatly optimize access to a sequence of points when the query involves changes to a smaller scale, since they permit to easily generate an approximation of the sequence of points that best suits the scale chosen. We also describe variations of *V-trees* that store sets of sequence of points and thematic maps.

V-trees operate in much the same way as positional B-trees do in the context of long fields [3, 4] and they can be viewed as a variant of R-trees [1, 13, 23]. However, unlike R-trees, the construction of V-trees takes into account just the sequence of points itself and a special node balancing criterion, and it is not concerned with minimizing the superposition of bounding boxes.

We stress that V-trees were designed as a storage method for long sequence of points, and not as a spatial access method. In this last category, in addition to R-trees, we find an enormous variety of data structures. Good surveys of spatial access methods can be found in [8, 12, 14, 16, 19, 22].

This paper is organized as follows. Section 2 contains the basic definitions we need in later sections. Section 3 introduces V-trees and gives some additional motivation. Section 4 sketches some operations over V-trees. Section 5 describes some useful variations of V-trees, while Section 6 discusses how to use V-trees to store sets of sequence of points and thematic maps. Finally, Section 7 contains the conclusions.

2 Vector Objects

In the context of this paper, we assume as given a 2D coordinate system and work with a family of simple 2D geometric figures defined as follows.

A *point* is a pair of real numbers, called the *x*- and *y*-coordinates of the point. A *segment* is a pair of points, called the *start* point and the *end* point of the segment.

A *polyline* is a finite sequence of points; the first and the last points of the sequence are called the *start* and the *end* points of the polyline. A polyline then defines a sequence of segments such that the end point of a segment is the starting point of the next segment. A polyline P is a *fragment* of a polyline Q iff the sequence of points that defines P is a subsequence of contiguous points of the sequence that defines Q . A polyline P is the *reverse* of a polyline Q iff the sequence of points that defines P is the reverse of the sequence of points that defines Q . A *long* polyline is a polyline whose sequence of points is long (according to some criterion that is left unspecified, but which will become intuitively clearer as the discussion proceeds). A polyline is *closed* iff its start and end points coincide; otherwise it is *open*.

We will use without definition the relationships *intersect*, *touch* and *contains*.

The *bounding box* of a polyline is the smallest rectangle such that the sides are parallel to the axis of the coordinate system and the rectangle contains the polyline.

A *polygon* P is a pair whose first element is a finite list N_0, N_1, \dots, N_{n-1} of distinct points, called the *vertices* of P , and whose second element is a finite list L_0, L_1, \dots, L_{n-1} of distinct polylines, called the *edges* of P , such that:

1. N_i is the first point of L_i and the last point of L_{i-1} , for $i \in [0, n - 1]$ (where sum is module n);
2. each polyline does not touch itself and two polylines touch each other only in the situation just described above;

3. each polyline does not intersect itself or another polyline.

A *thematic map* [10] M with *bounding box* B is a set of polygons, called *regions* of M , such that:

1. all regions of M are contained in B ;
2. any two regions of M do not overlap with each other.

Intuitively, a thematic map defines a geographic field such that each region of the map corresponds to a region in 2D space where the field has a uniform value. The region in the 2D space defined by the bounding box and not covered by any polygon is sometimes called the *background* of the thematic map, which also corresponds to a value of the field.

Finally, we observe that the above definition is simplistic in the sense that it captures neither the complete topological information usually associated with thematic maps nor the case where the regions have “holes”. However, it suffices to illustrate some of the problems associated with the storage of thematic maps.

3 V-Trees

3.1 Motivation for V-Trees

V-trees were inspired on the storage structure for long objects developed for the EXODUS system [4], which was in turn based on the ordered relations proposed in Stonebraker et alii [26].

Conceptually, a long object in EXODUS is just a byte string. Physically, a long object is stored on disk as a B⁺-tree, where keys denote positions (in bytes) within the object and leaves store consecutive blocks of bytes of the object (recall that leaves are always half occupied, at least, in B⁺-trees). The internal identifier of the object is a pointer to the root of the tree that stores it.

