A Qualitative Comparison Study of Data Structures for Large Line Segment Databases*

Erik G. Hoel†
Statistical Research Division
Bureau of the Census
Washington, D.C. 20233

Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Sciences
University of Maryland
College Park, Maryland 20742

Abstract

A qualitative comparative study is performed of the performance of three popular spatial indexing methods – the R*-tree, R+-tree, and the PMR quadtree – in the context of processing spatial queries in large line segment databases. The data is drawn from the TIGER/Line files used by the Bureau of the Census to deal with the road networks in the US. The goal is not to find the best data structure as this is not generally possible. Instead, their comparability is demonstrated and an indication is given as to when and why their performance differs. Tests are conducted with a number of large datasets and performance is tabulated in terms of the complexity of the disk activity in building them, their storage requirements, and the complexity of the disk activity for a number of tasks that include point and window queries, as well as finding the nearest line segment to a given point and an enclosing polygon.

1 Introduction

Spatial data consists of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension which includes time (e.g., [19,20]). The conventional approach to dealing with spatial data is to store it explicitly by parametrizing it and thereby obtaining a reduction to a point in a possibly higher dimensional space. This is usually quite easy to do in a conventional database management system since it is just a collection of records, where each record has many fields. In particular, we simply add a field (or several fields) to the record that deals with the desired item of spatial information. This approach is fine if we just want to perform a simple retrieval of the data.

However, if our query involves the space occupied by the data (and hence other records by virtue of their proximity), then the situation is not so straightforward. In such a case we need to be able to retrieve records based on some spatial properties which are not stored explicitly in the database. For example, in a road database, we may not wish to specify which roads intersect which other roads or regions. The problem is that the potential volume of such information may be very large and the cost of preprocessing it high, while the cost of computing it on the fly may be quite reasonable, especially if the spatial data is stored in an appropriate manner. Thus we prefer to store the data implicitly so that a wide class of spatial queries can be handled (e.g., see the discussion in [11]). In particular, we need not know the types of queries a priori.

Being able to respond to spatial queries in a flexible manner places a premium on the appropriate representation of the spatial data. In order to be able to deal with proximity queries we must effectively sort the data. Of course, all database management systems sort the data. The issue is which keys do they sort on. In the case of spatial data, we feel that the sort should be based on all of the spatial keys. In particular, unlike conventional database management systems, our sorts are based on the space occupied by the data. Such techniques are known as *spatial indexing* methods.

In this paper we study the performance of three popular spatial indexing techniques – the R*-tree [2] (a variant of the R-tree [9]), R+-tree [4,23], and the PMR quadtree [13,14]. Our goal is not to find the best data structure as we don't believe that this can be done. Instead, we demonstrate that they are comparable, while also indicating when and why their performance differs. We focus on a database consisting of a large collection of line segments such as that used in the Bureau of the Census TIGER/Line file [3] for representing the roads and other geographic features in the US. Our study uses queries that would be typically posed to such a database. Thus we see that our study is different from much of the previous work in the database literature (e.g., [9]), which has concentrated on the representation of rectangles, as is common in VLSI applications.

This paper is organized as follows. We first review a number of different methods of indexing spatial data with an emphasis on collections of line segments. Next, we describe the three representations that we study – i.e., the R*-tree, R+-tree, and the PMR quadtree. We also discuss the implementation issues that arise in making a qualitative comparison. This is followed by a description of the queries that we are using and an outline of the algorithms to implement them. The rest of the paper contains a discussion of the results of our experi-

*This work was supported in part by the National Sciencce Foundation under Grant IRI-90-17393.

†Also with the Center for Automation Research at the University of Maryland.

SIGMOD '92, *San Diego, CA, June* 1992, pp. 205-214

ments with the three representations using TIGER/Line files. We conclude with a brief discussion of our findings and a critique of our experiments, as well as suggestions for future work.

2 Spatial Indexing

Each record in a database management system can be conceptualized as a point in a multidimensional space. This analogy is used by many researchers to deal with spatial data as well by use of suitable transformations that map the spatial object into a point (termed a *representative point*) in either the same (e.g., [12]), lower (e.g., [17]), or higher (e.g., [10]) dimensional spaces. This analogy is not always appropriate for spatial data. One problem is that the dimensionality of the representative point may be too high [16]. One solution is to approximate the spatial object by reducing the dimensionality of the representative point. Another more serious problem is that use of these transformations does not preserve proximity.

To see the drawback of just mapping spatial data into points in another space, consider the representation of a database of line segments (e.g., [12]). We use the term *polygonal map* to refer to such a line segment database, consisting of vertices and edges, regardless of whether or not the line segments are connected to each other. Such a database can arise in a network of roads, power lines, rail lines, etc. Using a representative point, each line segment can be represented by its endpoints[1]. This means that each line segment is represented by a tuple of four items (i.e., a pair of x coordinate values and a pair of y coordinate values). Thus, in effect, we have constructed a mapping from a two-dimensional space (i.e., the space from which the lines are drawn) to a four-dimensional space (i.e., the space containing the representative point corresponding to the line).

This mapping is fine for storage purposes. However, it is not ideal for spatial operations involving search. For example, suppose that we want to detect if two lines are near each other, or, alternatively, to find the nearest line to a given point or line. This is difficult to do in the four-dimensional space since proximity in the two-dimensional space from which the lines are drawn is not necessarily preserved in the four-dimensional space into which the lines are mapped. In other words, although the two lines may be very close to each other, the Euclidean distance between their representative points may be quite large.

Thus we need different representations for spatial data. We believe that using data structures that are based on spatial occupancy is the best way to overcome these problems. Spatial occupancy methods decompose the space from which the data is drawn (e.g., the two-dimensional space containing the lines) into regions called *buckets*. They are also commonly known as *bucketing methods*. Traditionally, bucketing methods such as the grid file [15] have always been applied to the transformed data (i.e., the representative points). In contrast, we are interested in bucketing methods that are applied to the space from which the data is drawn (i.e., two-dimensions for line segments).

[1] Of course, there are other mappings but they have similar drawbacks. We shall use this example in the rest of our discussion.

There are four principal approaches to decomposing the space from which the data is drawn. None of them are perfect in the sense that they all have some drawbacks. These drawbacks are mentioned with their descriptions and are discussed in greater detail once the empirical results have been presented.

