Benchmarking Spatial Join Operations with Spatial Output*

Erik G. Hoel†
Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Sciences
University of Maryland
College Park, Maryland 20742
hoel@cs.umd.edu and hjs@cs.umd.edu

Abstract

The spatial join operation with spatial output is benchmarked using the R-tree, R*-tree, R$^+$-tree, and the PMR quadtree. The studied quantities are the time to build the data structure and the time to do the spatial join in an application domain consisting of planar line segment data. Experiments reveal that spatial data structures based on a disjoint decomposition of space and bounding boxes (i.e., the R$^+$-tree and the PMR quadtree with bounding boxes) outperform the other structures that are based upon a non-disjoint decomposition (i.e., the R-tree and R*-tree). As the size of the output of the spatial join increases with respect to the larger of the two inputs, methods based on a disjoint regular decomposition (i.e., the PMR quadtree regardless of the presence of bounding boxes) perform significantly better.

*This work was supported in part by the National Science Foundation under Grant IRI-92-16970 and ASC-93-18183, and by a grant from the Computer Research and Applications Group at Los Alamos National Laboratory.

†Author's current address is Graphic Data Systems Corp., 6200 South Syracuse Way, Englewood CO 80111

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 21st VLDB Conference
Zürich, Switzerland, 1995

1 Introduction

A spatial join involves two data sets. It is one of the most common operations in spatial databases. The term "join" is usually used in conjunction with a relational database management system [9]. In that context, a join is said to combine entities from two data sets into a single set for every pair of elements in the two sets that satisfy a particular condition. These conditions usually involve specified attributes that are common to the two sets. In the spatial variant of the join, the condition is interpreted as being satisfied (i.e., two elements are joined) when the elements of the pair cover some part of the space that is identical.

Our results are distinct from other studies (e.g., [3, 5, 6, 12, 19, 23, 24]) in that we stress the fact that the output of a spatial join operation doesn't just have a relational component; it also has a spatial component. Thus we don't always want to just report the object pairs that intersect. In particular, we want to report their locations as well so that they can serve as input to subsequent spatial operations (i.e., a cascaded spatial join as would be common in a spatial spreadsheet). Therefore, we also need to construct a map for the output. In other words, the time to build the spatial data structure plays an important role in the benchmark, in addition to the time required to perform the spatial join itself whose output is not always required to be spatial. It is interesting to observe that the spatial join operation was not a part of the Sequoia benchmark [8, 26] where the examples of what was termed a spatial join were really window operations (i.e., a spatial selection) as the second map was usually a subset of the entire map. Thus the contribution of our paper is, in part, an additional operation to the benchmark consisting of a spatial join with a spatial output.

The spatial join problem has been studied both algorithmically and empirically for a variety of spatial data structures. Spatial join algorithms for regular grid files [14] were first investigated in [3]. The grid file was also used as the spatial data structure when the spatial join was examined from the perspective of creating a spatial join index [24]. In this case, the spatial join index simulations were on grid files using differing node-splitting rules (i.e., a regular or irregular decomposition). These simulations showed that grid files with a regular decomposition result in considerably fewer leaf node intersections between two joining structures. Spatial joins were also examined using the generalized tree [12], an abstracted hierarchical data structure similar to an R-tree [13]. Using cost models on artificial data, the generalization trees were shown to outperform join indices if there was either high data structure update rates, or high levels of join selectivity. Other studies examined the R*-tree [3] in the context of spatially joining maps composed of large polygons [5, 6]. In this case, various acceleration techniques were compared for improving cpu speed (e.g., spatial filtering) as well as I/O performance (e.g., plane-sweep search ordering). Once a candidate set of polygons was obtained, geometric filtering (e.g., employing approximations of the polygonal object) were used prior to the final exact geometry testing.

A different approach makes use of the seeded tree [19]. This structure was designed to speed the more complex spatial join process when one of the two maps being joined is the result of an intermediate operation such as a selection. The seeded tree is constructed by copying the internal node structure of one map into the second map (where the second map is assumed to be the result of an intermediate operation), and then the features are inserted into the second map. This replication of the internal node structure greatly accelerates the join process as there is a one-to-one mapping between internal nodes in the two maps. The application of global clustering (i.e., the association of spatially adjacent spatial objects with physically consecutive disk pages) to a modified R*-tree was studied in [4]. Modified R*-trees (R*-trees without forced feature reinsertion) with global clustering were found to be more expensive in terms of cpu construction costs and data storage requirements than the standard unclustered R*-tree. Experimentation with the clustered R*-tree did show, however, that spatial joins were significantly improved, primarily because of greatly decreased I/O costs. Finally, in the data-parallel domain, the spatial join has been studied in the algorithmic and empirical context [17, 18]. Experimentation indicated that the data-parallel PMR quadtree [21] significantly outperformed data-parallel R-trees and R$^+$-trees [10] primarily because the PMR quadtree's regular decomposition is well-suited to the data-parallel domain. In the data-parallel domain, communication bottlenecks during a spatial join are greatly reduced by the regular decomposition of the PMR quadtree and the ability to quickly correlate a region in one map with a corresponding region in a second map. This ability to correlate regions will be seen to have a similar effect on the performance in the sequential domain as the size of the output of the spatial join increases with respect to the larger of the two inputs.

The rest of this paper is organized as follows. Section 2 gives a brief review of the six spatial data structures that we consider. Section 3 details the spatial join algorithms that are tested. Section 4 presents the execution times, disk I/Os, and storage requirements for the construction of the different data structures as well as their performance in a spatial join. Section 5 contains our conclusions about the relative performance of the different data structures.

2 Spatial Data Structures

In this paper we consider representations that sort the data objects with respect to the space that they occupy. This results in speeding up operations involving search. Our objects consist of lines. The effect of the sort is to decompose the space from which the data is drawn (e.g., the two-dimensional space containing the lines) into regions called *buckets*. One approach known as an R-tree [13] buckets the data based on the concept of a minimum bounding (or enclosing) rectangle. In this case, lines are grouped (hopefully by proximity) into hierarchies, and then stored in another structure such as a B-tree [7]. The drawback of the R-tree is that it does not result in a disjoint decomposition of space — that is, the bounding rectangles corresponding to different lines may overlap. Equivalently, a line may be spatially contained in several bounding rectangles, yet it is only associated with one bounding rectangle. This means that a spatial query may often require several bounding rectangles to be checked before ascertaining the presence or absence of a particular line. In this paper, we study two methods of this type: the R-tree (both linear and quadratic), and the R*-tree.

The non-disjointness of the R-tree is overcome by a decomposition of space into disjoint cells. In this case, each line is decomposed into disjoint sublines such that each of the sublines is associated with a different cell. There are a number of variants of this approach. 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 line, we have to retrieve all the cells that it occupies. This means that an object may be reported as satisfying a particular query more than once and thus there is a need to remove duplicate answers (e.g., [2]). Here we study two methods of this type: the R+-tree [10] and the PMR quadtree [21].