The root R of the tree contains a list of pairs of the form (c, p) , one for each child F of R , where p points to F and c is the position of the rightmost byte stored in the subtree whose root is F . The position associated with the last child is then the size of the object. An interior node N is similarly defined and corresponds to a substring s of the byte string that represents the object. That is, the positions stored in N represent displacements relative to the beginning of s .

Consider the example shown in Figure 1. The subtree whose root is the left child of the root stores bytes 1 to 421 and the other subtree stores the other bytes. The rightmost leaf stores 173 bytes. Byte 100 within this leaf is byte $192+100=292$ within the substring associated with the right child of the root and byte $421+292=713$ within the object as a whole.

Let us now return to long polylines. They can naturally be stored as long objects using B⁺-trees as above, but this strategy will hardly help processing geographic queries.

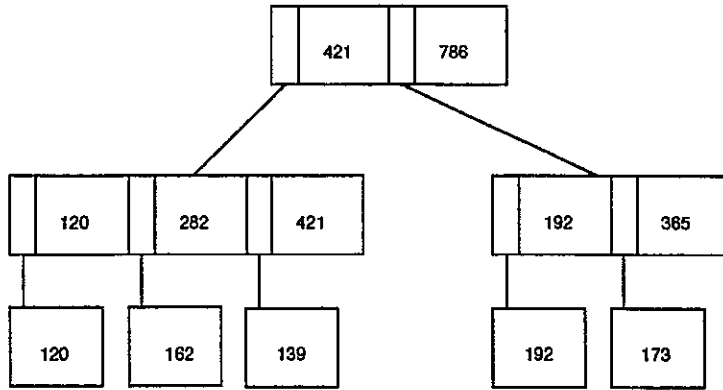


Figure 1: An example of positional B-tree.

Indeed, most of the time, the query processor will be interested in accessing the fragments of a polyline that fall within a given region. Very rarely, if at all, the processor will be interested in accessing points of the polyline according to their relative position within the polyline. In other words, we need a storage structure that will help access fragments of a polyline by their position in 2D space, and not by their relative position.

V-trees correct this deficiency, while retaining the basic idea of the EXODUS storage method. A V-tree is very similar to an R^+ -tree, where leaves store consecutive blocks of points of the polyline in question. An interior node N of the V-tree contains a list of pairs of the form (p, B) , one for each child C of N , where p points to C and B is the bounding box for all fragments stored in the subtree whose root is C .

In a sense, the byte displacements of the B^+ -trees give way to bounding boxes of fragments of the polyline. V-trees therefore facilitate accessing the fragments of the polyline that fall within a given region. Figure 2 shows a V-tree where the leaves V_{11} , V_{12} , V_{13} , V_{14} , V_{21} and V_{22} store the fragments that compose the polyline.

3.2 Definition of V-Trees

Given integers $m > 1$ and $n > 1$, a *V-tree* of order (m, n) is an m -way tree such that:

- All leaves are at the same level.
- Each leaf N contains a sequence of points of size between $n/2$ and n . The bounding box of N is the bounding box of the sequence of points N stores.
- Each interior node has between $m/2$ and m children, except the root, which has between 2 and m children.
- For each child M of an interior node N , N contains an entry consisting of a pointer to M and the bounding box of M . The bounding box of an interior node N is the bounding box covering all the bounding boxes of entries in N .

Given a polyline P , one may then store P in a V-tree V of order (m, n) by breaking P into consecutive fragments of size n and storing each fragment in a leaf of the V-tree.

