The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries [*]

Norio Katayama
Research and Development Department
NACSIS (National Center for Science Information Systems)
katayama@rd.nacsis.ac.jp

Shin'ichi Satoh
Research and Development Department
NACSIS (National Center for Science Information Systems)
satoh@rd.nacsis.ac.jp

Abstract

Recently, similarity queries on feature vectors have been widely used to perform content-based retrieval of images. To apply this technique to large databases, it is required to develop multidimensional index structures supporting nearest neighbor queries efficiently. The SS-tree had been proposed for this purpose and is known to outperform other index structures such as the R*-tree and the K-D-B-tree. One of its most important features is that it employs bounding spheres rather than bounding rectangles for the shape of regions. However, we demonstrate in this paper that bounding spheres occupy much larger volume than bounding rectangles with high-dimensional data and that this reduces search efficiency. To overcome this drawback, we propose a new index structure called the SR-tree (Sphere/Rectangle-tree) which integrates bounding spheres and bounding rectangles. A region of the SR-tree is specified by the intersection of a bounding sphere and a bounding rectangle. Incorporating bounding rectangles permits neighborhoods to be partitioned into smaller regions than the SS-tree and improves the disjointness among regions. This enhances the performance on nearest neighbor queries especially for high-dimensional and non-uniform data which can be practical in actual image/video similarity indexing. We include the performance test results that verify this advantage of the SR-tree and show that the SR-tree outperforms both the SS-tree and the R*-tree.

1 Introduction

Recently, similarity queries on feature vectors have been widely used to perform the content-based retrieval of images [1]. To apply this technique to large databases, it is required to develop multidimensional index structures efficiently supporting nearest neighbor queries. For example, the Informedia project [2], a digital video library project at Carnegie Mellon University, is working to incorporate the content-based retrieval capability into its digital video library and expecting the development of an index structure efficient for similarity queries on ten or more dimensional feature vectors. A feature vector is extracted from image characteristics, e.g., hue, saturation, intensity, texture, etc., and stored in a database along with images. Similarity queries are performed by conducting nearest neighbor queries in the feature vector space. A set of the images similar to a particular image can be retrieved by searching feature vectors close to that of the given image.

The SS-tree [3] had been proposed for this purpose and is known to outperform other index structures such as the R*-tree [4] and the K-D-B-tree [5] in high-dimensional nearest neighbor queries. One of its most important features is that it employs bounding spheres rather than bounding rectangles for the shape of regions. The center of a sphere is the centroid of underlying points and the SS-tree divides points into isotropic neighborhoods by utilizing centroids in the tree construction algorithm. However, we demonstrate in this paper that bounding spheres occupy much larger volume than bounding rectangles with high-dimensional data and that this reduces search efficiency. To overcome this drawback, we propose a new index structure called the SR-tree (Sphere/Rectangle-tree) which integrates bounding spheres and bounding rectangles.

The approach of the SR-tree is a kind of region shape refinement which can be found in the P-tree [6] and in the region approximation method of the spatial join algorithm [7]. The former employs multiple bounding rectangles with different orientations and composes polyhedra regions by their intersection. The resultant polyhedra region is smaller than a single bounding rectangle and achieves better selectivity in the search. The latter exploits the use of convex hulls, ellipses, circles, etc., to achieve more accurate approximation of spatial objects than using minimum bounding rectangles.

The distinctive feature of the SR-tree is that it specifies a region by the intersection of a bounding sphere and a bounding rectangle. The introduction of bounding rectangles permits neighborhoods to be partitioned into smaller regions than the SS-tree and improves the disjointness among regions. This enhances the performance on nearest neighbor queries especially for high-dimensional and non-uniform data which can be practical in actual image/video similarity indexing. We include the performance test results to verify this advantage of the SR-tree and show that the SR-tree outperforms both the SS-tree and the R*-tree.

This paper is organized as follows. Section 2 describes related work. Section 3 compares bounding spheres with bounding rectangles by showing the result of performance tests and discussing their properties. In Section 4, we present the new index structure, SR-tree, and evaluate its performance in Section 5. Section 6 contains conclusions.

[*]This work is based on the research activity while the authors had been visiting at the Robotics Institute, Carnegie Mellon University. Norio Katayama had visited at the Robotics Institute from March 18 to April 3, 1996 by the support of JSPS (Japan Society for the Promotion of Science). Shin'ichi Satoh had been a visiting researcher at the Robotics Institute, Carnegie Mellon University, from April 1995 to April 1997.

This paper has been published in the Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, 13–15 May 1997, Tucson, Arizona.

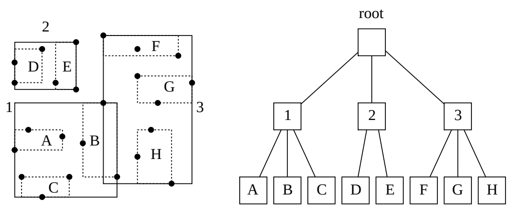


Figure 1: The R*-tree structure

2 Related Work

2.1 The K-D-B-Tree

The K-D-B-tree [5] is an index structure for multidimensional point data. It is a height-balanced tree similar to the B$^+$-tree and its tree structure is constructed by dividing the search space into subregions with coordinate planes recursively. Nodes and leaves correspond to subregions and a disk block is allocated for each of them. The distinctive characteristics of the K-D-B-tree is the disjointness among subregions on the same tree level. This makes the search path of a point query to be a single branch from the root to a leaf. Therefore, the search time of a point query is definitely logarithmic to the size of a data set.

However, the forced splits, i.e., the propagation of splits from a node to its descendants, are required to keep the disjointness among sibling regions. A forced split occurs when a region of an intermediate node is divided crossing its child regions. It can cause the creation of empty or nearly empty leaves and nodes. Therefore, the K-D-B-tree cannot ensure the minimum storage utilization. This reduces the performance of the K-D-B-tree on range queries and nearest neighbor queries.

2.2 The R*-Tree

The R*-tree [4], the most successful variant of the R-tree [8], is a multidimensional index structure for rectangle data. It is a height-balanced tree corresponding to a hierarchy of nested rectangles. Nodes and leaves correspond to rectangles in the hierarchy and a disk block is allocated for each of them. The rectangle of a node is determined by the minimum bounding rectangle of those of its children. The rectangle of a leaf is determined by the minimum bounding rectangle of the data entries contained in that leaf. Therefore, the rectangle of the root node corresponds to the minimum bounding rectangle of the whole data entries, while the rectangle of an intermediate node corresponds to the minimum bounding rectangle of the data entries contained in its lower leaves.

The R*-tree improves the performance of the R-tree by modifying the insertion and the split algorithm and by introducing the forced reinsertion mechanism. Although the R-tree and the R*-tree is originally designed for rectangles, it can be used solely for points (Figure 1) and known to be effective also as a point access method [4]. The R-tree and the R*-tree are different from the K-D-B-tree in the following respects: (1) the regions associated with the nodes and the leaves are determined by bounding rectangles rather than disjoint subregions and (2) the regions of the R-tree and the R*-tree are allowed to overlap each other. Because sibling regions can overlap each other, the search time of a point query depends to the amount of overlap and is not determined by the height of the tree. On the other hand, the R-tree and the R*-tree can ensure the minimum storage utilization, because they require no forced split.