The R+-tree partitions the lines into groups of arbitrary sublines having disjoint bounding rectangles which are grouped in another structure such as a B-tree. The groupings are such that the bounding rectangles are disjoint at each level of the structure. The drawback of the R+-tree is that the decomposition is data-dependent. This makes it more complex to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations such as overlay). In contrast, the PMR quadtree is based on a regular decomposition. The space containing the lines is recursively decomposed into four equal-area blocks on the basis of the number of lines that it contains (termed a *splitting threshold*). The decomposition process can be implemented by a tree structure. It is useful for set-theoretic operations as the partitions of the two data sets occur in the same positions.

As mentioned above, R-trees and R+-trees are closely related to B-trees. An R-tree or R+-tree of order (m, M) has the property 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 node has at least 2 entries unless it itself is a leaf node. Often the nodes correspond to disk pages. 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. 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. It is interesting to observe that the node capacity (i.e., bucket capacity) M in the R-tree and R+-tree plays a similar role as the splitting threshold in the PMR quadtree. We will make use of this analogy in our discussion where, at times, the terms will be used interchangeably. In the rest of this section we describe these data structures in more detail.

2.1 R-tree

Figure 1a is an example R-tree with $M = 3$ and $m = 2$ for the collection of line segments labeled $a - i$. Figure 1b shows the spatial extent of the bounding rectangles of the nodes in Figure 1a, 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.

The algorithm for inserting a line segment (i.e., a record corresponding to its enclosing rectangle) in an R-tree is analogous to that 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 causes the node to overflow. If yes, then split the node and distribute the $M + 1$ records in the two nodes. Splits are propagated up the tree.

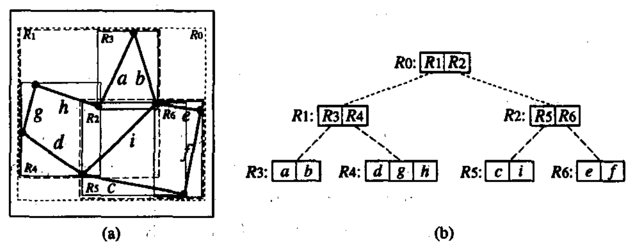

Figure 1: (a) The spatial extents of the bounding rectangles and (b) the R-tree for the example collection of line segments.

There are many possible ways to split a node. Guttman [13] distributes 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 area of the covering rectangles for the nodes (i.e., coverage).

Node-splitting algorithms that employ this strategy are classified by their computational complexity [13]. The linear algorithm selects two seed elements, one for each of the two resulting nodes, by choosing the pair of children with the largest normalized separation along any axis. This operation can be done in linear time. The remaining $M - 1$ children, where M is the node capacity, are then inserted in random order into one of the two resulting nodes whose covering rectangle will have to be expanded the least to accommodate the child. Ties are resolved by inserting the child into the node with the smaller area. We refer to the R-tree with this node-splitting rule as the *linear R-tree*. The quadratic node-splitting algorithm, selects the two seed elements by choosing the two children whose covering bounding box would waste the most amount of space. This can be determined in quadratic time $(O(M^2))$. The remaining $M - 1$ children are inserted into the two nodes in an order dependent upon the magnitude of their preference (i.e., the difference between the area increase required of each node for inclusion of the child) for one of the two resulting nodes. As is done with the linear algorithm, the children are inserted into the node whose covering rectangle will have to be expanded the least to accommodate it, with ties being resolved by inserting the child into the node with the smaller area. We refer to an R-tree built with this node-splitting rule as the *quadratic R-tree*.

2.2 R*-tree

The R*-tree [3] is a variant of the R-tree that uses more sophisticated node-insertion and node-splitting algorithms thereby reducing the storage requirements. When deciding which node is to contain the new line segment, it chooses 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). This reduces 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 found by examining all 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 which minimizes the sum of the perimeters of the two constituent nodes. If there is a tie, then choose one of the axes at random. Once the axis has been chosen, say the x-axis, choose the split among the $M - 2m + 2$ possibilities that results in a minimal amount of overlap between the two new constituent nodes. If there is a tie, then choose the split that minimizes the total area of the two new constituent nodes.

2.3 R$^+$-tree

The R$^+$-tree [10] partitions the lines into groups of arbitrary sublines having disjoint bounding rectangles which are grouped in another structure such as a B-tree. These sublines are termed *q-edges* [25]. The partition and the subsequent groupings are such that the bounding rectangles are disjoint at each level of the structure. An example R$^+$-tree is shown in Figure 2.

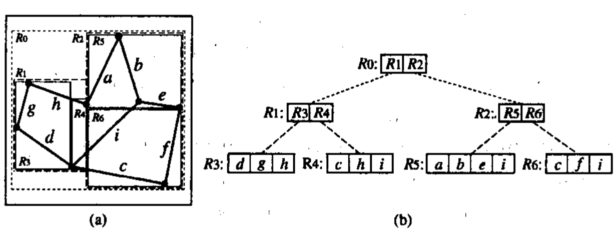

Figure 2: (a) The spatial extents of the bounding rectangles and (b) the R$^+$-tree for the example collection of line segments.

2.4 PMR Quadtree

The PMR quadtree (denoting *polygonal map random* [21, 22]) is an edge-based member of the PM quadtree family (see also edge-EXCELL [27]). It makes use of a probabilistic splitting rule where a block is permitted to contain a variable number of line segments. The PMR quadtree 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. During this process, the occupancy of each affected block is checked to see if the insertion causes it to exceed a predetermined *splitting threshold*. Note that the concept of a splitting threshold, although closely related, is different from the concept of a bucket capacity[1]. If the splitting threshold is exceeded, then the block is split *once*, and only once, into four blocks of equal size. The rationale for the use of a splitting threshold is to avoid splitting a node many times when there are a few very close lines in a block whose number exceeds the bucket capacity. In this manner, we avoid pathologically bad cases that would occur when a collection of line segments has endpoints that are very close together. This would result in a large number of subdivisions in order to separate the endpoints (for more details of this pathological behavior, see [21]).

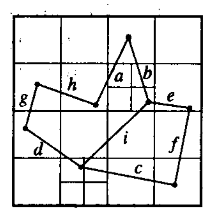

Figure 3: PMR quadtree with a splitting threshold of two for a collection of line segments.

Figure 3 is an example of a PMR quadtree corresponding to a set of 9 edges labeled a through i inserted in increasing lexicographical order. Observe that the shape of the PMR quadtree for a given data set is not unique; instead, it depends on the order in which the lines are inserted into it. This example assumes that the splitting threshold value is two. Generally, as the splitting threshold is increased, the construction times and storage requirements of the PMR quadtree decrease while the time necessary to perform operations on it will increase.