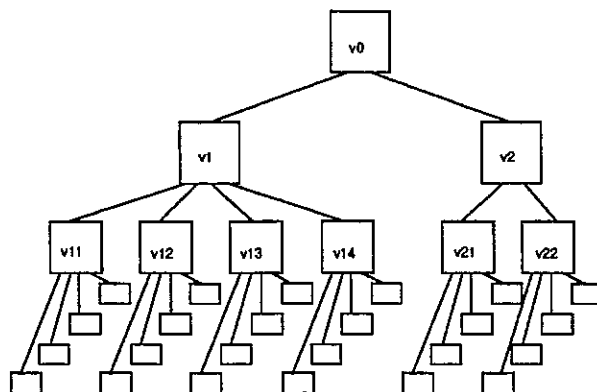
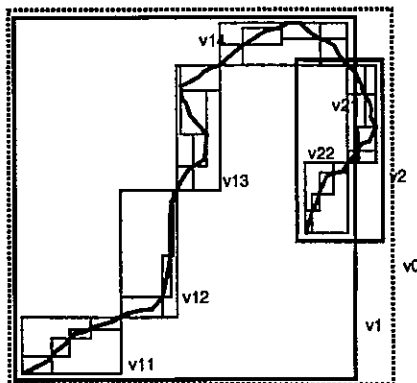


Figure 2: An example of V-tree.

In the process of breaking a polyline into consecutive fragments, the last point of a fragment becomes the first point of the next fragment so that the bounding boxes of the fragments completely cover the polyline. Naturally, all algorithms that reconstruct P from V must be aware that the last point of a fragment is the first one of the next one (except if the fragment is the last one and the polyline is not closed).

At this point, we may digress into some practical considerations. First, the interior nodes and the leaves need not occupy pages of uniform size, since they store entries of different nature. In other words, the parameters m and n are in principle unrelated. Second, one may link the leaves together to facilitate accessing consecutive fragments of a polyline. If the polyline is closed, the leaves may in fact be organized as a circular list.

4 Sample Operations for V-Trees

4.1 Sample Retrieval Operations for V-Trees

In this section, we describe three retrieval operations for V-trees that help assess the usefulness of the structure.

Consider first the clipping operation, that receives as input a polyline P stored in a V-tree V and a bounding box B , defined by a pair of points (the top-left and the bottom-right corners), and returns the fragments of P that fall inside B . The algorithm, shown

```

algorithm clipping
  input: V-tree, search box
  output: reported fragments
begin
  if node is not leaf then
    for each entry do
      if entry.bounding box intersects search box then
        call clipping for entry.sub-tree
      else
        report intersection between fragment stored in the leaf and search box
  end clipping

```

Figure 3: clipping algorithm.

in Figure 3, basically traverses V visiting all subtrees whose bounding boxes intersect B . For simplicity, the algorithm returns the answer as a sequence of polylines. Note how the clipping operation indeed illustrates why V-trees are an interesting storage strategy for long polylines.

We now discuss a variation of the clipping operation that reduces the number of nodes retrieved when the user is operating in a scale and precision which is smaller than the scale used to store polylines (we remind that we consider, for example, a 1:1,000,000 scale to be smaller than a 1:250,000 scale). In this case, it suffices to generate just an approximation of polyline which is good enough for the scale chosen. For example, if the user wants to visualize a polyline in a scale smaller than that used to store the polyline, then many points of the polyline will correspond to a single pixel on the screen.

Suppose that the user is operating with precision p (explicitly stated or induced by the scale chosen). The approximate operation, shown in Figure 4, will then stop descending a subtree T with root N and approximate all points represented by the leaves of T by the centroid of the bounding box B of N , if the diagonal of B is less than or equal to $2 * p$. This condition guarantees that the error, that is, the distance between the centroid and any of the points it approximates is less than or equal to p .