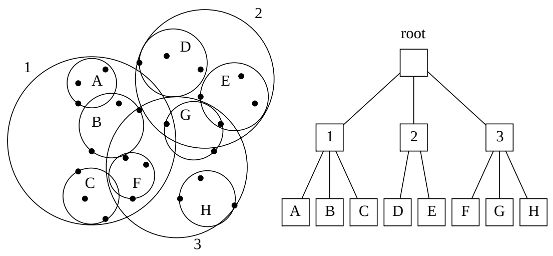


Figure 2: The SS-tree structure

2.3 The SS-Tree

The SS-tree [3] is an index structure designed for similarity indexing of multidimensional point data. It is an improvement of the R*-tree and enhances the performance of nearest neighbor queries by modifying the following respects.

Firstly, it employs bounding spheres rather than bounding rectangles for the region shape (Figure 2). The center of a sphere is the centroid of underlying points and the SS-tree permits to divide points into isotropic neighborhoods by utilizing centroids in the tree construction algorithms, i.e., the insertion algorithm and the split algorithm. On the insertion of a point, the insertion algorithm determines the most suitable subtree to accommodate the new entry by choosing a subtree whose centroid is the nearest to the new entry. When a node or a leaf is full, the split algorithm calculates its coordinate variance on each dimension from the centroids of its children and chooses the dimension with the highest variance for splitting it. These algorithms divide points into isotropic neighborhoods and enhance the performance on nearest neighbor queries. Another advantage of using bounding spheres for the region shape is that it only requires nearly half storage compared to bounding rectangles. Since a sphere is determined by the center and the radius, it can be represented with as many parameters as the dimensionality plus one. On the other hand, the number of parameters required for a rectangle is the double of the dimensionality, because a rectangle is determined by the lower and the upper bound of every dimension. This advantage permits almost doubling the fanout of nodes and reduces the height of trees.

Secondly, the SS-tree modifies the forced reinsertion mechanism of the R*-tree. When a node or a leaf is full, the R*-tree reinserts a portion of its entries rather than splits it, unless reinsertion has been made on the same tree level. On the other hand, the SS-tree reinserts entries unless reinsertion has been made at the same node or leaf. This promotes the dynamic reorganization of the tree structure.

2.4 The VAMSplit R-Tree

The VAMSplit R-tree [9] is an optimized R-tree, i.e., it is constructed in the top-down manner with a given data set. The tree construction algorithm of the VAMSplit R-tree is based on that of the k-d tree [10], a main memory index structure for multidimensional points. The VAMSplit R-tree constructs a tree structure by partitioning points recursively with a coordinate plane which is orthogonal to the dimension with the highest variance. This split algorithm has been used by the optimized k-d trees [11]. The VAMSplit R-tree applies this algorithm to the R-tree and refines the way of selecting a split point to guarantee the minimum number of disk blocks to be used. According to the result reported in [9], the VAMSplit R-tree outperforms both the R*-tree and the SS-tree.

2.5 The TV-Tree

The TV-tree [12] improves the performance of the R*-tree for high-dimensional feature vectors by employing the reduction of dimensionality and the shift (telescoping) of active dimensions. Dimensionality is reduced by ordering dimensions based on their importance and by activating only a few of more important dimensions for indexing. The shift of active dimensions occurs when feature vectors in a subtree have the same coordinate on the most important active dimension. Then, that dimension is made inactive and the less important dimension is newly activated for indexing. This approach is effective for such feature vectors that satisfies the following conditions: (1) dimensions can be ordered by their significance and (2) there exist such feature vectors that allow the shift of active dimensions. As mentioned in [3], the second condition does not always hold for real-valued feature vectors because their coordinates usually have wide diversity. If the second condition does not hold, the effectiveness of the TV-tree results in only the reduction of dimensions which can be commonly applied to other index structures. Thus, the effectiveness of the TV-tree is dependent to applications.

2.6 The X-Tree

The X-tree [13] is a variant of the R*-tree and improves the performance of the R*-tree by employing the overlap-free split and the supernode mechanism. The overlap-free split enables the search space to be divided into disjoint regions like the K-D-B-tree and improves the performance of point queries. A supernode is an oversized node which is arranged to circumvent the overlap among nodes and enhances the I/O throughput for reading and writing nodes. These approaches are not incompatible with the SR-tree. The effectiveness of these methods for the SR-tree is an open question.

3 Bounding Rectangles vs. Bounding Spheres

3.1 Performance Test

We evaluated the performance of the multidimensional index structures, the K-D-B-tree, the R*-tree, the SS-tree, and the VAMSplit R-tree, to clarify their advantages and disadvantages.

The following two data sets were used for the performance test:

(1) uniform data set

(2) real data set

Each data set consists of 16 dimensional points. The uniform data set is a synthetic data set which consists of the points distributed uniformly in the range [0, 1) on each dimension. The real data set consists of the real feature vectors of images which are 16-element histograms computed over a quantized version of the color space.

We constructed indices for these data sets and measured the CPU time and the number of disk reads on nearest neighbor queries. We employed the nearest neighbor search algorithm presented in [14]. A query is to find the nearest 21 points relative to a particular point in the data set. The result was evaluated as the average of 1,000 random trials.

Table 1: The maximum number of entries in a node and in a leaf

Index	Node	Leaf
K-D-B-tree	30	10
R*-tree	31	10
VAMSplit R-tree	31	10
SS-tree	56	12
SR-tree	20	12

Table 2: Tree heights (uniform data set)

Index	Data set size (× 1000)									
	10	20	30	40	50	60	70	80	90	100
K-D-B-tree	4	4	4	4	4	4	5	5	5	5
R*-tree	4	4	4	4	4	4	4	4	4	4
VAMSplit R-tree	4	4	4	4	5	5	5	5	5	5
SS-tree	3	3	4	4	4	4	4	4	4	4
SR-tree	4	4	4	4	5	5	5	5	5	5

Table 3: Tree heights (real data set)

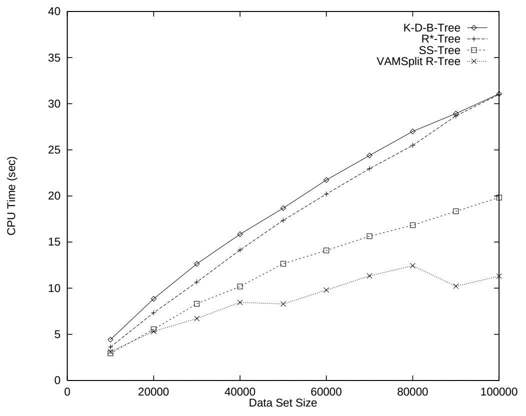
Index	Data set size (× 1000)									
	2	4	6	8	10	12	14	16	18	20
K-D-B-tree	3	4	4	4	4	4	4	4	4	4
R*-tree	3	3	4	4	4	4	4	4	4	4
VAMSplit R-tree	3	4	4	4	4	4	4	4	4	4
SS-tree	3	3	3	3	3	3	3	3	3	3
SR-tree	3	4	4	4	4	4	4	4	4	4

All tests are computed on a Sun Microsystems workstation, SPARCstation-20 (CPU: HyperSPARC 125 MHz, main memory: 224 Mbytes, OS: Solaris 2.4). All programs are implemented in C++[*]. The size of nodes and leaves is set to 8192 bytes to meet with the disk block size of the operating system. The size of the data area associated to each leaf entry is 512 bytes. The maximum number of entries in a node and in a leaf are shown in Table 1. Following the suggestion of the R*-tree [4] and the SS-tree [3], the minimum utilization parameter of each block is set to 40% for all of the index structures and the reinsert fraction parameter of the R*-tree and the SS-tree is set to 30%. The heights of the constructed trees are shown in Table 2 and 3. For K-D-B-trees, we employed the split algorithm of the R^+-tree [15], which is an extension of the K-D-B-tree to spatial objects, instead of the algorithm presented in [10], because the cyclic choice of splitting dimensions presented in [10] is likely to cause forced splits as reported in [16].