One approach buckets the data based on the concept of a minimum bounding (or enclosing) rectangle. In this case, objects are grouped (hopefully by proximity) into hierarchies, and then stored in another structure such as a B-tree. The R-tree [9] and the R*-tree [2] are examples of this approach. The drawback of these methods is that they do not result in a disjoint decomposition of space. The problem is that an object is only associated with one bounding rectangle, yet the area that it spans may be included in several bounding rectangles. This means that when we wish to determine which object is associated with a particular point (e.g., the containing rectangle in a rectangle database, or an intersecting line in a line segment database) in the two-dimensional space from which the objects are drawn, we may have to search the entire database.

The other approaches are based on a decomposition of space into disjoint cells, which are mapped into buckets. Their common property is that the objects are decomposed into disjoint subobjects such that each of the subobjects is associated with a different cell. They differ in the degree of regularity imposed by their underlying decomposition rules and by the way in which the cells are aggregated. The price paid for the disjointness is that in order to determine the area covered by a particular object, we have to retrieve all the cells that it occupies. This price is also paid when we want to delete an object. Fortunately, deletion is not so common.

The first method based on disjointness partitions the objects into arbitrary disjoint subobjects and then groups the subobjects in another structure such as a B-tree. The partition and the subsequent groupings are such that the bounding rectangles are disjoint at each level of the structure. The R+-tree [4,23] and the cell tree [8] are examples of this approach. They differ in the data with which they deal. The R+-tree deals with collections of rectangles, while the cell tree deals with convex polyhedra.

Methods such as the R+-tree and the cell tree (as well as the R*-tree) have the drawback that the decomposition is data-dependent. This means that it is difficult to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations such as overlay). In contrast, the remaining two methods, while also yielding a disjoint decomposition, have a greater degree of data-independence. They are based on a regular decomposition. We can either decompose the space into blocks of uniform size (e.g., the uniform grid [6]) or adapt the decomposition to the distribution of the data (e.g., a quadtree-based approach [13,21]). In the former case, all the blocks are of the same size. In the latter case, the widths of the blocks are restricted to be powers of two, and their positions are also restricted.

The uniform grid is ideal for uniformly distributed data, while quadtree-based approaches are suited for arbitrarily distributed data. In the case of uniformly distributed data, quadtree-based approaches degenerate to

a uniform grid, albeit they have a higher overhead. Both the uniform grid and the quadtree-based approaches lend themselves to set-theoretic operations. Thus they are ideal for tasks which require the composition of different operations and data sets. In general, since spatial data is not usually uniformly distributed, the quadtree-based regular decomposition approach seems to be the more flexible and therefore is the one we use in our comparisons. The drawback of quadtree-like methods is their sensitivity to positioning in the sense that the placement of the objects relative to the decomposition lines of the space in which they are embedded effects their storage costs and the amount of decomposition that takes place. This is overcome by using a bucketing adaptation that decomposes a block only if it contains more than n objects.

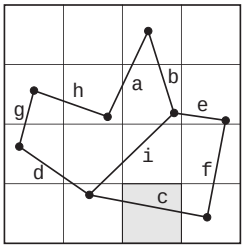

Figure 1: Uniform grid for a collection of line segments.

All of the spatial occupancy methods that we discussed are characterized as employing spatial indexing because with each block we only store information with respect to whether or not it is occupied by the object or part of the object. This information is usually in the form of a pointer to a descriptor of the object. For example, in the case of a collection of line segments in the uniform grid of Figure 1, the shaded block only records the fact that a line segment crosses it or passes through it. The part of the line segment that passes through the block (or terminates within it) is termed a *q-edge*. Each q-edge in the block is represented by a pointer to a record containing the endpoints of the line segment of which the q-edge is a part [13]. This pointer is really nothing more than a spatial index and hence the use of this term to characterize this approach. Thus no information is associated with the shaded block as to what part of the line (i.e., q-edge) crosses it. This information can be obtained by clipping [5] the original line segment to the block. This is important for often we do not have the necessary precision to compute these intersection points anyway.

3 Data Structures

The basic rules for the formation of an R-tree are very similar to those for a B-tree. All leaf nodes appear at the same level. Each entry in a leaf node is a 2-tuple of the form (R,O) such that R is the smallest rectangle that spatially contains line segment O. O points to the actual line segment. Each entry in a non-leaf node is a 2-tuple of the form (R,P) such that R is the smallest rectangle that spatially contains the rectangles in the child node pointed at by P. An R-tree of order (m,M) means that each node in the tree, with the exception of the root, contains between $m \leq \lceil M/2 \rceil$ and M entries. The root has at least two entries unless it is a leaf node.

For example, consider the collection of line segments given in Figure 1. Let $M = 3$ and $m = 2$. One possible R-tree for this collection is given in Figure 2a. Figure 2b shows the spatial extent of the bounding rectangles of the nodes in Figure 2a, with broken lines denoting the rectangles corresponding to the subtrees rooted at the non-leaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual line segments were inserted into (and possibly deleted from) the tree.

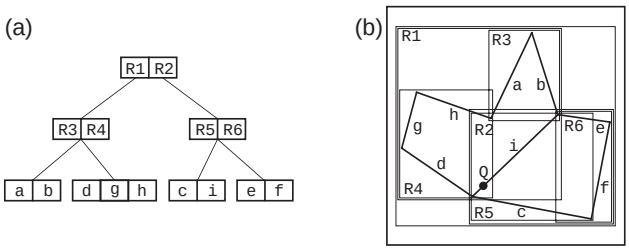

Figure 2: (a) R-tree for the collection of line segments in Figure 1 and (b) the spatial extents of the bounding rectangles.

The algorithm for inserting a line segment (i.e., a record corresponding to its enclosing rectangle) in an R-tree is analogous to that used for B-trees. New line segments are added to leaf nodes. The appropriate leaf node is determined by traversing the R-tree starting at its root and at each step choosing the subtree whose corresponding bounding rectangle would have to be enlarged the least. Once the leaf node has been determined, check to see if insertion of the line segment will cause the node to overflow. If yes, then split the node and redistribute the $M + 1$ records in the two nodes. Splits are propagated up the tree.

There are many possible ways to split a node. One possible goal is to distribute the records among the nodes so that the likelihood that the nodes will be visited in subsequent searches will be reduced. This is accomplished by minimizing the total areas of the covering rectangles for the nodes (i.e., coverage). An alternative goal is to reduce the likelihood that both nodes are examined in subsequent searches. This is accomplished by minimizing the area common to both nodes (i.e., overlap). At times these goals may be contradictory.