We now turn to the subcase of the contains operation that tests if a point w is in the region defined by a closed polyline P that does not cross itself and which is stored in a V-tree V . The algorithm, shown in Figure 5, is a variant of that discussed in [20]. It basically draws a line L , parallel to the X-axis, starting on w and extending to the right, and counts how many times L crosses P ; if the number is odd, then w is inside P , otherwise w is outside. The algorithm is optimized in the sense that, if L does not cross the bounding box B associated with a subtree T , then it will not traverse T , since L does not definitely cross any of the fragments of the boundary of P stored in T .

```

algorithm approximate
  input: V-tree, precision
  output: reported points
begin
  if node is not leaf then
    for each entry do
      if entry < precision then
        report center of entry.bounding box
      else
        call approximate for entry.sub-tree
    else
      report sequence of points stored in the leaf
end approximate

```

Figure 4: approximate algorithm.

4.2 Inserting and Deleting from V-Trees

Consider first the insert operation that inserts a new point w into a polyline P , stored in a V-tree V . For simplicity, we assume that, in addition to w and V , the operation receives as input a pointer to the exact position within a leaf of V (already in main memory) where w must be inserted.

The algorithm is identical to that of B-trees. The new point w is inserted into the appropriate leaf of V and nodes are recursively split towards the root. We only remark that, if the node is a leaf, then the first point of the sequence stored in the new node must be repeated as the last point of the new sequence stored in the old node. Moreover, the bounding box of each node that was split must be recomputed and the corresponding entry in the parent node must be updated, and so on recursively towards the root.

The delete operation also accepts as input a point w , identified by a pointer to a leaf in main memory, and a polyline P , stored in a V-tree V . The algorithm is also identical to that of B-trees. The point w is deleted from the appropriate leaf of V and nodes are recursively merged towards the root. We again observe that, if two adjacent leaves are merged, then the last point of left leaf must be dropped since it is the first point of the right leaf. The bounding box of merged nodes must also be recomputed and the corresponding entry in the parent node must be updated.

Naturally, we exemplified just the simplest format of the insertion and deletion operations. More complex variations could insert and delete entire fragments of a polyline.

5 Variants of V-Trees

In this section, we introduce two useful variants of V-trees: the static V-trees and the V*-trees.

```

algorithm contains
  input: V-tree, point
  output: flag
begin
  draw line from point to the right
  call contains1
  if counter is odd then
    assign true to flag
  else
    assign false to flag
end contains

algorithm contains1
  input: V-tree, line
  output: counter
begin
  if node is not leaf then
    for each entry do
      if line intersect entry.bounding box then
        call contains1 for entry.sub-tree
    else
      if line crosses fragment stored in the leaf then
        increment counter
end contains1

```

Figure 5: contains algorithm.

Given integers $m > 1$ and $n > 1$, a *static V-tree*, or *SV-tree*, of order (m, n) is an m -way tree such that:

- All leaves are at the same level.
- Each leaf N contains a sequence of points of size n , except possibly the rightmost leaf. The bounding box of N is the bounding box of the sequence of points N stores.
- Each interior node has m children, except possibly the rightmost node in each level of the tree (including the root).
- For each child M of an interior node N , N contains an entry consisting of a pointer to M and the bounding box of M . The bounding box of an interior node N is the bounding box covering all the bounding boxes of entries in N .

An algorithm to construct an SV-tree V of order (m, n) for a polyline P would first sequentially insert all points of P into buckets of n elements, creating the leaves of V . As

for the regular V-trees, the last point in a leaf is the first one of the next leaf. Note that only the last leaf will have less than n entries. Then, the algorithm would recursively create the internal nodes of V up to the root by sequentially inserting the bounding boxes and the references to the newly created nodes into buckets of m elements. Note again that only the last node of a given level will have less than m entries.

This algorithm immediately suggests another variation of static V-trees that avoids having nodes with less than $m/2$ entries (and leaves with less than $n/2$ entries). The algorithm to create trees of this second variation simply distributes, at a given level, the sequence of entries so that the number of entries in any two buckets differ by at most one.

Finally, we may define V^* -trees by analogy with B*-trees: it suffices to split two nodes into three, when necessary, instead of splitting a node into two. Each node will then contain $2/3$ of the maximum allowed number of entries.