It is interesting to point out that although a block can contain more line segments than the splitting threshold, this is not a problem. In fact, it can be shown that the maximum number of line segments in a block 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), provided that the block is not at the maximal depth allowed by the particular implementation of the PMR quadtree [25].

The PMR quadtree often acts as an adaptive grid to index various blocks which contain spatial data. In solid modeling, the quadtree blocks contain complex spatial objects such as B-splines, Bezier curves, sur-

[1] In our discussions, the concept of a *bucket* and a *block* are the same, and we use the terms interchangeably.

face patches, etc. [20]. Frequently, a bounding box is stored in the node so that the physical extent of the object can be determined easily. This is also the case with each of the studied R-tree variants. In the general case, the PMR quadtree should have a bounding box (or some other such approximation) around each feature. In most of the previous studies (e.g., [15, 16, 21, 22]), bounding boxes were not employed with PMR quadtrees containing point or line data, though they have been with quadtrees containing more complex spatial objects. They were omitted for point and line PMR quadtrees primarily as a performance optimization. In this paper, in addition to the customary PMR quadtree we also study variant of the PMR quadtree which, like the R-trees, associates a bounding box with each feature or object identifier tuple stored in the quadtree leaf nodes. As we will see, incorporating the bounding box information increases the size of each tuple stored in the quadtree. However, this increase in size may be compensated by improved spatial join performance when the size of the output of the spatial join is not too large with respect to the larger of the two inputs.

3 Spatial Join Algorithms

For each spatial data structure that we consider, we assume that the physical representation of the spatial objects (in our case the coordinate values of the line segment endpoints) are stored in a secondary buffered array structure (termed the *feature table*). Within the spatial data structures, only descriptors (or pointers) to the objects in the feature table are stored.

3.1 R-trees and R$^+$-trees

Each of the R-tree variants employed a spatial join algorithm similar to one described in [6] in the context of polygon map spatial joins. The spatial join algorithm uses techniques that are intended to decrease both cpu time consumption and disk I/O. These techniques, restricting the search space, and employing a local plane-sweep order with pinning are detailed in [6]. The R-tree[2] spatial join algorithm is a coordinated tree traversal that begins with the two root nodes. For the two nodes being considered (initially the two root nodes), their bounding boxes are intersected to determine the overlapping area O between the two nodes. The children of each node are then compared against O. If a child does not intersect O, then it cannot intersect any children in the other map and is removed from further consideration. For all children in the first node that intersect O, their bounding boxes are intersected against the bounding boxes of all children in the second node that also intersect O. All intersections between the two sets of children are recorded.

[2]For sake of brevity, we will use the term *R-tree* in this discussion though the described algorithm can be applied to each of the four R-tree variants (including the R$^+$-tree).

Once a set of intersecting children has been determined, the node intersection process is recursively applied to each pair of intersecting child nodes. The child nodes are considered in an order based upon their plane-sweep order with pinning. Pinning basically keeps (or "pins") in main memory the child node in one map which intersects the largest number of child nodes in the second map which have not yet been processed. By employing the pinning technique, in addition to a plane-sweep order, a read schedule for the child nodes may be determined.

If a pair of bounding boxes is found to intersect, then, if the children correspond to leaf nodes, the associated line segment must be read from the feature table and the two lines are then intersected. Otherwise, if the two children are internal nodes, then the algorithm is recursively applied to the intersecting subnodes within the two children. This process terminates when all intersecting nodes have been considered.

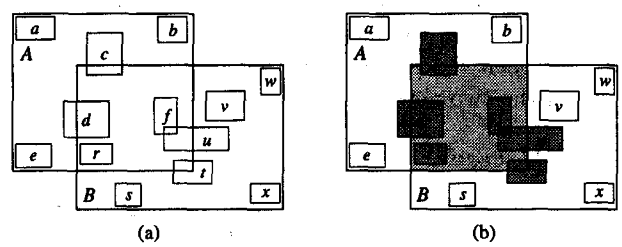

Figure 4: Example of two-stage child intersection detection.

In Figure 4a, two intersecting leaf nodes (labeled A and B) are shown. The bounding boxes of the objects contained in leaf node A correspond to the rectangles labeled $a - f$, while the bounding boxes of the object contained in leaf node B are labeled $r - x$. The base region of intersection between leaf nodes A and B is shown in Figure 4b as the light shaded region. The bounding boxes of objects contained in leaf node A that intersect the base region of intersection, correspond to the dark shaded rectangles in Figure 4b (objects c, d, and f). Similarly, the bounding boxes of objects contained in leaf node B that intersect the region of intersection are represented in Figure 4b by the dark shaded rectangles labeled r, t, and u. The set of bounding boxes (c, d, f) must be then intersected against the set (r, t, u). If any bounding boxes are found to intersect, then the corresponding features (lines) must be read from the feature table and an exact intersection check is performed. In Figure 4b, the two bounding boxes that intersect are labeled f and u. As will later be discussed and detailed in Table 3, this two-stage process dramatically reduces the number of intersection checks that must be performed during a spatial join for the R-tree variants.

3.2 PMR Quadtrees

The algorithm for performing a PMR quadtree spatial join is basically a simple synchronized tree traversal at the leaf level. Each quadtree node is visited in the order prescribed by the tree structure. If the joining leaf node in each quadtree is the same size, then all of the line segments in the first node are intersected with all of the line segments in the second node. If one of the joining leaf nodes is larger than the other leaf node, then the lines in the first node are intersected with all the lines in the second smaller leaf node. Once all the intersections have been performed, then the larger leaf node is joined with the next smaller leaf node in the second map. This process is repeated for all small nodes that correspond to the single larger node. If one of the two joining nodes is empty, then the two nodes are skipped. The process is completed when each quadtree has been traversed in its entirety.

As a performance optimization for the PMR quadtree with bounding boxes, before performing the line segment intersection, the two corresponding line segment bounding boxes are first checked for intersection. This is a much simpler task and can greatly speed the spatial join process. Otherwise, two I/O operations may be required for an intersection as the coordinates values of the line segment endpoints are stored in the buffered feature table. One significant advantage of the PMR quadtree spatial join algorithm is that each node of the two joining PMR quadtrees is only visited once. This is in direct contrast to the R-tree spatial join algorithm, where any given leaf node may be visited many times due to the irregular decomposition of space.