The R*-tree is a variant of the R-tree that uses more sophisticated node insertion and splitting algorithms. When deciding which node, say R, is to contain the new line segment, say N, we choose the one for whom the resulting minimum bounding rectangle has the minimum increase of amount of overlap with its brothers (i.e., the other nodes pointed at by its father). Note that this is superior to choosing the node whose bounding rectangle would have to be enlarged the least since such a choice does not reduce the likelihood that the remaining nodes are examined in subsequent searches.

Once the node to be split has been chosen, we must determine the axis (i.e., x or y) it is to be split upon, and the position of the split. The axis is determined by examining all of the possible vertical and horizontal splits (i.e., so each resulting node has at least m and

at most $M + 1 - m$ bounding rectangles), and choosing the split for which the sum of the perimeters of the two constituent nodes is minimized. Now that the axis has been chosen, say the x-axis then we choose the split among the $M - 2 \cdot m + 2$ possibilities that results in a minimal amount of overlap between the two new constituent nodes.

Searching for points or line segments in an R-tree is straightforward. The only problem is that a large number of nodes may have to be examined since a line segment may be contained in the bounding rectangles of many nodes while its corresponding record is only contained in one of the leaf nodes (e.g, in Figure 2, `i` is contained in its entirety in `R1`, `R2`, `R4`, and `R5`). For example, suppose we wish to determine the identity of the line segment in Figure 2 that passes through point `Q`. Since `Q` can be in either of `R1` or `R2`, we must search both of their subtrees. Searching `R1` first, we find that `Q` could only be contained in `R4`. Searching `R4` does not lead to the line segment that contains `Q` even though `Q` is in a portion of bounding rectangle `R4` that is in `R1`. Thus, we must search `R2` and we find that `Q` can only be contained in `R5`. Searching `R5` results in locating `i`, the desired line segment.

The R$^+$-tree is an extension of the k-d-B-tree [18]. The R$^+$-tree is motivated by a desire to avoid overlap among the bounding rectangles. Each line segment is associated with all the bounding rectangles that it intersects. All bounding rectangles in the tree (with the exception of the bounding rectangles for the line segments at the leaf nodes) are non-overlapping[2]. Thus there may be several paths starting at the root to the same line segment. This may lead to an increase in the height of the tree. However, retrieval time is sped up.

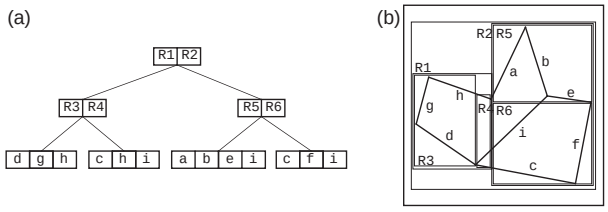

Figure 3: (a) R$^+$-tree for the collection of line segments in Figure 1 and (b) the spatial extents of the bounding rectangles.

Figure 3 is an example of one possible R$^+$-tree for the collection of line segments in Figure 1. This particular tree is of order (2,3) although in general it is not possible to guarantee that all nodes will always have a minimum of 2 entries. In particular, pages are not guaranteed to be m/M full without very complicated record insertion and deletion procedures. Notice that line segments `c` and `h` appear in two different nodes, while line segment `i` appears in three different nodes. Of course, other variants are possible since the R$^+$-tree is not unique.

The difference between the R$^+$-tree and the k-d-B-tree is that once a space has been partitioned, the R$^+$-tree finds the minimum enclosing d-dimensional rectangles for the objects within the d-dimensional rectangles, say S, that result from the split, while the k-d-B-tree leaves rectangles S alone. For example, Figure 4 is the k-d-B-tree for Figure 1. Note the close similarity to Figure 3. This distinction minimizes dead space in the R$^+$-tree.

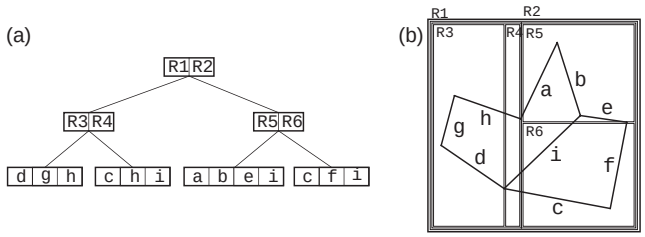

Figure 4: (a) k-d-B-tree for the collection of line segments in Figure 1 and (b) the spatial extents of the bounding rectangles.

Use of the k-d-B-tree leads to faster building times than the R$^+$-tree while the storage costs are the same. Point search queries are slightly faster in the R$^+$-tree than in the k-d-B-tree since a search can fail earlier in the former on account of the minimization of the dead space. Range queries and nearest line segment queries will be a bit faster in the R$^+$-tree than in the k-d-B-tree since the bounding rectangles in the nonleaf nodes of the R$^+$-tree can lead to more pruning than in the k-d-B-tree.

In our experiments (as in [7]) we use a hybrid that lies somewhere between the k-d-B-tree and the R$^+$-tree as we use minimum bounding rectangles for the line segments in the leaf nodes while we don't do so in the nonleaf nodes. This leads to a simplified R$^+$-tree construction algorithm while only sacrificing the minimum bounding rectangles in the nonleaf nodes. This sacrifice is not very costly when we realize that the number of nonleaf nodes is much smaller than the number of leaf nodes. Thus all usage of the term R$^+$-tree in the following refers to this hybrid structure.

Whenever a line segment is inserted into an R$^+$-tree, we apply a recursive top-down process that places it in every leaf node that it intersects. Next, we check if the leaf nodes in which the line segment was inserted should be split on account of it being too full. Splits are propagated up the tree. The R$^+$-tree implementations described in the literature do not specify a splitting policy, and it should be clear that there are a number of possible ways to proceed. We take the approach that a node should be split in a way that minimizes the total number of resulting portions of line segments (bounding rectangles when the node is not a leaf node). This requires that we try all possible vertical and horizontal split lines, and for each split we calculate the number of line segments (or bounding rectangles) that are intersected by the split line. We then select the split line with the minimum number of intersections. In case of a tie, we choose the split line that yields the most even distribution of line segments (or bounding rectangles) among the two constituent nodes.