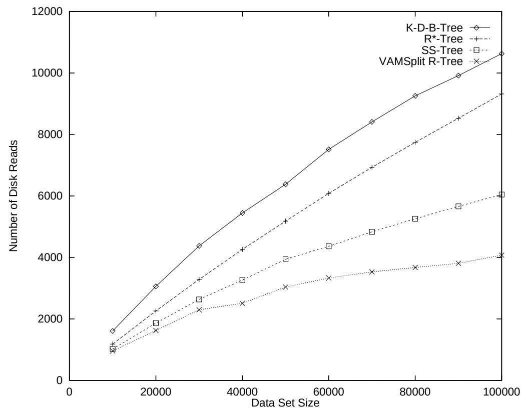
The results for the uniform and the real data set are shown in Figure 3 and 4 respectively. In these figures, the graph (a) shows the CPU time and the graph (b) shows the number of disk reads. The horizontal axis indicates the size of the data set. The size varies from 10,000 to 100,000 for the uniform data set and from 2,000 to 20,000 for the real data set.

These results show that the VAMSplit R-tree outperforms the other index structures. However, the comparison between the VAMSplit R-tree and the other index structures is not necessarily fair, because the VAMSplit R-tree is an optimized index structure taking advantage of full knowledge of the data set while the others are designed to be fully dynamic [9]. Among the dynamic index structures, the SS-tree exhibits the best performance and performs much better than the R*-tree and the K-D-B-tree. This supports the results reported in [3] and the superiority of the SS-tree to the R*-tree is confirmed.

[*]The C++ library of the SR-tree is available with the C and the C++ language interface. Please, contact the authors to try the SR-tree.

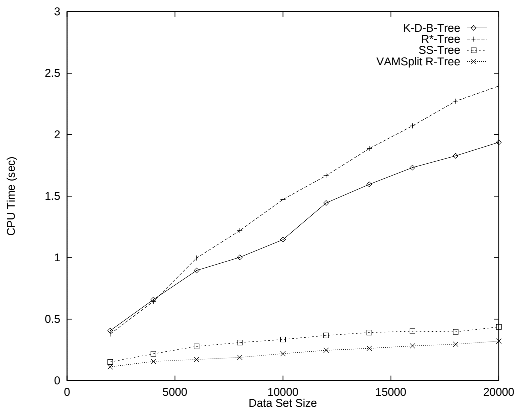


(a) CPU time

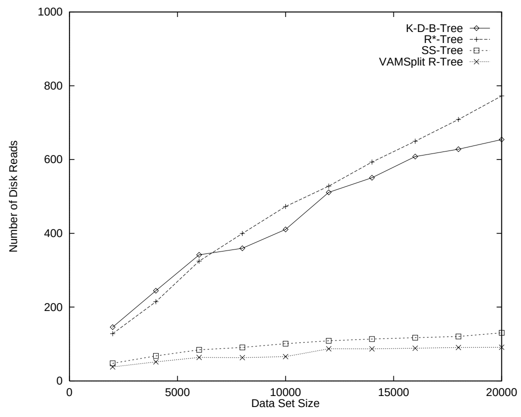


(b) Number of disk reads

Figure 3: Performance of K-D-B-trees, R*-trees, SS-trees, and VAMSplit R-trees (uniform data set)



(a) CPU time



(b) Number of disk reads

Figure 4: Performance of K-D-B-trees, R*-trees, SS-trees, and VAMSplit R-trees (real data set)

3.2 Properties of Bounding Rectangles

The significant feature of the test results in Section 3.1 is the superiority of the SS-tree to the R*-tree and the K-D-B-tree. The SS-tree performs much better than both the R*-tree and the K-D-B-tree especially for the real data set. The SS-tree is about four times faster than the R*-tree.

This superiority can be explained by the following properties of the SS-tree and the R*-tree:

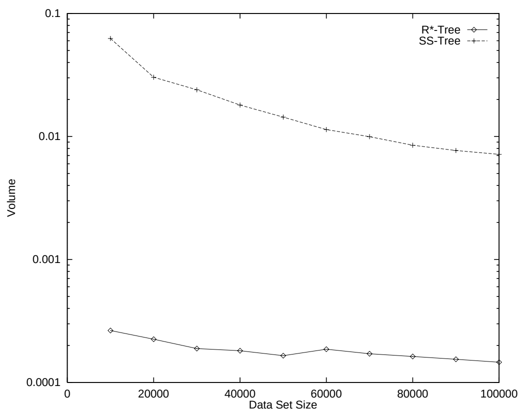
- The SS-tree divides points into isotropic neighborhoods by utilizing bounding spheres.
- The R*-tree divides points into small regions by utilizing bounding rectangles.

To verify these properties, we measured the volumes and the diameters of the leaf-level regions of the SS-trees and the R*-trees constructed for the uniform data set in Section 3.1. Here, the diameter of a region means the diameter of a bounding sphere for the SS-tree and the diagonal of a bounding rectangle for the R*-tree. The results are shown in Figure 5. Figure 5-(a) and 5-(b) graph the average volume and the average diameter respectively.

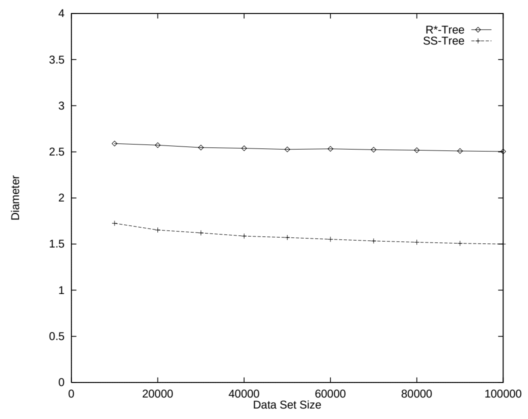
These results show that the average volume of bounding rectangles is much smaller than that of bounding spheres. The former is about 2% of the latter. By contrast, the average diameter of bounding rectangles is much longer than that of bounding spheres. The former is about 2.5, while the latter is about 1.5. Thus, the SS-tree divides points into short-diameter regions, while the R*-tree divides points into small-volume regions. This is why the SS-tree outperforms the R*-tree. Since the diameter of regions has more influence on the performance of nearest neighbor queries than their volumes, the SS-tree, whose average diameter is smaller than that of the R*-tree, exhibits better performance on nearest neighbor queries.