4 Experimental Results

The performance of the six spatial structures (linear R-tree, quadratic R-tree, R*-tree, R+-tree, PMR quadtree, and PMR quadtree with bounding boxes) is compared using TIGER/Line File [28] maps comprising the Washington DC metropolitan area (containing approximately 260,000 line segments[3]). Extracts from this collection of data were made in order to obtain four disjoint data sets. The first extract, termed *roads*, consists of all line segments corresponding to the road network of the Washington area. The roads data set includes 200,482 lines. The second extract, termed *water*, is composed of all hydrological features in the Washington area (37,495 lines). The third extract, termed *boundary*, consists of the 18,505 lines that correspond to all non-visible boundaries in this area (i.e., Zip Code boundaries, town boundaries, political boundaries, etc.). The fourth extract, termed *non-roads*, contains the 59,601 lines that correspond to all non-road features in this area (i.e., water features, boundary features, railroads, pipelines, landmarks, etc.). The non-roads data set is a proper superset of the water and boundary data sets. In order to test the sensitivity of the performance of the operations to the size of the output (i.e., the number of intersections), we also constructed a number of artificial data sets by extracting line segments at random from the entire data set for the Washington DC area.

In addition to employing standard metrics for the performance comparisons such as disk I/Os, feature intersection tests, and data structure size, we also measure cpu execution times (elapsed times) including the time to access the feature table. We have observed that although some structures may exhibit superior performance with respect to other structures in terms of disk I/Os, their cpu times may be significantly larger (e.g., data structure construction time for the R*-tree and R+-tree). Note that all performance tests are made using a buffer size of 128 KB on a 90 MHz Pentium (90.1 SPECint92, 72.7 SPECfp92)[4]. All pages are 1 KB for the PMR quadtree while for the R-tree variants, the number of bytes per page is 20 bytes times the node capacity. Tests were run a sufficient number of times to get a consistent execution time.

Below, we first discuss the time necessary to build the data structures followed by the time to perform the spatial join. The build times will be seen to be an important factor in the performance of a spatial join with a spatial output. We also measure the execution time as a function of the size of the output (i.e., the number of intersections). This turns out to be the key factor in the performance and can be seen by examining Figure 13. Actual conclusions about the relative merits of the different data structures are made in Section 5.

4.1 Data Structure Construction

Table 1 details data structure construction performance for the six spatial structures using a node capacity of 50 for the R-tree variants, and a splitting threshold of 8 for the PMR quadtrees for the roads data set. These values were chosen as they are commonly used in previous studies (e.g., [3, 10, 13, 16]), and they provide a reasonable compromise between optimal build and join performance. The PMR quadtree was implemented using a linear quadtree [1, 11] which is a pointer-less representation that stores the leaf nodes of the quadtree in a B+-tree [7]. In the table,

[3] The regions comprising this data set are Washington DC, Montgomery Co., Prince Georges Co., Arlington Co., Alexandria, VA, Fairfax Co., Fairfax, VA, and Falls Church, VA.

[4] Increasing the buffer size has not led to observed dramatic decreases in execution time [6].

spatial structure	time	disk I/Os	splits	storage
R-tree (linear)	304	21,922	6,482	7,110
R-tree (quadratic)	318	20,519	6,539	7,145
R^+-tree	276	29,135	8,180	8,152
R^*-tree	2,139	21,127	5,648	6,599
PMR quadtree	246	19,099	5,057	8,195
PMR (w/bboxes)	258	24,613	5,308	10,051

Table 1: Construction performance of the six spatial data structures on the roads data set. For the PMR quadtree, the node splits are B^+-tree node splits.

"node splits" for the PMR quadtrees actually corresponds to the number of B^+-tree page splits in the linear quadtree representation. The actual number of quadtree node splits is 32,737.

Mirroring results from an earlier study that compared the R^*-tree, the R^+-tree, and the PMR quadtree [16], we find that the disjoint decompositions (i.e., the PMR quadtrees and the R^+-tree) exhibit better performance in terms of cpu time relative to the other non-disjoint decompositions (i.e., the R-trees and the R^*-tree). Their build times are roughly ten times faster than that of the R^*-tree, and 20–30% faster than the linear and quadratic R-trees. The R^*-tree's performance suffers from several computationally expensive operations that occur during the course of inserting a line segment. For example, the `ChooseSubtree` procedure (as defined by Beckmann *et al.* [3]) is used to select the appropriate insertion path. This insertion path selection operation requires $O(M^2)$ bounding box operations for each line segment insertion, where M is the node capacity. We observed that this single operation consumed approximately 30% of the time spent constructing the structure. Additionally, the node-splitting procedure, where 30% of the lines are reinserted when a node overflows, resulted in the forced reinsertion of 343,364 line segments (an overhead of 171% additional line insertions).

In terms of disk I/O, the PMR quadtree required the fewest operations (19,099). Its performance was nearly equaled by the quadratic R-tree (20,519). The R^*-tree and the linear R-tree required approximately 10% more disk I/Os (21,127 and 21,922 respectively), while the PMR quadtree with bounding boxes consumed 30% more disk I/Os (24,613). Finally, the R^+-tree was the most disk I/O intensive, requiring over 50% more (29,135) than the quadratic R-tree. Incorporating bounding boxes in the PMR quadtree did not significantly affect build times (a 5% increase), but it did result in increased amounts of disk I/O relative to the standard PMR quadtree (a 30% increase; 24,613 versus 19,099). This increase is due primarily to having fewer tuples on each page of the B^+-tree; 60 versus 120 due to the need to store the bounding boxes.

In terms of storage requirements, the R^*-tree used the fewest resources (6,599 KB), consuming approximately 10 – 20 % less space than the other R-tree variants (7,110 – 8,152 KB). The R^*-tree (6,599 KB) used 20% less space than the standard PMR quadtree (8,195 KB) while using 35% less space than the PMR quadtree with bounding boxes (10,051 KB).

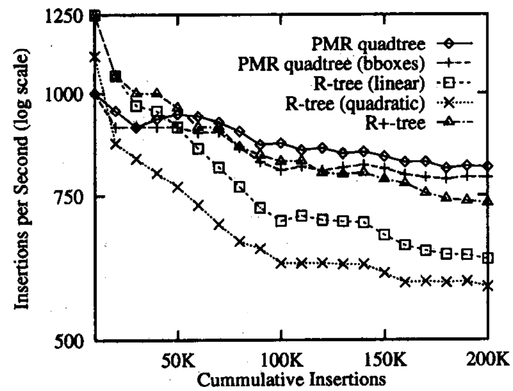

Figure 5: Lines per second construction speeds on the roads data set.