The third data structure that we examine is the PMR quadtree [13,14]. It is a member of a family of data structures that adaptively sort the line segments into

[2] From a theoretical viewpoint, the bounding rectangles for the line segments at the leaf nodes should also be disjoint. However, this may be impossible when many line segments intersect at a point.

buckets of varying size. There is a one-to-one correspondence between buckets and blocks in the two-dimensional space from which the line segments are drawn. There are a number of approaches to this problem [19]. They differ by being either vertex based or edge based. Their implementations make use of the same basic data structure. All are built by applying the same principle of repeatedly breaking up the collection of vertices and edges (making up the polygonal map) into groups of four blocks of equal size (termed *brothers*) until obtaining a subset that is sufficiently simple so that it can be organized by some other data structure.

The PMR quadtree is edge-based and makes use of a probabilistic splitting rule. A block is permitted to contain a variable number of line segments. It is constructed by inserting the line segments one-by-one into an initially empty structure consisting of one block. Each line segment is inserted into all of the blocks that it intersects or occupies in its entirety. During this process, the occupancy of each affected block is checked to see if the insertion causes it to exceed a predetermined *splitting threshold*. If the splitting threshold is exceeded, then the block is split *once*, and only once, into four blocks of equal size. The rationale is to avoid splitting a node many times when there are a few very close lines in a block. this avoids pathologically bad cases.

A line segment is deleted from a PMR quadtree by removing it from all the blocks that it intersects or occupies in its entirety. During this process, the occupancy of the block and its siblings (the ones that were created when its predecessor was split) is checked to see if the deletion causes the total number of line segments in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the block and its siblings, then they are merged and the merging process is recursively reapplied to the resulting block and its siblings.

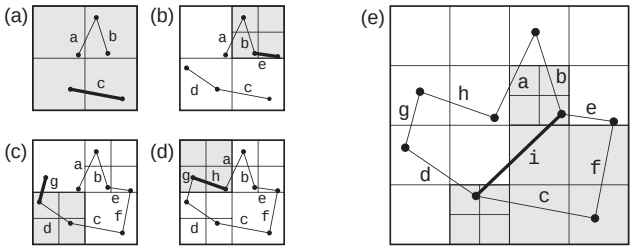

Figure 5: PMR quadtree for the collection of line segments of Figure 1. (a) - (e) illustrate snapshots of the construction process with the final PMR quadtree given in (e).

Figure 5(e) is an example of a PMR quadtree corresponding to a set of 9 edges labeled `a` through `i` inserted in increasing order. Observe that the shape of the PMR quadtree depends on the order in which the lines are inserted into it. Figure 5(a)–(e) shows some of the steps in the process of building the PMR quadtree of Figure 5(e). This structure assumes that the splitting threshold value is two. In each part of Figure 5(a)–(e), the line segment that caused the subdivision is denoted by a thick line, while the gray regions indicate the blocks where a subdivision has taken place. The insertion of line segments `c`, `e`, `g`, `h`, and `i` causes the subdivisions in parts a, b, c, d, and e, respectively, of Figure 5. The insertion of line segment `i` causes three blocks to be subdivided (i.e., the SE block in the SW quadrant, the SE quadrant, and the SW block in the NE quadrant). The final result is shown in Figure 5(e).

Generally, as the splitting threshold is increased, the storage requirements of the PMR quadtree decrease while the time necessary to perform operations on it will increase. Note that although a bucket can contain more line segments than the splitting threshold, this is not a problem. In fact, it can be shown [19] that the maximum number of line segments in a bucket is bounded by the sum of the splitting threshold and the depth of the block (i.e., the number of times the original space has been decomposed to yield this block).

4 Implementation Issues in the Qualitative Comparisons

Comparing the different data structures is not an easy task. For example, in the case of the R$^+$-tree, the details of how they are built and what are ideal parameters are left open. Another problem is that each data structure can be implemented in a different way, thereby making it easy to bias the comparison in favor of a particular structure. These differing implementations make it very difficult to conduct a meaningful comparison since an observer can conclude that we are in effect comparing implementations rather than data structures. It is hard to formulate a meaningful response to such a criticism.

Our approach is to try to compare the data structures using implementations that are commonly associated with them. When to our knowledge, these implementations are not favorable to the data structure, we modify the implementations so that better performance is achieved. This may lead to a slightly different data structure than has been described in the literature. When this is the case, we carefully explain our motivation and show why it is a reasonable modification.

Our goal is to achieve good execution times for the queries outlined in Section 5. This caused us to look a bit more closely at the definitions of the data structures that we were testing and to see what changes in the implementations would enable them to perform better. When the issue is one of time versus storage, we sometimes opted for the more costly solution from the storage standpoint. Since we are dealing with large databases that are expected to be disk-resident, our main statistics are in terms of operations that are expected to cause reading a page of data that is not currently resident in main memory (termed *disk accesses*). This is not difficult to detect as our data structures are organized in terms of pages. Thus we can easily distinguish between operations that access the same page and those that need to access another page. Of course, our disk access statistics only mean that there was a potential for a disk access since depending on the paging policy, the page could be memory-resident. This is often the case. We don't always give raw execution times as they are very difficult to obtain in an accurate manner.

It is difficult to choose parameters that make the testing environments similar. One approach is to stipulate

that the data structures use the same amount of storage. This is not always possible. For example, the R$^+$-tree requires much more space than the R*-tree since the R$^+$-tree stores a line segment with more than one block to insure that the bounding blocks are disjoint. However, the page (i.e., node) sizes can be the same.

The most common implementation of the PMR quadtree is as a linear quadtree which means that only the leaf blocks are retained. Recall that each leaf block can contain several line segments. Each of these line segments is represented by a 2-tuple (L, O) where L is the locational code[3] of the block (a 2-tuple containing the depth of the corresponding node in the quadtree and an integer corresponding to the bit interleaved value of the x and y coordinate values of its lower left corner) and O is a pointer to a segment table that contains the endpoints of the line segment. The segment table is assumed to be on disk where the endpoints of the line segments are stored with whatever precision is deemed necessary. Each 2-tuple (L, O) is stored in a B-tree indexed on the basis of the value of L. With such an implementation, and using 4 bytes per entry, each 2-tuple requires 8 bytes of storage. Using 1K byte pages for the B-tree nodes, we can store 120 line segments on each page. As we see, the notion of a bucket (i.e., a node in the PMR quadtree) is really just a conceptual tool for grouping similar line segments. In particular, the line segments associated with a particular PMR quadtree node should be stored on the same page.