It may seem strange that a region with a smaller volume has a much longer diameter. However, it is possible for a rectangle in high-dimensional space, because the difference between its edge length and its diagonal length grows as the number of dimensions increases. For example, the diagonal length of a D-dimensional unit cube is $\sqrt{D}$ though its edge length is just one, e.g., $\sqrt{2}$ for a 2-dimensional unit square and 4 for a 16-dimensional unit hypercube. Therefore, a bounding rectangle does not necessarily have a short diameter even if its volume is small.

With the above measurement and the consideration, we can conclude that the reason of the superiority of the SS-tree is the shortness of region diameters and that a bounding rectangle of the R*-tree suffers from the difference between



(a) Volume



(b) Diameter

Figure 5: The average volume and the average diameter of the leaf-level regions of the SS-trees and the R*-trees constructed for the uniform data set

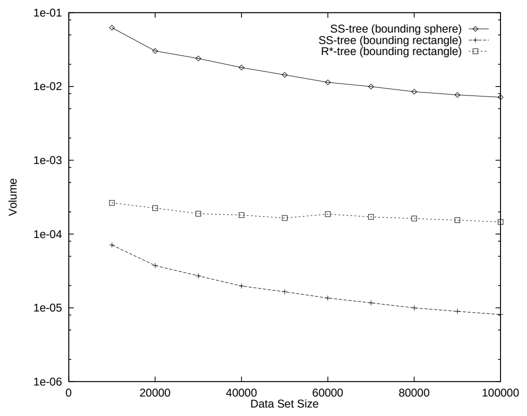


Figure 6: The average volume and the average diameter of the leaf-level regions of the SS-trees constructed for the uniform data set

its edge length and its diagonal length in high-dimensional space.

3.3 Properties of Bounding Spheres

The SS-tree outperforms the R*-tree by employing a bounding sphere whose center is the centroid of underlying points. However, as shown in Figure 5-(a), the bounding spheres of the SS-tree occupy much larger volume than the bounding rectangles of the R*-tree. Regions with larger volume tend to produce more overlap among themselves. This reduces the search efficiency of range queries and nearest neighbor queries. Thus, bounding spheres are not necessarily superior to bounding rectangles in every respect. They are disadvantageous in terms of volume.

To clarify this property, we measured the average volume of the leaf-level regions of SS-trees when they are determined by bounding rectangles instead of bounding spheres. The result of the SS-trees constructed for the uniform data set in Section 3.1 is shown in Figure 6. The horizontal axis indicates the size of the data set and the vertical axis indicates the average volume of the bounding spheres and the bounding rectangles. The average volumes of the leaf-level regions of the R*-trees are also plotted for comparison. These results show that the average volume of the bounding rectangles of the SS-tree leaves is much smaller than that of the bounding spheres. When the data set size is 100,000, the average volume of the bounding rectangles of the SS-tree leaves is about 1/900 of that of the bounding spheres and about 1/18 of the bounding rectangles of the R*-tree leaves. This means that the average volume of the leaf-level regions of the SS-trees will be about 1/900 if the regions are determined by bounding rectangles instead of bounding spheres.

3.4 Discussions

According to the performance test and the measurement above, the properties of bounding rectangles and bounding spheres are summarized as follows:

- Bounding rectangles permit to divide points into small-volume regions. However, they have much longer diameters than bounding spheres, because of the different behavior of their edge length and their diagonal length especially in high-dimensional space.
- Bounding spheres permit to divide points into short-diameter regions. However, they tend to have larger volumes than bounding rectangles.

Thus, bounding rectangles and bounding spheres have both merits and demerits. Bounding rectangles are advantageous in terms of volume. On the other hand, bounding spheres are advantageous in terms of diameter. For nearest neighbor queries, bounding spheres are more advantageous than bounding rectangles, because the lengths of region diameters have more influence to the performance on nearest neighbor queries than the volumes of regions. However, the most desirable property is to divide points into regions both with small volumes and with short diameters.

Based on these consideration, we come to think of the combined use of a bounding rectangle and a bounding sphere. Because their properties are complementary to each other, their intersection seems to permit dividing points into regions with small volumes and short diameters. To realize this idea, we developed the SR-tree (Sphere/Rectangle-tree) presented in the next section. The effectiveness of this combination will be disclosed in the rest of this paper.

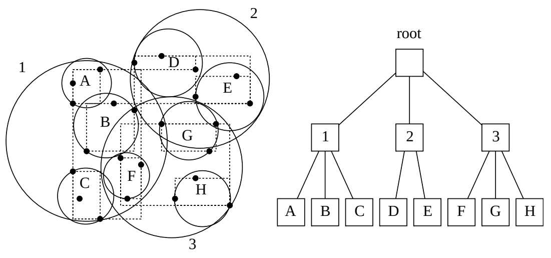


Figure 7: The SR-tree structure

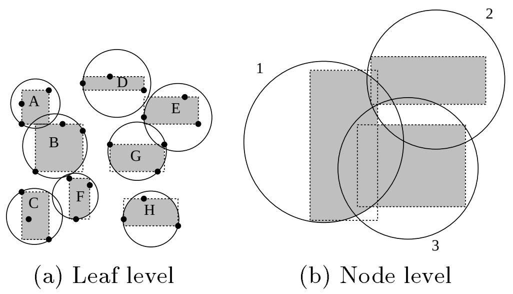


Figure 8: Regions specified by the intersection of a bounding sphere and a bounding rectangle

4 The SR-Tree

4.1 Index Structure

The structure of the SR-tree is based on that of the R-tree [8], in common with the R*-tree [4] and the SS-tree [3], and corresponds to the nested hierarchy of regions as shown in Figure 7. However, the distinctive feature of the SR-tree is that it specifies a region by the intersection of the bounding sphere and the bounding rectangle of underlying points as shown in Figure 8.

A leaf of the SR-tree has the following structure:

$$\begin{array}{ll} L : (E_1, \ldots, E_n) & (m_L \le n \le M_L) \\ E_i : (\boldsymbol{p},\ \mathit{data}). & \end{array}$$

A leaf L consists of entries $E_1, \ldots, E_n$ $(m_L \le n \le M_L)$ where m_L and M_L are the minimum and the maximum number of entries in a leaf. Each entry contains a point $\boldsymbol{p}$ and its attribute *data*. This structure is the same with that of the SS-tree.

A node of the SR-tree has the following structure:

$$\begin{array}{ll} N : (C_1, \ldots, C_n) & (m_N \le n \le M_N) \\ C_i : (\boldsymbol{S}, \boldsymbol{R}, w, \mathit{child\_pointer}). & \end{array}$$

A node N consists of entries $C_1, \ldots, C_n$ $(m_N \le n \le M_N)$ where m_N and M_N are the minimum and the maximum number of entries in a node. Each entry corresponds to a child of the node and consists of the following four components: a bounding sphere $\boldsymbol{S}$, a bounding rectangle $\boldsymbol{R}$, the number of points w, and a pointer to the child *child_pointer*. The way to compute $\boldsymbol{S}$ and $\boldsymbol{R}$ is explained in the next section. The variable w is the total number of points contained in the subtree whose top is the child pointed by *child_pointer*. The difference of this structure to that of the SS-tree is the introduction of the bounding rectangle $\boldsymbol{R}$. On the other hand, the difference of this structure to that of the R*-tree is the introduction of the bounding sphere $\boldsymbol{S}$ and the number of points w.