It is also interesting to observe the slowdown experienced by each data structure as the number of lines in the structure grows. In Figure 5, the number of line insertions per second on the roads data set is plotted for all but the R^*-tree structure[5]. From the figure, the R^+-tree's insertion performance is roughly 1250 lines per second for the first 10,000 lines of the roads data set. This rate falls to 735 lines per second (cumulative) by the time the build operation is completed after inserting 200,482 lines. Each of the other structures exhibits similar performance decreases, though none as steep as the R^+-tree. These decreases are expected and are due to the increased height of the tree structures. Interestingly, in an earlier study [16], the R^+-tree was reported as exhibiting the fastest construction times relative to the R^*-tree and the PMR quadtree. That study was performed using data sets whose size was on the order of 50,000 line segments. From Figure 6, we see that the R^+-tree outperforms all other structures up through 50,000 line insertions. As the number of line insertions grows toward 200,000, we observe that the R^+-tree's performance decreases faster than the two PMR quadtrees. This results in the two PMR quadtrees outperforming the R^+-tree by 7–12% on the larger data sets used in this study.

Figure 6 shows the construction times for the PMR quadtree, both with and without bounding boxes, for the roads data set. From the figure, it can be observed that as the splitting threshold increases, the build times decrease. This is due to fewer node splits and

[5] The R^*-tree, which is omitted from the figure, exhibits performance starting at 109 segments per second, and falling slightly to 104 segments per second by the completion of the build operation.

a shallower tree structure. As the splitting threshold grows past 30, build times begin to increase slightly. This is because the PMR quadtree nodes begin to occupy a significant portion of the B+-tree pages, and PMR quadtree nodes are more likely to exist on more than one page. This increases the amount of time required to perform basic node manipulations.

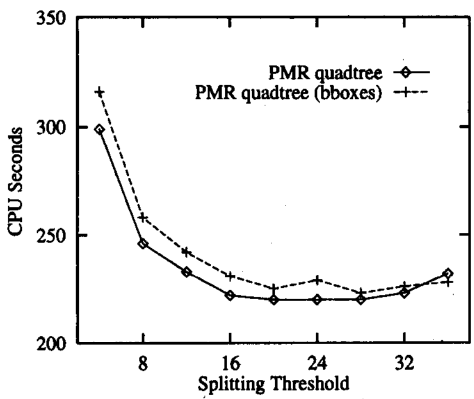

Figure 6: Construction times for PMR quadtrees on the roads data set for varying splitting threshold values.

Figure 7 shows construction times for R-tree variants other than the R*-tree on the roads data set as a function of node capacity. The figure shows that build times fall dramatically between node capacities 75 and 100. This is due to the height of the R-trees decreasing by one. The build times then begin to increase as the node capacity grows past 100 because of the increased expense of determining which node to insert a line segment into, as well as the increased cost of splitting a node. Note the relative rate of build time increase for the linear and quadratic R-trees. In particular, the construction time for the quadratic R-tree increases at a faster rate than the linear R-tree because of the more expensive node-splitting algorithm (i.e., $O(M^2)$ versus $O(M)$, where M is the node capacity).

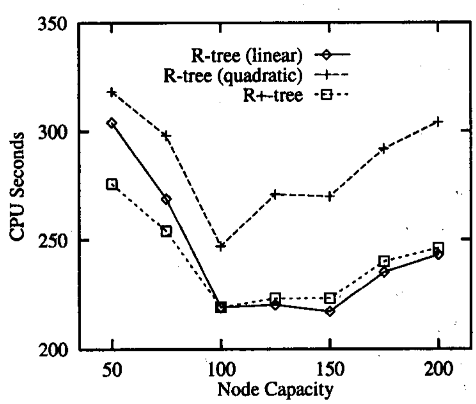

Figure 7: Construction times for R-trees on the roads data set for varying node capacities.

Figure 8 shows the construction times for the R*-tree on two data sets (roads and water) for various node capacities. As we can see, the R*-tree exhibits a significant decrease in construction performance as the node capacity M increases. This is primarily because the R*-tree construction algorithm requires $O(M^2n)$ bounding box intersections, where n is the size of the input data set. The double log plot of the construction times highlights this relationship, with the performance curve appearing linear. For small node capacities (i.e., 50), building the R*-tree is roughly one order of magnitude slower than any of the other data structures. For larger node capacities (i.e., 200), building the R*-tree is approximately two orders of magnitude slower than the other data structures.

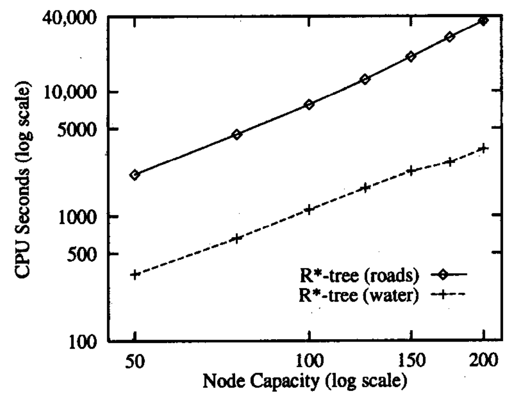

Figure 8: Double logarithmic plot of construction times for R*-trees on the roads (200,482 lines) and water (37,495 lines) data sets for varying node capacities.

4.2 Spatial Join Performance

The performance of the spatial join was measured for each of the six spatial structures using the four extracted data sets (roads, non-roads, water, and boundary). Two joins are studied in greater detail. The first joins the roads data set and the water data set, resulting in 6,404 intersections. The second joins the roads and the boundary data set, resulting in 10,983 intersections. Other joins were also tested so that we could see the effect of the size of the output (the number of intersections). The performance of each data set was measured when the join resulted in the generation of an output map containing all points of intersection (termed *spatial output*), and when it resulted in a list of tuples containing identifiers of the intersecting lines (termed *non-spatial output*).

4.2.1 Roads and Water Spatial Join

Table 2 summarizes the performance of each data structure on the roads versus water spatial join (node capacity fifty, splitting threshold eight). For spatial joins which result in a spatial output, the PMR quadtree with bounding boxes outperformed the other structures in terms of cpu time (155 seconds). It was

spatial structure	spatial time	spatial I/Os	non-spatial time	non-spatial I/Os
R-tree (linear)	162	11,514	143	11,310
R-tree (quadratic)	157	11,501	137	11,298
R^+-tree	164	11,452	153	11,171
R^*-tree	191	8,575	142	8,372
PMR quadtree	211	6,137	198	5,851
PMR (w/bboxes)	155	6,233	141	5,953

Table 2: Spatial join of the roads and water data sets.

trivially faster than the next fastest structure, the quadratic R-tree with 157 seconds. Adding bounding boxes to the PMR quadtree reduces the join time relative to the standard PMR quadtree by almost 30% (155 seconds versus 211 seconds).