6 Handling Sets of Polylines and Thematic Maps

6.1 Storing Sets of Polylines

In previous sections, we discussed how to store a single (long) polyline, be it open or closed. In this section we briefly address how to store a set of (long) polylines.

Let Π be a set of polylines in what follows.

The approach we suggest is to store each polyline P in Π in a separate V-tree, approximate P by its bounding box B_P and then insert B_P into any of the (many) spatial access methods designed to access rectangles.

If the access method chosen is an R-tree, we will call the final structure an *VR-tree*. Figure 6 illustrates a VR-tree. For each VR-tree, there will always be certain level l such that:

- if we drop all nodes below level l , the resulting structure is a regular R-tree, called the *R-tree component* of the VR-tree;
- each subtree rooted at a node of level l is a V-tree, called a *V-tree component* of the VR-tree.

We note that, unlike all other spatial access methods that try to minimize bounding box overlapping, VR-trees store complete fragments of the polylines in the leaves of their V-tree components.

We also note that a VR-tree is not a balanced search tree in the sense that the leaves will not be at the same level, since they actually belong to distinct V-trees which are subtrees of the VR-tree. Therefore, a VR-tree is not a V-tree with the leaves storing points belonging to distinct polylines.

Let Ψ be the VR-tree storing the set of polylines Π .

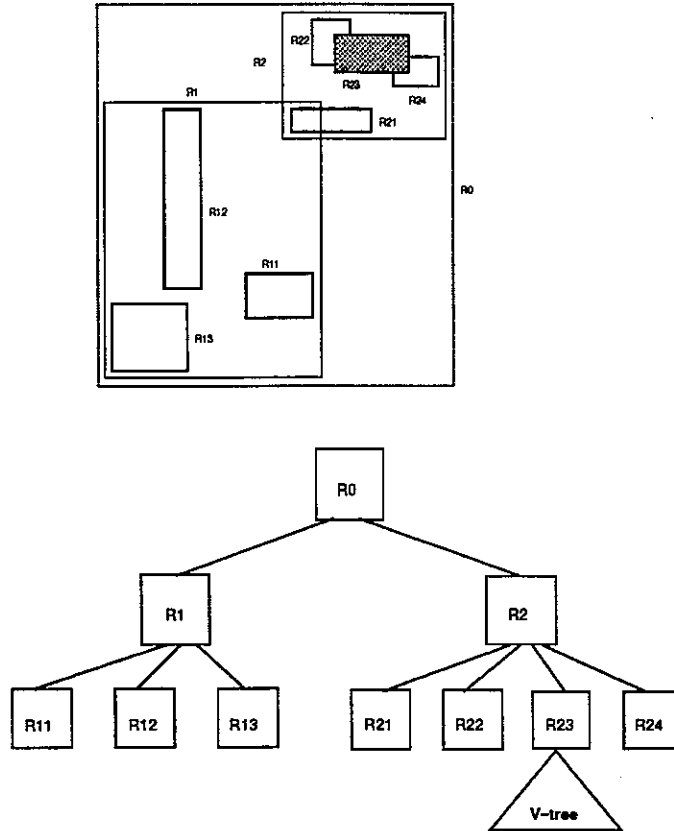


Figure 6: An example of VR-tree.

The approach just outlined allows implementing search operations for Π which are similar to those described in Section 4.1, but which traverse Ψ . However, it still permits separate access to each polyline in Π , since the V-tree storing the polyline is an independent structure.

The insertion of a new polyline P in Π is decomposed into the creation of a V-tree V for P , the insertion of the bounding box B_P of P into the R-tree component of Ψ and the addition of V as a new V-tree component of Ψ . The deletion of a polyline P from Π is likewise decomposed into the deletion of the V-tree V for P and the deletion of the bounding box B_P of P from the R-tree component of Ψ . Again, since polylines are stored in independent structures, direct insertions and deletions of points to/from a polyline remain unaffected.