Another issue is the format of a node in the R-tree variants. The most efficient, storage-wise, is to represent each node as a record containing the size of the space spanned by the node (i.e., the minimum bounding rectangle) and pointers to the son nodes. However, this is quite inefficient from an execution-time point of view since in order to determine the bounding rectangle of a son node, we must access the son node which causes a disk access. An alternative, and what we do, is to represent each node as a set of 2-tuples (R, O) where R is the smallest rectangle that contains the data stored in son O. For line segments, this means that each 2-tuple requires 5 entries – 4 for the x and y coordinate values of the bounding rectangle and one entry for the pointer to the son node. In the case of a leaf node, O is really a pointer to a segment table that contains the endpoints of the line segment just like in the PMR quadtree. With such an implementation, and using 4 bytes per entry, each 2-tuple requires 20 bytes of storage and thus each 1K byte page contains a maximum of 50 line segments.

For our experiments, we used our own implementations of the data structures. The PMR quadtree was tested using the QUILT geographic information system [22] where it is implemented using a linear quadtree as described above. It is disk resident and makes use of a buffer pool of 16 pages, each of which contains 1K bytes. Pages are replaced using a least-recently-used policy. The splitting threshold value 4 was chosen on the basis of the observation that we are dealing with collections of line segments that correspond to road networks and that it is rare for more than 4 roads (i.e., line segments) to intersect.

For the R-tree variants, given a page size S and k bytes per 2-tuple, we find that M is approximately S/k. We let m be 40% of M in accordance with the values reported to be best by the originators of the R*-tree [2]. Recall from Section 3 that our implementation of the R$^+$-tree is a hybrid (more like a k-d-B-tree) in that our splitting rule for the insertion does not yield a minimum bounding rectangle. Both R-tree variants were also implemented using a buffer pool of 16 pages and a least recently used page replacement policy.

[3] Also known as a Morton code etc. (see [20] for more details).

5 Queries

We focus on five queries. The queries are specified in a way that is independent of the implementation of the underlying data structure. Each query is a bit more complex than the previous one.

1. Given an endpoint of a line segment, find all the line segments that are incident at it (this is a variant of a *point query*).
2. Given an endpoint of a line segment, find all the line segments that are incident at the other endpoint of the line segment.
3. Given a point in the two-dimensional space containing the line segments, find the nearest line segment using a Euclidean distance metric.
4. Given a point in the two-dimensional space containing the line segments, find the minimal enclosing polygon by outputting its constituent line segments.
5. Given a rectangular window, find all line segments in the window (this is also known as a *range query* or a *window query*).

Queries 1 and 2 are simple search queries that don't require that the space occupied by the line segments be sorted. They only involve the endpoints of the line segments. They are more realistic than a point query which would just return the block containing the point.

Queries 3 and 4 benefit from sorting the space occupied by the line segments. The execution of query 4 requires that we find a line segment that is near the query point and then traverse the boundary of the polygon that surrounds it. The traversal is performed by repeatedly executing query 2 and determining the right line segment from the ones that are returned. This could be facilitated if the line segments that are stored within a bucket in a PMR quadtree, or a page in an R-tree, were sorted. In general, sorting these line segments is not easy as depending on the data structure there are several possible sort techniques – e.g., by orientation around a common point, by x intercept value, by y intercept value, etc.

Query 3 can be used to find the nearest subway line to a particular house, etc. Query 5 can be viewed as a generalization of queries 1 and 3 although its execution is quite different. Query 5 can be used to find all roads that pass through a given region (rectangular here).

The algorithms for executing these queries are straightforward. In the case of the R-tree variants, the algorithms are very similar with the exception that the point query will be faster in the R$^+$-tree than the R*-tree due to the disjointness of the search space. Queries 1 and 2 are point queries and their algorithms are well

known. Queries 3 and 4 require that we find the nearest line segment to a query point. An algorithm for this task for a PMR quadtree is given in [11] and a similar approach is used in the R-tree variants.

Query 5 is executed by a tree traversal that checks for overlap of the nodes with the window rectangle. The R-tree variant algorithms are quite standard [20], while the PMR quadtree uses a new window decomposition algorithm [1].

6 Empirical Results

For each query type and map, 1000 tests were performed using an R*-tree, an R+-tree, and a PMR quadtree. Tests were run on 6 maps of counties in Maryland where each map contained approximately 50,000 line segments. The counties included urban areas (Baltimore), suburban areas (Anne Arundel), and rural areas (Cecil, Charles, Garrett, and Washington).

The PMR quadtrees were built using a maximum depth of 14, thereby enabling an image of size 16K by 16K to be handled. For all the data structures, a minimum bounding square was computed for each map, and all coordinate values were normalized with respect to a 16K by 16K region. This meant that each map was assumed to contain 2^{28} pixels and hence the maximum depth of the PMR quadtree was 14. Of course, the depth of the B-tree implementations of all the trees (including the PMR quadtree) was considerably smaller (i.e., 4).

We also measured the costs of building the structures in terms of storage, disk accesses and cpu time (on an HP 720; 57 MIPS). Table 1 shows the number of bytes in the B-tree, and disk accesses arising in the building process for each map and data structure. The storage requirements of the R+-tree and the PMR quadtree were comparable, with the PMR quadtree using between 13 and 43% more than the R*-tree, and the R+-tree using between 26 and 43% more than the R*-tree. The disk accesses for all three structures were also comparable, with the PMR quadtree requiring the fewest, and the R*-tree the most for all but one of the six data sets. We do not include the segment table size since it is the same for all of the structures. Of course, the number of disk accesses do not tell the whole story of the building process. In particular, there is a considerable amount of other activity going on such as optimization of the splits in the case of the R*-tree which is not at all present in the PMR quadtree and R+-tree.

In terms of execution time, we found that the building times of the R+-tree were the smallest, with the PMR quadtree taking 1.5–1.7 times more, and the R*-tree was more costly by a factor of 7.8–9.1. Building the PMR quadtree was costlier than building the R+-tree because whenever a q-edge is inserted into a PMR quadtree node we must keep the contents of the resulting B-tree nodes sorted which means that data may have to be moved. In contrast, the 2-tuples that comprise the R+-tree nodes need not be sorted. Thus a 2-tuple corrsponding to a q-edge can simply be inserted as the last element in the R+-tree node. Of course, this does affect the execution times of some of the queries. The R*-tree building times suffered from the computationally expensive node overflow technique where 30% of the bounding boxes are reinserted into the structure. If the reinsertion fails to resolve the overflow, the node is then split in a locally desirable manner.

map name	segs	size (Kbytes) R*	R+	PMR	disk accesses R*	R+	PMR	cpu seconds R*	R+	PMR
Anne Arundel	46335	1284	1840	1821	12965	12178	13556	1233	142	237
Baltimore	48068	1345	1907	1925	11901	11007	10640	1310	144	233
Cecil	46900	1325	1722	1541	11034	10644	8829	1137	131	209
Charles	50998	1429	1854	1618	14738	11632	10477	1205	142	225
Garrett	49895	1421	1797	1637	13828	11833	9551	1190	138	211
Washington	49575	1402	1840	1812	12452	11132	10610	1175	150	231

Table 1: Data structure building statistics.