In terms of disk I/O, each of the PMR quadtrees required considerably fewer operations (6,137 – 6,233 disk I/Os) than any of the R-tree variants (8,575 – 11,514 disk I/Os). This is primarily due to the ability of the PMR quadtree, as well as any other spatial structure employing a regular decomposition of space, to rapidly spatially correlate the contents of one map with another. With the regular decomposition of the PMR quadtree, there will exist either a one-to-one, one-to-many, or many-to-one mapping between the two joining data sets at the leaf level. This is in contrast with the R-tree variants which will often have a many-to-many mapping between joining data sets. The many-to-many mapping between leaf nodes among two different data sets prevents a simple traversal of each data set where each page is read into memory a single time. The more complex the many-to-many mapping, the more often a page must be read from disk. The incorporation of bounding boxes into the PMR quadtree accelerates the join process with respect to that for a PMR quadtree without bounding boxes as considerable amounts of pruning can be done at the leaf node level thereby saving accesses to the secondary storage structure (the buffered feature table). The more sophisticated and computationally expensive node-splitting rule utilized by the R^*-tree resulted in considerably fewer disk I/Os as compared with the other R-tree variants (8,575 versus 11,452 – 11,514). This large decrease in disk I/Os was offset by the increased amount of time necessary to construct the output map thereby resulting in the R^*-tree taking 27 – 34 more seconds than the other R-tree variants.

Table 3 highlights the number of intersection tests that are performed on each structure during a spatial join of the roads and water data sets. For example, the linear R-tree performs 24,814 line-to-line intersection tests corresponding to the intersecting bounding boxes in the two datasets, which, of course, are the same for each of the other non-disjoint R-tree variants, the quadratic R-tree and the R^*-tree. The linear

spatial structure	pairs tested: lines	naive lines	internal	leaf
linear R	24,814	17,325,074	4,582	15,674
quad R	24,814	16,219,892	4,889	14,863
R^+	25,583	16,554,775	3,560	16,069
R^*	24,814	14,408,052	2,903	9,805
PMR	1,267,939	1,267,939	—	323,804
PMR (bb)	37,118	1,267,939	—	323,804

Table 3: Spatial join data for the roads and water data sets detailing line and node testing. In the table "R" denotes R-tree; "R^*" denotes R^*-tree; "PMR (bb)" denotes PMR quadtree with bounding boxes.

R-tree also had 4,582 internal node intersection tests between the joining structures, as well as 15,674 leaf node intersection tests. The column labeled "naive lines" corresponds to the number of line intersection tests that would be required if bounding boxes and spatial filtering were not employed in the spatial join algorithm (see Figure 4). Bounding boxes and spatial filtering are very simple techniques for spatial join acceleration, even for simple features such as line segments. From Table 3, the incorporation of bounding boxes into the PMR quadtree reduced the join time by 57 seconds (27%), with the number of line versus line intersection tests falling from over 1.2 million to 37,118, a 97% reduction. The trade-off is the increased storage requirement for the bounding boxes resulting in the PMR quadtree without bounding boxes occupying 1,856 KB less disk space (18.5%; see Table 1).

The number of internal node and leaf node intersection tests (as shown in Table 3) is a useful measure of the "goodness" of the spatial decompositions. From the table, the R^*-tree requires the fewest node intersection tests, both internally and at the leaf level as compared with the other R-tree variants. The R^+-tree requires the largest number of leaf node intersection tests. This is not surprising as the R^+-tree has a disjoint decomposition which results in each line segment possibly being represented more than once inside the structure. The PMR quadtrees are not directly comparable in the node intersection test sense as the tested quadtree implementation is pointerless thus having no internal nodes (as opposed to pointer-based, see [11]), and the node size is much smaller given a splitting threshold of eight.

Join times without spatial output are nearly equivalent for all structures except the PMR quadtree without bounding boxes (137 – 153 seconds). Surprisingly, for the roads and water spatial join, the quadratic R-tree slightly outperforms the R^*-tree (137 seconds versus 142 seconds).

Figures 9 and 10 show execution times for a spatial join with a spatial output of the roads and water data sets for the six structures when the node capacities and splitting thresholds are allowed to vary. The two

PMR quadtrees exhibit optimal performance at differing splitting thresholds. In Figure 9, the standard PMR quadtree performs best with a splitting threshold of 8 to 12, while the PMR quadtree with bounding boxes performs the best with splitting thresholds of approximately 20. The PMR quadtree with bounding boxes outperforms the standard PMR quadtree primarily because the bounding boxes facilitate pruning the number of line intersection tests required to join two quadtrees (recall the "lines" and "naive lines" pairs tested entries in Table 3).

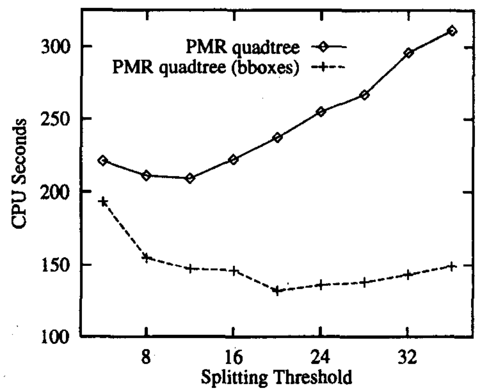

Figure 9: Execution times for a spatial join for the roads and water data sets with spatial output for PMR quadtrees.

In Figure 10, we observe that the R+-tree outperforms all other R-tree variants for all tested node capacities. Interestingly, the R*-tree, which performed relatively poorly for the smaller node capacities (50 – 100), exhibited good performance for the larger node capacities (150 – 200). Unfortunately, the build times for the R*-tree are significantly larger than for the other data structures for large node capacities and larger data sets (refer to Figures 7 and 8).

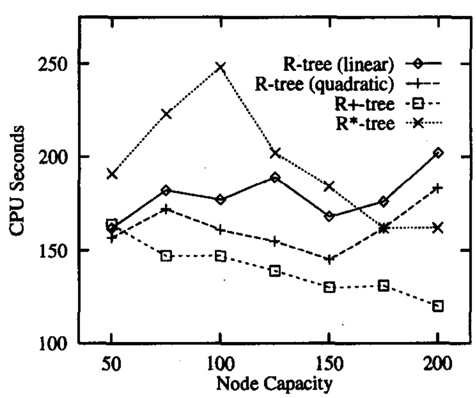

Figure 10: Execution times for a spatial join of the roads and water data sets with spatial output for the R-tree variants.

Figure 11 shows the disk I/O performance of the PMR quadtree when the node capacities and splitting thresholds are allowed to vary in the case of spatial output for the spatial join of the roads and water data sets. Both PMR quadtrees exhibit decreasing amounts of disk I/O as the splitting thresholds increase. This is due, in part, to the decreased size of the data structures. In particular, as the splitting threshold increases, the number of q-edges decreases, asymptotically approaching 1.0. The amount of additional disk I/O required by the PMR quadtree with bounding boxes is not significant.