4.2 Insertion

The insertion algorithm of the SR-tree is based on that of the SS-tree. We applied the centroid-based algorithm of the SS-tree to the SR-tree, because its effectiveness for nearest neighbor queries is confirmed through our performance test as shown in Section 3.1. Since the algorithm of the SS-tree can be understood by referring to the papers of the SS-tree [3] and its predecessors, i.e., the R-tree [8] and the R*-tree [4], we only mention its outline and the difference between the algorithm of the SS-tree and that of the SR-tree. The insertion algorithm of the SS-tree determines the most suitable subtree to accommodate the new entry by choosing a subtree whose centroid is the nearest to the new entry. When a node or a leaf is full, the SS-tree reinserts a portion of its entries rather than splits it unless reinsertion has been made at the same node or leaf. Otherwise, the split algorithm calculates its coordinate variance on each dimension from the centroids of its children and chooses the dimension with the highest variance for splitting it.

The insertion algorithm of the SR-tree differs from that of the SS-tree in the way of updating regions on the insertion of a new entry. The SR-tree needs to update both bounding spheres and bounding rectangles, while the SS-tree only needs to update bounding spheres. The way of updating bounding rectangles is the same with that of the R-tree and the R*-tree. However, the way of updating bounding spheres is different from that of the SS-tree. Because a region of the SR-tree is the intersection of a bounding rectangle and a bounding sphere, the SR-tree determines the bounding sphere of a parent node by utilizing both the bounding spheres and the bounding rectangles of its children as follows:

(1) The center of a bounding sphere, $\boldsymbol{x}$ $(x_1, \ldots, x_i, \ldots, x_D)$, is computed as follows:

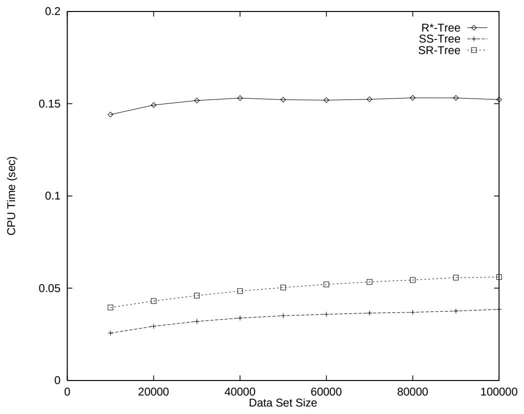
$$x_i = \frac{\sum_{k=1}^{n} C_k.x_i \times C_k.w}{\sum_{k=1}^{n} C_k.w} \qquad (1 \le i \le D),$$

where k $(1 \le k \le n)$ is an index to the children $C_1, \ldots, C_n$, i $(1 \le i \le D)$ is an index to the dimensions, $C_k.x_i$ denotes the i-th coordinate of the center of the child C_k, and $C_k.w$ denotes the number of points contained in the subtree whose top is the child C_k. This definition is the same with that of the SS-tree.

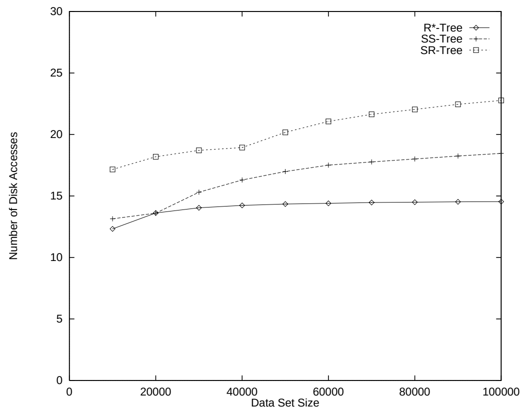
(2) The radius of a bounding sphere, r, is computed as follows:

$$\begin{aligned} r &= \min\ (d_s, d_r), \\ d_s &= \max_{1 \le k \le n} (\|\boldsymbol{x} - C_k.\boldsymbol{x}\| + C_k.r), \\ d_r &= \max_{1 \le k \le n} (MAXDIST(\boldsymbol{x}, C_k.\boldsymbol{R})), \end{aligned}$$

where k $(1 \le k \le n)$ is an index to the children $C_1, \ldots, C_n$, i $(1 \le i \le D)$ is an index to dimensions, $C_k.\boldsymbol{x}$ and $C_k.r$ denote the center and the radius of the bounding sphere of the child C_k, and $C_k.\boldsymbol{R}$ denotes the bounding rectangle of the child C_k. The function $MAXDIST(\boldsymbol{p}, \boldsymbol{R})$ computes the maximum distance from a point $\boldsymbol{p}$ to a rectangle $\boldsymbol{R}$ and is defined as follows:



(a) CPU time



(b) Number of disk accesses

Figure 9: Insertion cost of R*-trees, SS-trees, and SR-trees (uniform data set)

$$MAXDIST(\boldsymbol{p}, \boldsymbol{R}) \equiv \max_{\boldsymbol{q} \in \boldsymbol{R}} (\|\boldsymbol{p} - \boldsymbol{q}\|).$$

This can be computed easily by pursuing such a vertex of the rectangle $\boldsymbol{R}$ that is the farthest from the point $\boldsymbol{p}$.

In the above definition, d_s denotes the maximum distance from the center of a parent node to the bounding spheres of its children, and d_r denotes the maximum distance from the center of a parent node to the bounding rectangles of its children. Although the SS-tree determines the radius r by d_s, the SR-tree determines the radius r by choosing the smaller one between d_s and d_r. This permits the radius of the SR-tree to be smaller than that of the SS-tree and reduces the overlap of bounding spheres.

4.3 Deletion

In common with the R*-tree and the SS-tree, the deletion algorithm of the SR-tree is the same with that of the R-tree [8]. When the deletion of an entry causes no under-utilization of any leaf or node, the entry is simply removed from the tree. Otherwise, the under-utilized leaf or node is removed from the tree and orphaned entries are reinserted to the tree.

4.4 Nearest Neighbor Search

In common with the R*-tree and the SS-tree, the nearest neighbor search of the SR-tree is performed by applying the algorithm presented in [14]. This algorithm finds a number of points nearest to some given point. It is a depth-first search and consists of two stages. Firstly, it collects the given number of points to make a candidate set. Secondly, it revises the candidate set with visiting every leaf whose region overlaps the range of the candidate set. It terminates when there are no more leaves to visit and the final candidate set is the search result. The tree is traversed in order of the distance from the search point to each region. At every visited node, the distance from the search point to the region of each child is computed and the closer child is visited prior to the farther ones.