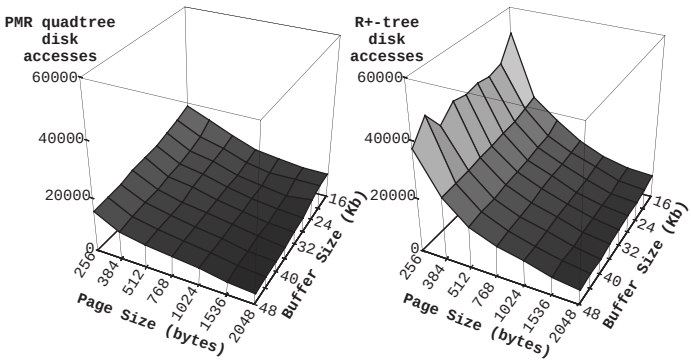

Figure 6: Disk accesses during build by page size and buffer size for the PMR quadtree and the R+-tree.

Figure 6 shows the effect of changing the page size and the size of the buffer pool on the number of disk accesses for the R+-tree and the PMR quadtree. In particular, they decrease as the page sizes and the size of the buffer pool increase. Moreover, for identical page and buffer pool configurations, the number of disk accesses for the PMR quadtree is smaller than for the R+-tree. This is a direct result of the fact that the R+-tree pages contain fewer line segment tuples than the PMR quadtree since the 2-tuples for the former require 20 bytes versus 8 bytes for the latter. In the rest of the experiments we used pages of size 1K and a buffer pool of 16 pages.

Two of the test queries (i.e., finding the nearest line and the surrounding polygon) involve the use of randomly generated query points. We tried two different techniques to generate a query point at random. The first testing technique used a uniform distribution. The problem with such an approach is that many of the query points lie outside the boundaries of the maps of interest, or in large empty areas.

The second testing technique correlates the query points with the data. In particular, regions with high concentrations of line segments are more likely to be queried than sparse regions. Observe that the PMR quadtree provides a good approximation to such a data model as the regions with many line segments are generally small and bunched together while regions with few line segments are generally large. Therefore, we used a two-stage process to generate the query points. We first generated the PMR quadtree block at random using a uniform distribution based on the total number of blocks—not their size. Next, having obtained a random block, we generated a query point at random within the block. In this case, we did draw the coordinate values of the query point from a uniform distribution.

Window queries were attempted using 0.01 percent of the total area represented by the data structure (a similar strategy was used in the original testing of the R*-tree [2]). For example, for a 1K by 1K map, this area is 100 pixels wide (i.e., a 10 by 10 region) while for a 16K by 16K map, this area is 160 by 160 pixels.

One of the problems we encountered was that the line segment distributions for the maps were quite different (recall Table 1). In particular, polygons in urban areas usually consisted of 5-6 line segments corresponding to a city block. On the other hand, in rural areas, the density of line segments is much lower and polygons have much higher line segment counts (e.g., a polygon can be formed by the boundary of a stream and a road where the road and the stream meander in tandem). For example, in our surrounding polygon queries (i.e., query 4), the average polygon size that we encountered was 19 in Baltimore county (an urban and suburban mix) while it was 132 in Charles county (rural). This affects many of the queries. For example, the algorithm for the surrounding polygon query (i.e., query 4) performs one nearest line query (i.e., query 3) and then proceeds to traverse a polygon which is just a repeated application of the query that finds the second endpoint of a line segment given its first endpoint (i.e., query 2). Clearly, its performance differs depending on the average number of line segments in a polygon.

In order to lessen the impact of these differences on the execution times of the queries, for a given map, we normalize the performance of the R-tree data structures (i.e., the R*-tree and the R+-tree) against the performance of the PMR quadtree. For each quantity that was measured using a particular data structure, we tabulate its normalized range. The normalized range highlights the average normalized value for the 6 maps making it easier to see variability. Because of the normalization, the values for the PMR quadtree are always 1.

For each query (7 when the two random point generation methods are used), data structure (3), and map (6), we computed the number of disk accesses and segment accesses, and bounding box computations (in the case of the R*-tree and R+-tree) or bounding bucket computations (in the case of the PMR quadtree). Thus 378 values were computed for the queries. Normalization enables us to reduce it to three figures – one for each of the three quantities measured, when they are comparable. Each figure shows the normalized range (plotted in the vertical direction) as a function of the query type (plotted in the horizontal direction) for each of the different data structures. They are summarized in Figures 7-9. In order to understand the orders of magnitude of the quantities that we are comparing, we tabulate in Table 2 the results of our experiments for Charles county, which were typical of all the maps.

Figure 7 only shows the performance of the R+-tree normalized against the R*-tree. The problem is that although the bounding bucket computations in the PMR quadtree play a similar role to that of the bounding box computations in the R-trees, they usually differ by two orders of magnitude in favor of the PMR quadtree. Hence it was not feasible to plot them using normalized ranges.