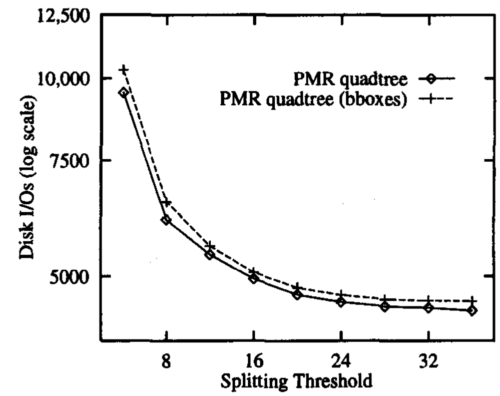

Figure 11: Spatial join of the roads and water data sets with spatial output disk I/Os for PMR quadtrees.

Figure 12 shows the disk I/O performance of the R-tree variants for different node capacities as was done for the PMR quadtrees in Figure 11. Not surprisingly, the R*-tree, with its expensive node-splitting rule, exhibits the best performance and outperforms the other R-tree variants. The other R-tree variants require 15 – 75% more disk I/Os than the equivalent R*-tree. As the node capacity increases, the amount of disk I/O decreases for these structures.

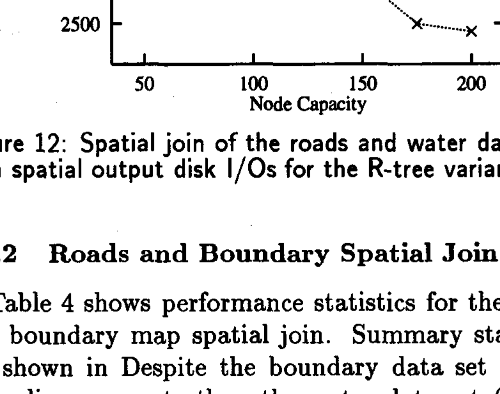

Figure 12: Spatial join of the roads and water data sets with spatial output disk I/Os for the R-tree variants.

4.2.2 Roads and Boundary Spatial Join

Table 4 shows performance statistics for the roads and boundary map spatial join. Summary statistics are shown in Despite the boundary data set having fewer line segments than the water data set (18,505

spatial structure	spatial time	spatial I/Os	non-spatial time	non-spatial I/Os
R-tree (linear)	180	11,733	156	11,254
R-tree (quadratic)	194	12,738	168	12,288
R^+-tree	176	10,543	165	10,030
R^*-tree	246	9,376	172	8,970
PMR quadtree	217	6,065	197	5,410
PMR (w/bboxes)	162	6,194	141	5,521

Table 4: Spatial join of the roads and boundary data sets.

spatial structure	spatial time	spatial I/Os	non-spatial time	non-spatial I/Os
R-tree (linear)	232	13,026	197	12,064
R-tree (quadratic)	255	14,091	218	13,086
R^+-tree	231	12,785	211	11,828
R^*-tree	380	11,935	226	11,097
PMR quadtree	256	7,709	224	6,290
PMR (w/bboxes)	247	8,467	215	6,365

Table 5: Spatial join of the roads and non-roads data sets.

and 37,495 lines respectively), there were more intersections detected when joining the roads and boundary data sets (10,983 versus 6,404 for the roads and water spatial join). The most interesting difference between this spatial join and one described earlier (roads and water) is the relative performance of the R^*-tree. Because the roads and boundary spatial join has almost twice as many reported intersections, and the build time for constructing the spatial output for R^*-tree joins is considerably higher than for the other spatial structures, the R^*-tree's performance with a spatial output declines relative to the other structures. For the roads and boundary spatial join, the R^*-tree is 52% slower than the fastest structure (the PMR quadtree with bounding boxes), while it was only 23% slower than the fastest structure (again the PMR quadtree with bounding boxes) for the smaller roads and water spatial join. In contrast, the performance of the R^+-tree only declined from 6 to 9% slower than the PMR quadtree with bounding boxes. Based upon these two spatial joins, and coupled with the data structure build statistics described in Figures 7 and 8, it is clear that as the size of the spatial join output increases, the relative performance of the R^*-tree will continue to decline. Note that since the data sets are quite different in terms of locality, the number of disk I/Os may decrease or show little change even though the size of the output increases. For example, see the PMR quadtree in Tables 2 and 4.

4.2.3 Roads and Non-roads Spatial Join

Table 5 corresponds to the spatial join of the roads and non-roads data sets. This is a larger spatial join, both in terms of both input and output map sizes (18,739 intersections). Many of the previously observed performance differences between the six spatial structures (see Tables 2 and 4) become even more pronounced with the larger data sets. Most notably, the spatial structures that are faster to build (the R-trees, the R^+-tree, and the PMR quadtrees) outperform the R^*-tree (231 – 256 seconds versus 380 seconds, respectively). Observe again that despite comparable performance when there is no spatial output, the R^*-tree's performance deteriorates much more, in a relative sense, than the other structures when there is spatial output.

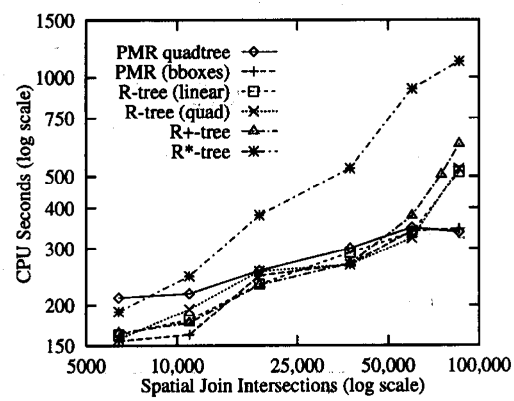

Figure 13: Execution times for a spatial join with spatial output according to output size.

Figure 13 displays the execution times of a spatial join with spatial output according to the number of intersecting lines determined by the spatial join. The data is taken from Tables 2, 4, and 5, as well as some artificial data sets formed by extracting line segments at random from the entire data set for the Washington DC area. From the figure, it is apparent that as the number of intersections found in the spatial join increases, the disjoint decompositions outperform the non-disjoint decompositions. The implications of this conclusion are discussed in greater detail in Section 5.

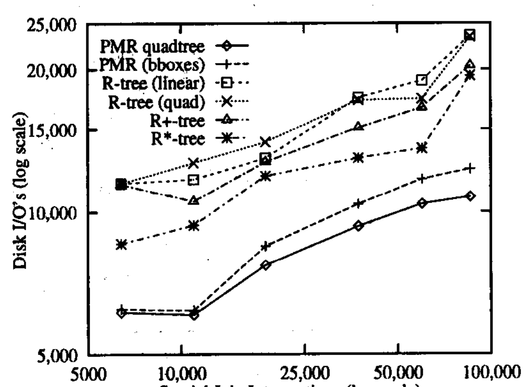

Figure 14: Disk I/Os for a spatial join with spatial output according to output size.