Although the traversal algorithm is common to the R*-tree, the SS-tree, and the SR-tree, the SR-tree differs from both the R*-tree and the SS-tree in the way of computing the distance from a search point to each region. Because a region of the SR-tree is the intersection of a bounding sphere and a bounding rectangle, the minimum distance from a search point to a region is defined as the longer one between the minimum distance to its bounding sphere and the minimum distance to its bounding rectangle. Therefore, the minimum distance from a search point $\boldsymbol{p}$ to the region of a child C_k, which is denoted by d, is computed as follows:

$$\begin{aligned} d &= \max(d_s, d_r), \\ d_s &= \max(0,\ \|\boldsymbol{p} - C_k.\boldsymbol{x}\| - C_k.r), \\ d_r &= MINDIST(\boldsymbol{p}, C_k.\boldsymbol{R}), \end{aligned}$$

where $C_k.\boldsymbol{x}$ and $C_k.r$ denote the center and the radius of the bounding sphere of the child C_k, and $C_k.\boldsymbol{R}$ denotes the bounding rectangle of C_k. The function $MINDIST(\boldsymbol{p}, \boldsymbol{R})$ computes the minimum distance from a point $\boldsymbol{p}$ to a rectangle $\boldsymbol{R}$ and is defined as follows:

$$MINDIST(\boldsymbol{p}, \boldsymbol{R}) \equiv \min_{\boldsymbol{q} \in \boldsymbol{R}} (\|\boldsymbol{p} - \boldsymbol{q}\|).$$

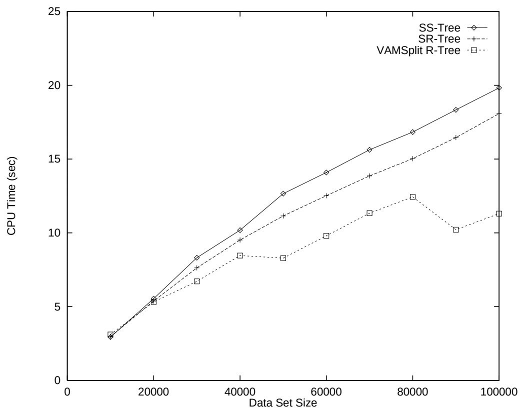
The algorithm of computing this function is presented in [14].

In the above definition, d_s denotes the minimum distance from the search point $\boldsymbol{p}$ to the bounding sphere of the child C_k, and d_r denotes the minimum distance to the bounding rectangle of the child C_k. While the R*-tree determines the minimum distance d by d_r and the SS-tree by d_s, the SR-tree determines the minimum distance d by choosing the longer one between d_s and d_r. This provides the better estimation of the distance from the search point $\boldsymbol{p}$ to the nearest point in a region and enhances the performance of nearest neighbor searching.

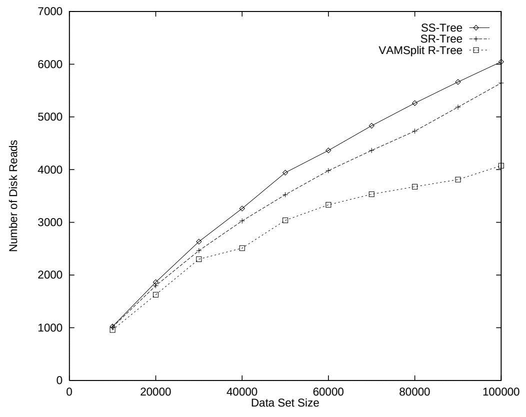
5 Evaluation of the SR-tree

5.1 Performance Test

We measured the performance of the SR-tree under the same conditions with the test in Section 3.1. The maximum number of entries in a node and in a leaf are shown in Table 1 and the heights of trees are shown in Table 2 and 3. Figure 9 shows the average cost of inserting a new entry into SR-trees, SS-trees, and R*-trees for the uniform data set. Figure 9-(a)

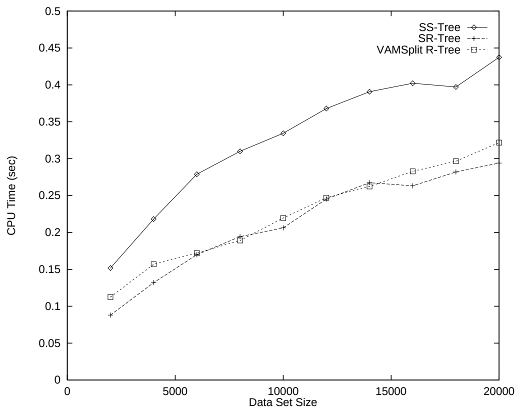


(a) CPU time

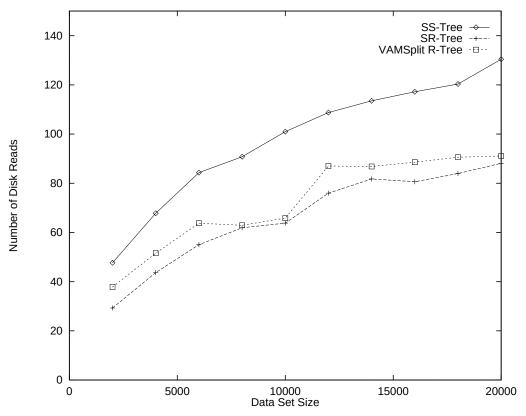


(b) Number of disk reads

Figure 10: Performance of SR-trees (uniform data set)



(a) CPU time



(b) Number of disk reads

Figure 11: Performance of SR-trees (real data set)

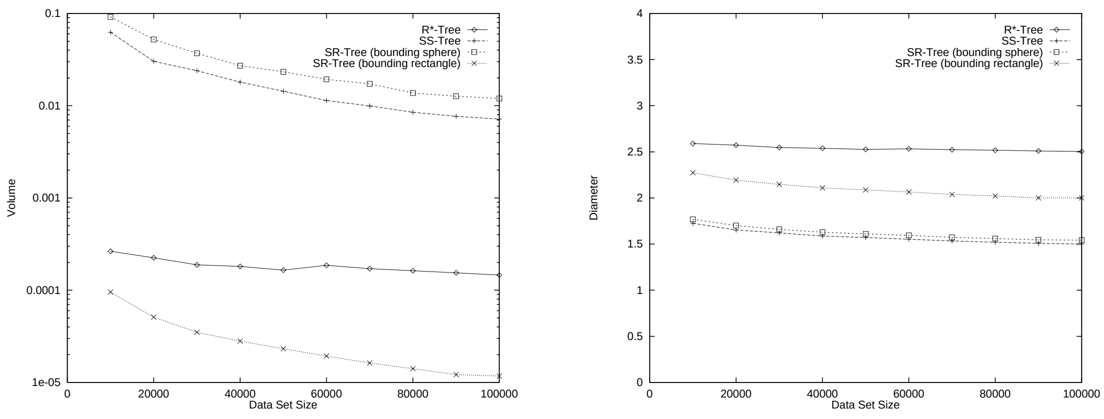
and 9-(b) show, respectively, the CPU time and the number of disk accesses, i.e., the total number of disk reads and disk writes. The SR-tree and the SS-tree require less CPU time than the R*-tree, because the centroid-based insertion algorithm of the SS-tree requires significantly less CPU time than the algorithm of the R*-tree [3]. The SR-tree requires more CPU time and more disk accesses than the SS-tree, because the SR-tree contains not only bounding spheres but also bounding rectangles.