We may define a variation of VR-trees by using SV-trees (or V*-trees) to store the polylines, thus generating the family of SVR-trees (or V*R-trees). If we allow a polyline to be stored indistinctly either by a V-tree, a SV-tree or a V*-tree, we call the resulting family the *GR-trees*. This last family is more flexible in the sense that it allows storing each polyline in a set using the variation of V-tree that suits it best.

6.2 Storing Thematic Maps

This section briefly addresses how to extend the discussion about VR-trees to cover thematic maps [10].

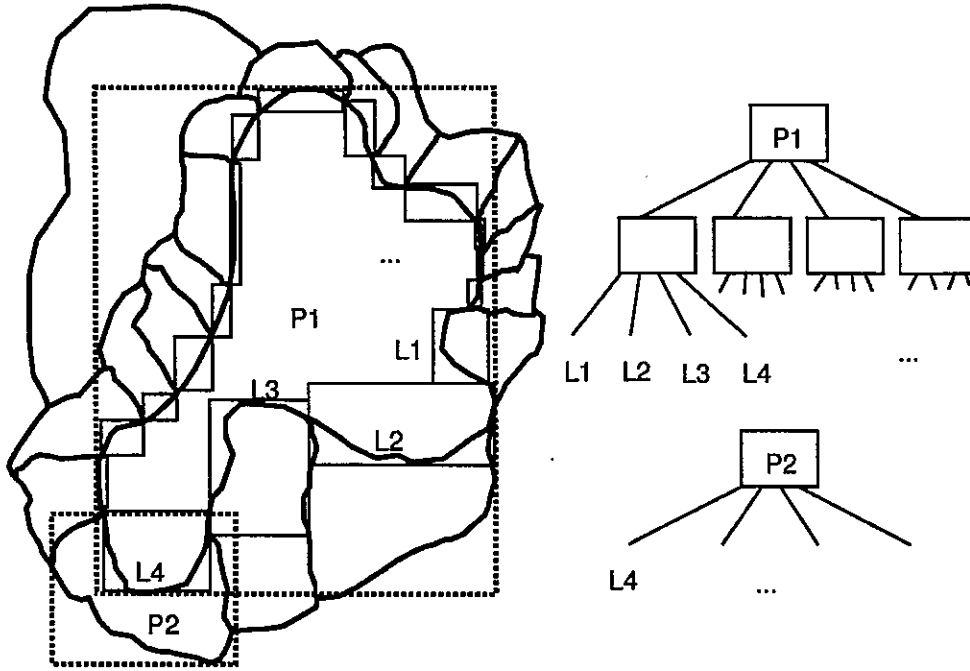


Figure 7: V-trees for polygons

Recall from Section 2 that a thematic map is a set of non-overlapping polygons, called regions, and that the regions considered in this paper do not have “holes”. Therefore, storing a thematic map means storing a set of polygons.

However, observe that, given two polygons P and Q , they may have a common edge, or a polyline L that is an edge of P may be the reverse of a polyline that is an edge of Q . Hence, L may be stored just once and the structures storing P and Q may point to the structure storing L and indicate whether L itself or the reverse of L is an edge of the polygon. This strategy obviously saves storage space, and it also gives some extra flexibility to the query optimizer, which can now access the edges of the polygons directly.

More concretely, let M be a thematic map. One may then use

- a VR-tree T_L to store the set of all edges of all polygons of M (a polyline and its reverse is stored just once). Each edge may have a header pointing back to polygons (note that a polyline may be an edge of at most two polygons);
- a second VR-tree T_P to store the set of polygons of M , where each V-tree of T_P will represent a polygon of M in such a way that the leaves will store the sequence of identifiers and bounding boxes of the polylines, stored in T_L , that are edges of the polygon (and an indication of whether the polyline is reversed or not), exactly as an R-tree representing a polyline stores in its leaves the points that form the polyline. This accommodates polygons with an unbounded number of edges.