Figure 14 shows the disk I/Os for a spatial join with spatial output for according to the number of intersecting lines determined by the spatial join. It is apparent from the figure that the two PMR quadtrees with their regular decomposition outperform the R-tree variants across the spectrum of spatial join output sizes.

5 Comparison of the Structures

Our experiments (most notably Figure 13) have revealed a number of interesting results. Most importantly, they show that when the output of the spatial join is spatial, then spatial data structures based on a disjoint decomposition of space (the R$^+$-tree and the PMR quadtree) outperform spatial data structures based on a non-disjoint decomposition such as the numerous variants of the R-tree including the R*-tree. This difference is primarily because of the need to build the data structure as part of the output.

These differences in execution time and disk I/Os become more pronounced as the output becomes larger. In our tests, the difference became significant when the output size was 25% or more of the larger of the two inputs. This is especially true for the spatial data structures based on a regular decomposition such as the PMR quadtree with respect to the R$^+$-tree and to an even greater extent with respect to the R*-tree. This difference is primarily because the bounding box information which is used so effectively in the R-tree variants and the PMR quadtree with bounding boxes to limit the number of lines that must be tested for possible intersection is no longer as useful. The reason is that bounding boxes do not prune enough of the intersections when the output size is large. An alternative explanation of this result is obtained by noting that R-trees and R$^+$-trees are particularly useful in distinguishing between occupied and unoccupied space. In these examples, most of the space is occupied, thereby diminishing the utility of these representations.

In contrast, representations based on a regular decomposition are more useful in such an environment as they provide a correlation between occupied space in the two data sets that are being joined. This was verified by our observations that as the size of the output increased, the use of bounding boxes with the PMR quadtree did not lead to a significant improvement in performance (Figure 13) whereas it did so when the output was smaller in Tables 2 and 4. Moreover, as the size of the output becomes larger, the bounding boxes in the PMR quadtree need more nodes as each node contains fewer line segments due to the inclusion of the bounding boxes. Thus more node intersections must be performed each of which may require a disk I/O operation thereby canceling the effect of the pruning resulting from the use of the bounding boxes.

When the output of the spatial join is not required to be spatial, then the R*-tree has comparable performance to that of the R$^+$-tree and the two variants of the PMR quadtree as long as the output is considerably smaller than that of the larger of the two inputs (10%). However, as the output gets larger, the R*-tree has been observed to require (not shown here) about 50%. more time than the PMR quadtree (with and without bounding boxes), while having only a slightly worse performance than the R$^+$-tree.

These observations lead us to conclude that when the size of the output of the spatial join is of the same order of magnitude as the largest of the two inputs (e.g., larger than 25%), then regardless of whether the output is spatial or not, the PMR quadtrees yield significantly better execution time performances than any of the R-tree variants. However, for spatial joins that result in more modest sized outputs, the R$^+$-tree and the PMR quadtree with bounding boxes prove superior to the other structures. In the context of disk I/Os, however, the two PMR quadtrees outperform the R-tree variants for all output sizes.

A case can still be made, however, for the use of the R*-tree in a spatial join with spatial output as its storage requirements are somewhat smaller than those of the PMR quadtree (20%) for our example data set of over 260,000 line segments. Of course, the R*-tree's construction time is significantly higher than the other structures. This difference is compounded when the structure is used where operations are cascaded so that the output of one spatial operation serves as input to another spatial operation. We also observe that the number of disk I/O operations is always lower for the PMR quadtree than any of the remaining structures at the expense of higher cpu costs for each disk I/O operation due to the added complexity of the operations on each page that is retrieved since each page in a PMR quadtree contains many quadtree nodes while each page in an R-tree and an R$^+$-tree contains just one R-tree or R$^+$-tree node.

References

[1] D. J. Abel. A B$^+$-tree structure for large quadtrees. *Computer Vision, Graphics and Image Processing*, 27(1):19–31, July 1984.

[2] W. G. Aref and H. Samet. Uniquely reporting spatial objects: Yet another operation for comparing spatial data structures. In *Proc. of the 5th Intl. Symp. on Spatial Data Handling*, pages 178–189, Columbia, SC, Aug. 1992.

[3] 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 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[4] T. Brinkhoff and H. P. Kriegel. The impact of global clustering on spatial database systems. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pages 168–179, Santiago, Chile, Sept. 1994.

[5] T. Brinkhoff, H. P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 197–208, Minneapolis, May 1994.

[6] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 237–246, Washington, DC, May 1993.

[7] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[8] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J. B. Yudd. Client-server paradise. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pages 558–569, Santiago, Chile, Sept. 1994.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[10] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proc. of the 1987 ACM SIGMOD Intl. Conf. on Management of Data*, pages 426–439, San Francisco, May 1987.

[11] I. Gargantini. An effective way to represent quadtrees. *Comm. of the ACM*, 25(12):905–910, Dec. 1982.

[12] O. Günther. Efficient computation of spatial joins. In *Proc. of the 9th IEEE Intl. Conf. in Data Engineering*, pages 50–59, Vienna, Apr. 1993.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, Boston, June 1984.

[14] K. Hinrichs and J. Nievergelt. The Grid file: a data structure designed to support proximity queries on spatial objects. In *Proc. of the WG'83 (Intl. Workshop on Graphtheoretic Concepts in Computer Science)*, pages 100–113, Linz, Austria, 1983.

[15] E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In *Advances in Spatial Databases – 2nd Symp., SSD'91*, pages 237–256. Springer–Verlag, Berlin, 1991.

[16] E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 205–214, San Diego, June 1992.

[17] E. G. Hoel and H. Samet. Data-parallel spatial join algorithms. In *Proc. of the 1994 Intl. Conf. on Parallel Processing*, volume 3, pages 227–234, St. Charles, IL, Aug. 1994.

[18] E. G. Hoel and H. Samet. Performance of data-parallel spatial operations. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pages 156–167, Santiago, Chile, Sept. 1994.

[19] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 209–220, Minneapolis, May 1994.

[20] M. Mäntylä. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, MD, 1987.

[21] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, Aug. 1986.

[22] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proc. of the 1987 ACM SIGMOD Intl. Conf. on Management of Data*, pages 270–277, San Francisco, May 1987.

[23] J. Orenstein. An algorithm for computing the overlay of k-dimensional spaces. In *Advances in Spatial Databases – 2nd Symp., SSD'91*, pages 381–400. Springer–Verlag, Berlin, 1991.

[24] D. Rotem. Spatial join indices. In *Proc. of the 7th Intl. Conf. on Data Engineering*, pages 500–509, Kobe, Apr. 1991.

[25] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison–Wesley, Reading, MA, 1990.

[26] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 storage benchmark. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 2–11, Washington, DC, May 1993.

[27] M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, 1981.

[28] U. S. Bureau of the Census, Washington, DC. *TIGER/Line Census Files: 1990 Technical Documentation*, 1991.