A number of conclusions can be drawn from these figures. From Figure 8, the number of disk accesses that we measured showed that the PMR quadtree seemed to have a slight edge over the R-trees. However, the differences were not that great. Moreover, the R+-tree was usually better than the R*-tree because of the disjoint decomposition of space that the R+-tree induces. The differences in the nearest line query is explained below in conjunction with the discussion of the two random point generation methods. An exception was the polygon query (i.e., query 4). At a first glance, this seems contradictory because this query consists of one application of the nearest line query (i.e., query 3) and repeated application of a next point query (i.e., query 2). In particular, on a query by query comparison, the R+-

Charles County				
query	metric	PMR	R+	R*
Point1	disk accesses	1.55	2.07	2.74
	segment comps	3.48	2.43	2.39
	bbox / node comps	1.00	105.02	149.89
Point2	disk accesses	1.72	2.29	2.90
	segment comps	4.43	3.38	3.35
	bbox / node comps	2.00	209.75	299.10
Nearest Line (2-stage)	disk accesses	2.21	2.52	3.35
	segment comps	11.23	27.02	36.16
	bbox / node comps	5.33	248.01	389.05
Nearest Line (1-stage)	disk accesses	7.18	6.75	3.38
	segment comps	22.32	75.08	40.35
	bbox / node comps	8.77	387.86	765.98
Polygon (2-stage)	disk accesses	13.19	18.46	14.07
	segment comps	451.43	388.23	389.85
	bbox / node comps	185.98	16996.69	23730.10
Polygon (1-stage)	disk accesses	12.62	18.67	13.43
	segment comps	368.10	347.95	333.55
	bbox / node comps	152.35	14101.58	20387.28
Range	disk accesses	2.93	3.24	3.50
	segment comps	14.70	8.17	6.88
	bbox / node comps	16.57	149.24	179.76

Table 2: Data for Charles county.

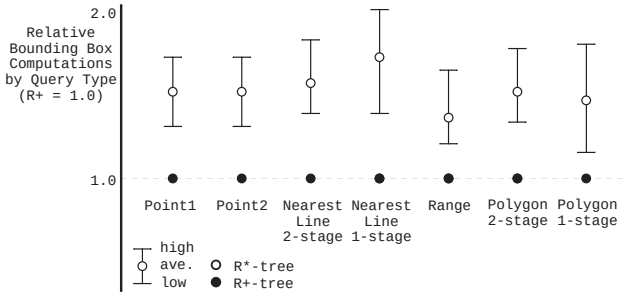

Figure 7: Relative bounding box computations.

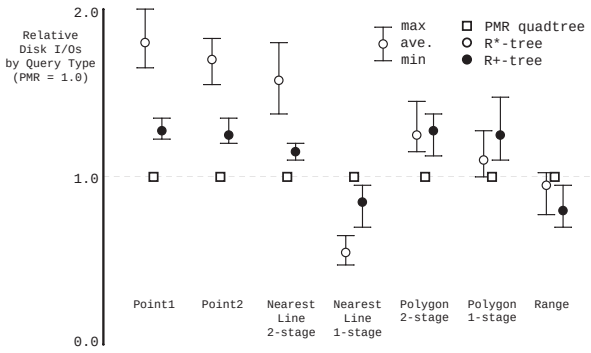

Figure 8: Relative disk accesses.

tree was superior; yet, for the repeated application of the point queries, the R*-tree was slightly better. This is not really surprising when we recall that the R*-tree takes less space and thus the locality of the polygon means that a disk access will be less likely.

Each segment comparison means an access to the segment table which is disk-resident. Of course, since the segments are usually in proximity, they will be stored close to each other and thus although many segments will be involved, there will only be minor differences in disk activity. From Figure 9 we find that they are comparable with the exception of the range and nearest line queries. The advantage of the R-trees for the point queries is relatively small when we consider the order of magnitude of the measured quantities. The performance of the R*-tree and the R+-tree was comparable with the exception of the two-stage nearest line query where the R+-tree seemed to perform decidedly better than the R*-tree. This is most likely due to the disjoint decomposition of space induced by the R+-tree.

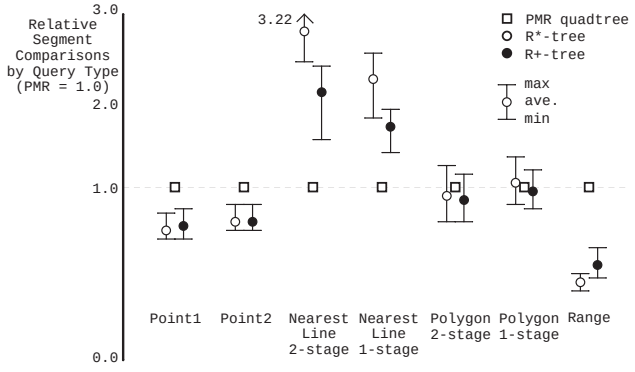

Figure 9: Relative segment comparisons.

The data for the segment comparisons should be examined in conjunction with the data for the bounding box and bounding bucket comparisons. The reason is that unlike the R-trees, no bounding boxes are stored in the PMR quadtree. Thus in the case of the PMR quadtree, the segment comparisons and the bounding bucket comparisons (a very small number) together play the same role as the segment and bounding box comparisons in the R-trees. When we examine the bounding box comparisons (e.g., Table 2 which is typical), we see that this is where much of the computation in both of the R-trees takes place. The advantage of the PMR quadtree over the R-trees in this respect is as high as several orders of magnitude.

It should be clear that the number of segment comparisons in the PMR quadtree can be reduced by modifying the definition of the PMR quadtree so that a minimum bounding rectangle is stored with every line segment or every block. Thus we would have a 3-tuple with a total of 6 entries in contrast to a 2-tuple in the PMR quadtree with just 2 entries. The storage costs would be higher but not by much since the locational code component of the 2-tuple already localizes the bounding rectangle and thus very little additional information needs to be stored (i.e., considerably less than 16 bytes will be required for the bounding rectangle). However, when we examine the relative difference in the absolute number of segment comparisons, we find that it may not be worthwhile to introduce this added complexity.

The results of the two types of random point generation methods (i.e., using one or two stages) did not show a big difference in the surrounding polygon query because its execution time is dominated by the task of finding the successive endpoints of the line segments that comprise the polygon. The number of segment comparisons was comparable for all three structures. For both methods, the number of disk accesses was fewest for the PMR quadtree. Not surprisingly, the R+-tree was better with the two-stage method than the R*-tree, while the results were reversed for the one-stage method.

For the nearest line segment query, the PMR quadtree had fewer disk accesses for the two-stage method than both of the R-trees, while the R*-tree had the most. The situation was reversed for the one-stage method – i.e., the R*-tree had the least, while the PMR quadtree had the most. This was expected as the two-stage method generates more query points where the line segment density is highest and the locality provided by methods based on disjointness (i.e., the R+-tree and the PMR quadtree) result in superior performance. Regardless of the query point generation method, the segment comparisons were fewest for the nearest line segment query when using the PMR quadtree. This is because the PMR quadtree sorts the line segments and is able to prune the search space. The R+-tree performed better than the R*-tree in this case because of the disjointness of the space decomposition induced by it.