Alternatively, one may use a single VR-tree T_M to represent M , that is, a V-tree of T_M will represent either a polyline or a polygon of M , using the strategy just described.

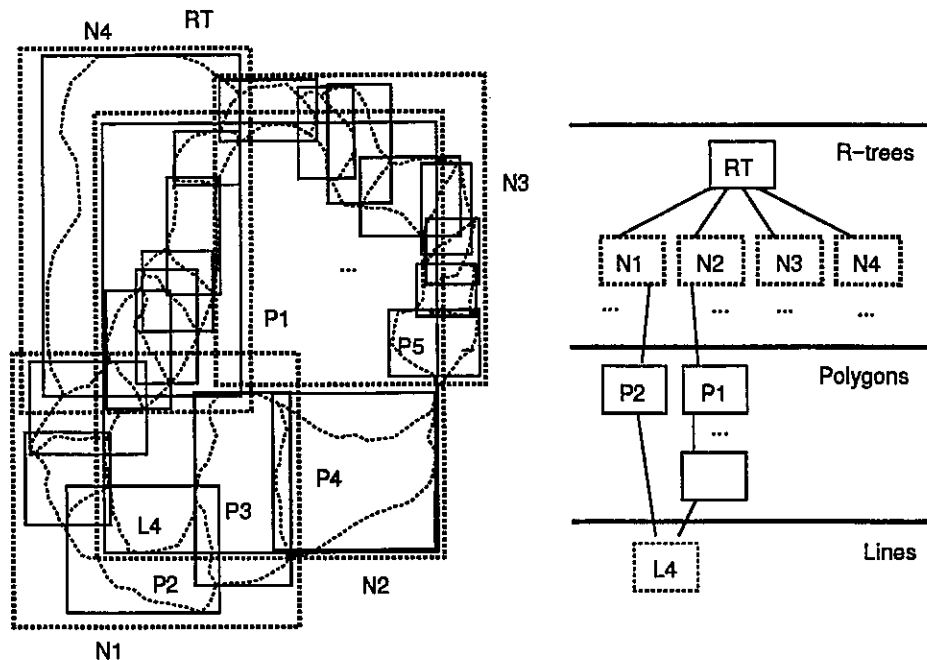


Figure 8: VR-trees for polygons

Figure 7 shows a thematic map that has two polygons P_1 and P_2 with a common edge L_4 , whereas Figure 8 shows the VR-tree T_P that stores the set of polygons of the thematic map. T_P will then have a V-tree T_{P_1} for P_1 and a V-tree T_{P_2} for P_2 , each having an entry with the bounding box and the identifier of L_4 . Hence, it is possible to reach L_4 through the geometries of both P_2 and P_1 walking through their V-trees.

Finally, to add even more flexibility to the query optimizer, one may index the vertices of polygons through a separate structure, which can be an R-tree (storage saving would not be significant in this case). Each vertex may carry additional information about the polygons that share it.

7 Conclusions

We introduced in this paper a storage method, that we called V-trees, for long polylines. We have also discussed how to store sets of long polylines and thematic maps using V-trees in conjunction with R-trees.

The major motivation was the storage and retrieval of representations of long vector objects that are part of a geographic database. We have shown, through sample algorithms, how V-trees facilitate the access to fragments of a polyline and the generation of approximations of polylines in smaller scales. These characteristics facilitate the processing of queries over geographic databases.

The reader may find in reference [15] an early evaluation of V-trees and their variants, using test data synthesized from real geographic data.

Acknowledgements

The results reported in this paper are part of a research project carried out in cooperation between the Rio Scientific Center of IBM Brasil and the Brazilian National Institute for Space Research (INPE).

The first author also wishes to thank the Brazilian National Research Council (CNPq) for partially supporting his research through a scholarship grant.