Although the SR-tree requires more creation cost than the SS-tree, it enhances the query performance remarkably. Figure 10 and 11 show the results of the SR-tree for the uniform and the real data set respectively. The results of the SS-tree and the VAMSplit R-tree, which are already shown in Section 3.1, are also plotted for comparison. They clearly depict that the SR-tree outperforms the SS-tree for both data sets. For the uniform and the real data set, the SR-tree reduces the CPU time to 91% and 67% of the SS-tree and the number of disk reads to 93% and 68% of the SS-tree respectively. In comparison with the VAMSplit R-tree, the VAMSplit R-tree outperforms the SR-tree for the uniform data set. However, the SR-tree slightly outperforms the VAMSplit R-tree for the real data set. It is remarkable that the SR-tree exhibits the comparable performance to the VAMSplit R-tree, considering that the SR-tree is a dynamic index structure while the VAMSplit R-tree is a static, i.e., optimized, index structure.

5.2 The Advantage of the SR-tree

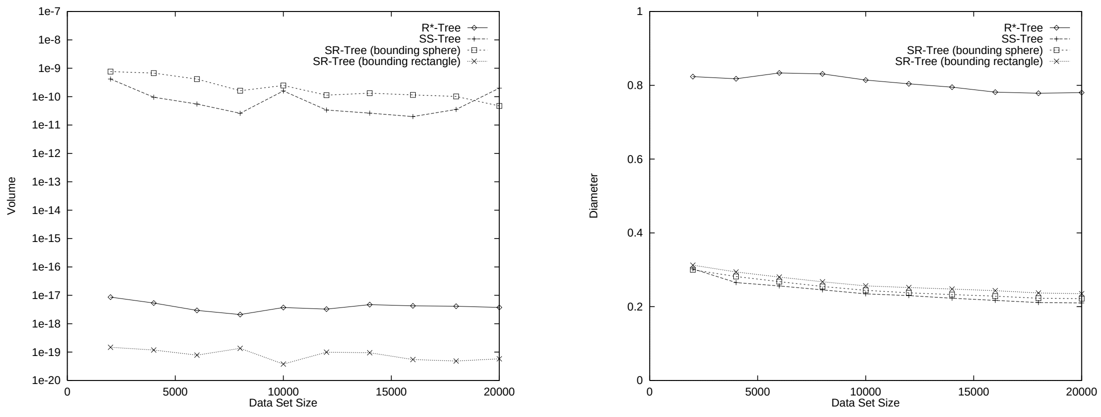
The SR-tree outperforms the SS-tree by dividing points into regions with both small volumes and short diameters. To clarify this advantage of the SR-tree, we measured the volumes and the diameters of the leaf-level regions of the SR-trees constructed for the performance test in Section 5.1. The results for the uniform and the real data set are shown in Figure 12 and 13. Figure 12-(a) and 13-(a) graph the average volume of the leaf-level regions, while Figure 12-(b) and 13-(b) graph the average diameter of the leaf-level regions. The results of the R*-tree and the SS-tree are also plotted for comparison.

For the SR-trees, the precise volume and the precise diameter are not measured, because it is quite difficult to compute them for the intersection of a sphere and a rectangle. Instead, we measured the volumes and the diameters of their bounding spheres and bounding rectangles. These measurements indicate the upper limit of the real volume and the real diameter, because a region, which is the intersection of its bounding sphere and its bounding rectangle, has a smaller volume and a shorter diameter than its bounding sphere and its bounding rectangle. Therefore, the real av-



(a) Volume (b) Diameter

Figure 12: The average volume and the average diameter of the leaf-level regions of R*-trees, SS-trees, and SR-trees (uniform data set)



(a) Volume (b) Diameter

Figure 13: The average volume and the average diameter of the leaf-level regions of R*-trees, SS-trees, and SR-trees (real data set)

erage volume of the leaf-level regions of the SR-tree is not larger than the average volume of their bounding rectangles which is marked by × in Figure 12-(a) and 13-(a), and the real average diameter of the leaf-level regions of the SR-tree is not longer than the average diameter of their bounding spheres which is marked by □ in Figure 12-(b) and 13-(b).

These results exhibit the following characteristics:

(1) Figure 12-(a) and 13-(a) show that the leaf-level regions of the SR-tree have smaller volumes than those of the R*-tree and the SS-tree on average. They are about 1/1000 of those of the SS-tree for the uniform data set and about $1/10^9$ of those of the SS-tree for the real data set.

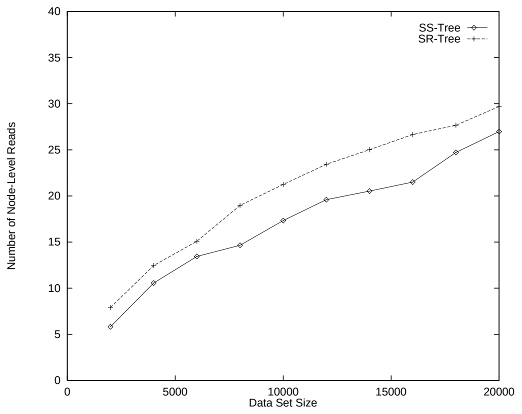
(2) Figure 12-(b) and 13-(b) show that the leaf-level regions of the SR-tree have as short diameters as those of the SS-tree.

These characteristics verify that the SR-tree divides points into regions with both smaller volumes and shorter diameters. Moreover, their volumes are even smaller than those of the R*-tree. This improves the disjointness among regions and enhances the performance on nearest neighbor queries as shown in Section 5.1.

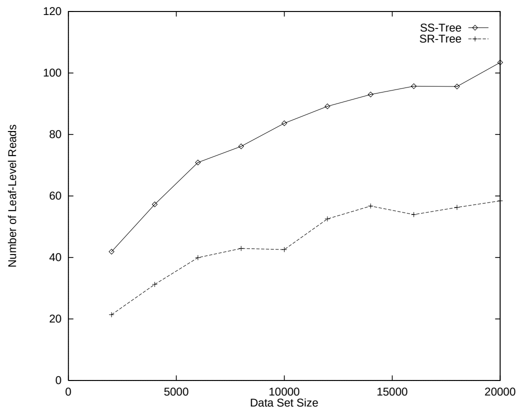
5.3 The Fanout Problem

It had been pointed out that the fanout, i.e., the maximum number of branches in an intermediate node, decreases in the higher dimensionality, because the size of a node entry grows as dimensionality increases [3]. Since a node entry of the SR-tree contains both a bounding sphere and a bounding rectangle, its size is three times larger than that of the SS-tree and one-and-a-half of that of the R*-tree. Therefore, the fanout of the SR-tree is one third of the SS-tree and two thirds of the R*-tree. The reduction of the fanout may require more nodes to be read on queries and possibly cause the reduction of the query performance.

To analyze this problem, we measured the number of node-level reads and leaf-level reads separately. Figure 14 shows the results for the real data set. The SR-tree incurs more node-level reads than the SS-tree. However, the total number of disk reads of the SR-tree is smaller than that of the SS-tree, because the SR-tree saves leaf-level reads more than the increase of node-level reads. Thus, the SR-tree reduces the number of disk reads, even though it suffers from the fanout problem.



(a) Node level



(b) Leaf level

Figure 14: The number of node-level reads and leaf-level reads (real data set)

5.4 Evaluation on Dimensionality and Distribution

To investigate the characteristics of the SR-tree, we measured the performance of the SR-tree with varying the dimensionality and the distribution of data.