7 Concluding Remarks

Although our study has been restricted to collections of line segments, the data structures that we examined are not necessarily restricted to such objects. For example, the R-tree variants have been frequently applied to rectangles. Since they are based on the concept of a bounding box for the spatial data type under consideration, it is easy to extend them to other data types as long as we always use a minimal bounding box. On the other hand, although the PMR quadtree is designed especially for representing collections of line segments, it can also be adapted to represent other data types. Not surprisingly, our studies did not result in claims of overwhelming superiority for any of the data structures. Qualitatively speaking, they are similar. The differences in relative storage costs were as expected.

In terms of choosing a representation for a specific application, the choice can only be made once the repertoire of operations that is to be executed is known. The R+-tree and the PMR quadtree are best when the operations involve search since they result in a disoint decomposition of space. If the results of the operations are to be composed with the results of other operations such as overlay of maps of different types, then the fact that the decomposition induced by the PMR quadtree is oriented so that the decomposition lines are always in the same positions makes it preferable to the R+-tree. The R*-tree is more compact than the R+-tree but its performance is not as good as the R+-tree due to the non-disjointness of the decomposition induced by it.

An interesting observation is that in the case of the PMR quadtree, the splitting threshold plays a role simi-

lar to bucket capacity. One possible way to equalize the comparisons is to use a splitting threshold value that would yield an average bucket (node) occupancy similar to an average page occupancy in the R-trees. This is not difficult to compute. Using our implementations of 1K byte pages, we found that the average number of line segments in an R*-tree page was 36 while it was 32 in an R+-tree tree page. The average number of line segments in a bucket with a splitting threshold value of x is usually $.5x$. This would mean that a PMR quadtree splitting threshold value of approximately 64 may lead to comparable results in the sense that in this case the average page and bucket occupancies will be about the same. Unfortunately, we still need to account for the differing amounts of information stored for each line segment in the structures (i.e., whether or not a bounding box is stored for each line segment in the PMR quadtree). Also, to make the comparisons meaningful, we must address the issue of how the individual line segments stored in a page or bucket are organized (i.e., are they sorted?). Of course, whatever we do we must bear in mind that the real danger of such sophisticated experiments is that the data structures may be crippled by the quest for identical experimental conditions.

References

1. W. G. Aref and H. Samet, A data-driven window decomposition algorithm for efficient spatial query processing, University of Maryland Computer Science TR 2866, College Park, MD, March 1992.
2. N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, The R*-tree : an efficient and robust access method for points and rectangles, *Proc. of the SIGMOD Conf.*, Atlantic City, June 1990, 322–331.
3. Bureau of the Census, TIGER/Line precensus files: 1990 technical documentation, Bureau of the Census, Washington, DC, 1989.
4. C. Faloutsos, T. Sellis, and N. Roussopoulos, Analysis of object oriented spatial access methods, *Proc. of the SIGMOD Conf.*, San Francisco, May 1987, 426–439.
5. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Second Edition, Addison-Wesley, Reading, MA, 1990.
6. W. R. Franklin, Adaptive grids for geometric operations, *Cartographica 21*, 2&3(Summer & Autumn 1984), 160–167.
7. D. Greene, An implementation and performance analysis of spatial data access methods, *Proc. of the Fifth IEEE Intl. Conf. on Data Engineering*, Los Angeles, February 1989, 606–615.
8. O. Günther, Efficient structures for geometric data management, Ph.D. dissertation, UCB/ERL M87/77, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, Berkeley, CA, 1987 (Lecture Notes in Computer Science 337, Springer-Verlag, Berlin, 1988).
9. A. Guttman, R-trees: a dynamic index structure for spatial searching, *Proc. of the SIGMOD Conf.*, Boston, June 1984, 47–57.
10. K. Hinrichs and J. Nievergelt, The grid file: a data structure designed to support proximity queries on spatial objects, *Proc. of the WG'83 (Intl. Workshop on Graphtheoretic Concepts in Computer Science)*, M. Nagl and J. Perl, eds., Trauner Verlag, Linz, Austria, 1983, 100–113.
11. E. G. Hoel and H. Samet, Efficient processing of spatial queries in line segment databases, in *Advances in Spatial Databases – 2nd Symp., SSD'91*, (O. Günther and H. J. Schek, eds.), Lecture Notes in Computer Science 525, Springer-Verlag, Berlin, 1991, 237–256.
12. H. V. Jagadish, On indexing line segments, *Proc. of the Sixteenth Intl. Conf. on Very Large Data Bases*, D. McLeod, R. Sacks-Davis, and H. Schek, eds., Brisbane, Australia, August 1990, 614–625.
13. R. C. Nelson and H. Samet, A consistent hierarchical representation for vector data, *Computer Graphics 20*, 4(August 1986), 197–206 (also *Proc. of the SIGGRAPH'86 Conf.*, Dallas, Aug. 1986).
14. R. C. Nelson and H. Samet, A population analysis for hierarchical data structures, *Proc. of the SIGMOD Conf.*, San Francisco, May 1987, 270–277.
15. J. Nievergelt, H. Hinterberger, and K. C. Sevcik, The grid file: an adaptable, symmetric multikey file structure, *ACM Trans. on Database Systems 9*, 1(March 1984), 38–71.
16. J. A. Orenstein, Redundancy in spatial databases, *Proc. of the SIGMOD Conf.*, Portland, OR, June 1989, 294–305.
17. J. A. Orenstein and T. H. Merrett, A class of data structures for associative searching, *Proc. of the Third ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, Waterloo, Canada, April 1984, 181–190.
18. J. T. Robinson, The k-d-B-tree: a search structure for large multidimensional dynamic indexes, *Proc. of the SIGMOD Conf.*, Ann Arbor, MI, April 1981, 10–18.
19. H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.
20. H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.
21. H. Samet and R. E. Webber, Storing a collection of polygons using quadtrees, *ACM Trans. on Graphics 4*, 3(July 1985), 182–222.
22. C. A. Shaffer, H. Samet, and R. C. Nelson, QUILT: a geographic information system based on quadtrees, *Intl. Journal of Geographical Information Systems 4*, 2(April–June 1990), 103–131.
23. M. Stonebraker, T. Sellis, and E. Hanson, An analysis of rule indexing implementations in data base systems, *Proc. of the First Intl. Conf. on Expert Database Systems*, Charleston, SC, April 1986, 353–364.
24. M. Tamminen, The EXCELL method for efficient geometric access to data, *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series No. 34, Helsinki, Finland, 1981.