References

- [1] N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (May 1990), 322-332.
- [2] B. Becker, H.W. Six and P. Widmayer. Spatial Priority Search: An Access Technique for Scaleless Maps. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (May 1991), 128-138.
- [3] H-T.Chou, D.J. DeWitt, R. Katz and A.C, Klug. Design and Implementation of the Winsconsin Storage System. *Software Practice and Experience* 15(10):843-962, Oct. 1985.
- [4] M.J. Carey, D.J. DeWitt, J.E. Richardson and E.J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proc. 12th Int'l. Conf. on Very Large Data Bases* (Aug. 1986), 91-100.
- [5] D. Comer. The Ubiquitous B-tree. In *ACM Computing Surveys*, 11(2):121-131.
- [6] M.J. Egenhofer and A. Frank. Towards a Spatial Query Language: User Interface Considerations. In *Proc. 14th Int'l. Conf. on Very Large Data Bases* (Aug. 1988), 124-133.
- [7] M.J. Egenhofer. *Spatial Query Languages*. PhD thesis, University of Maine, Orono, ME, May 1989.
- [8] C. Faloutsos, T. Sellis and N. Roussopoulos. Analysis of Object Oriented Spatial Access Methods. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, (May 1987), 260-269.
- [9] M. Goodchild. Geographical Information Science. *Int'l. Journal of Geographic Information Systems*, 6(2), 1992.
- [10] M. Goodchild and K.K. Kemp, *Introduction to GIS*. National Center for Geographic Information and Analysis, University of Santa Barbara, Santa Barbara, CA, 1990.
- [11] O. Gunther. *Efficient Structures for Geometric Data Management*, volume 337 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1988.
- [12] O. Gunther and J. Bilnes. Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):342-356, 1991.

- [13] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD Conf. on Management of Data* (June 1984), 599-609.
- [14] E. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Line Segment Databases. In *Proc. ACM SIGMOD Conf. on Management of Data*, (May 1992), 205-214.
- [15] M.R. Mediano, M.A. Casanova and M. Dreux, V-Trees - A Storage Method for Long Vector Data. Technical Report CCR155, Rio Scientific Center, IBM Brasil (Feb. 1994).
- [16] J. Nievergelt. 7+2 Criteria for Assessing and Comparing Spatial Data Structures. In *Proc. of SSD '89*, volume 409 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1990.
- [17] B.C. Ooi. *Efficient Query Processing*. In A. Buchmann, O. Gunther, T.R. Smith, and Y.-F. Wang, editors, *Design and Implementation of Large Spatial Databases*, volume 471 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1989.
- [18] B.C. Ooi, K.J. McDonell and R. Sacks-Davis. Spatial kd-tree: An Indexing Mechanism for Spatial Databases. In *Proc. of IEEE Comp. Software Applications Conf.* (1987), 433-438.
- [19] J. Orenstein. A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (May 1990), 343-352.
- [20] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction* Springer-Verlag, Berlin (1985).
- [21] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases using Packed R-trees. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (May 1985), 17-31.
- [22] H. Samet. *The design and analysis of spatial data structures* Addison-Wesley (1990).
- [23] T. Sellis, N. Roussopoulos and C. Faloutsos. The R^+ -Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. 13th Int'l. Conf. on Very Large Data Bases* (Sept. 1987), 507-518.
- [24] M. Scholl and A. Voisard. *Thematic map modeling*. In A. Buchmann, O. Gunther, T.R. Smith, and Y.-F. Wang, editors, *Design and Implementation of Large Spatial Databases*, volume 471 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1989.
- [25] M. Scholl and A. Voisard. *Object-oriented database systems for geographic applications: an experiment with O₂*. In F. Bancilhon, C. Delobel, and Paris Kanellakis Editors, editors, *Building an Object-Oriented Database System: the Story of O₂*. Morgan and Kaufmann Pub., 1991.
- [26] M. Stonebraker et alii. Document Processing in a Relational Database System. *ACM Transactions on Information Systems*, 1(2), April 1983.