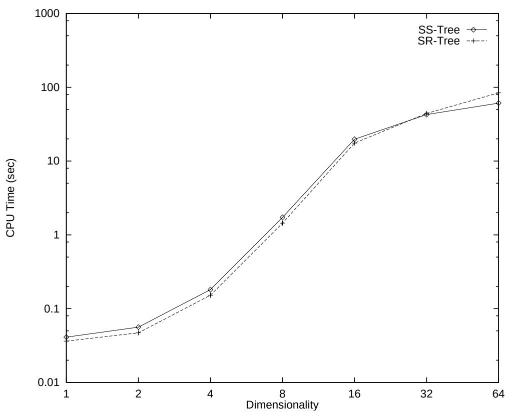
First, we measured the performance of the SR-tree for the uniform data set with varying its dimensionality from 1 to 64. The data set size is fixed to 100,000 for every dimensionality. Figure 15-(a) and 15-(b) show the CPU time and the number of disk reads respectively. Figure 16 shows the proportion of accessed leaves, and Figure 17 shows the minimum, the average, and the maximum of the distances between the points within the uniform data set. For all figures, the horizontal axis indicates dimensionality. These results show that the uniform data set is unsuitable for evaluating index structures on nearest neighbor queries in high dimensionality, because the uniform data set is too hard for the touchstone. As shown in Figure 16, the proportion of accessed leaves reaches 100% in 32 and 64 dimensions for both the SR-tree and the SS-tree. This means that these indices completely failed to divide points into neighborhoods and are forced to access all leaves. This failure derives from the distribution of the distances between the points within the uniform data set. As shown in Figure 17, the minimum of the distances grows drastically as dimensionality increases and the ratio of the minimum to the maximum increases up to 24% in 16 dimensions, 40% in 32 dimensions, and 53% in 64 dimensions. This means that the variation in the distances reduces as dimensionality increases and that each point contained in the uniform data set has similar distances to the others in high dimensionality. This property of the uniform data set makes it essentially difficult to divide points into neighborhoods. Thus, the uniform data set is unsuitable for evaluating the performance of index structures on nearest neighbor queries in high dimensionality.

Therefore, we devised another data set, i.e., the cluster data set. We designed this data set to be more practical in high dimensionality than the uniform data set. This data set consists of multiple clusters and each cluster contains a fixed number of points within a sphere. Therefore, the total number of points is the number of clusters multiplied by the number of cluster elements. The location and the radius of each cluster is chosen randomly within the unit cube and the location of each point is chosen by generating a point on the sphere surface uniformly and then shifting it along radius randomly. We measured the performance of the SR-tree for the cluster data set in which the number of clusters is 100 and the number of cluster elements is 1000. The results are shown in Figure 18. Figure 18-(a) and 18-(b) show the CPU time and the number of disk reads respectively. These results show that the SR-tree is effective from the lower dimensionality to the higher dimensionality and improves the performance about 100% compared to the SS-tree.

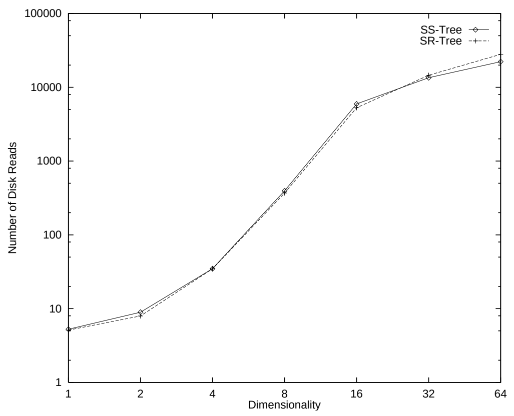
Secondly, we measured the performance of the SR-tree with varying the distribution of data. To produce various distributions, we varied the number of clusters from 1 to 100,000 for the cluster data set with fixing the dimensionality to 16. The total number of points is fixed to 100,000. Therefore, the number of points in a cluster is 100,000 divided by the number of clusters. For the cluster data set, varying the number of clusters corresponds to varying its uniformity. When the number of clusters is 1, points are distributed in a single sphere. When the number of clusters is greater than 1 and less than 100,000, points are distributed in multiple spheres located within the unit cube. When the number of clusters is 100,000, points are uniformly distributed within the unit cube. The results are shown in Figure 19. Figure 19-(a) shows the CPU time and Figure 19-(b) shows the number of disk reads. The horizontal axis indicates the number of clusters. These results show that the SR-tree improves the performance more when the data set is less uniform. For example, the SR-tree improves the SS-tree by 42% when the number of clusters is 1, 88% when the number of clusters is 100, and 36% when the number of clusters is 100,000. This property is consistent with the result of Section 5.1 where the SR-tree exhibits more performance improvement for the real data set than for the uniform data set. These results imply that the SR-tree is more effective for less uniform data sets.

6 Conclusions

In this paper, a new multidimensional index structure called the SR-tree is proposed for high-dimensional nearest neighbor queries. The distinctive feature of the SR-tree is the combined utilization of bounding spheres and bounding rectangles. The performance test of the R*-tree and the SS-tree revealed that bounding spheres permit to divide points into regions with short diameters and that bounding rectangles permit to divide points into regions with small volumes. Although the SS-tree outperforms the R*-tree by taking ad-



(a) CPU time



(b) Number of disk reads

Figure 15: Performance of SR-trees and SS-trees with varying dimensionality (uniform data set)

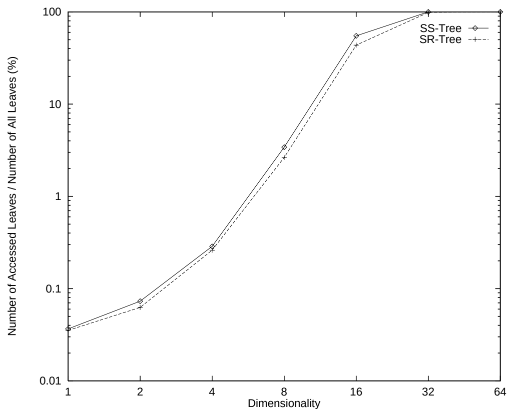


Figure 16: The ratio of the accessed leaves to the whole leaves

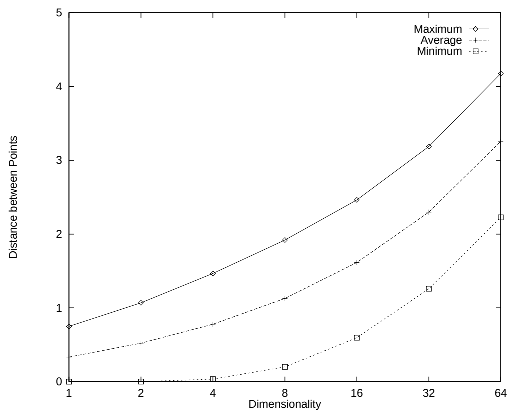


Figure 17: The maximum, the average, and the minimum of the distances between the points contained in the uniform data set

vantage of bounding spheres, we demonstrated that bounding spheres occupy larger volumes than bounding rectangles on average and that this may reduce the disjointness among regions. The SR-tree permits to divide points into regions with both small volumes and short diameters by specifying a region with the intersection of its bounding sphere and its bounding rectangle. This improves the disjointness among regions and enhances the performance on nearest neighbor queries. The performance test verifies that the SR-tree divides points into regions with both small volumes and small diameters and shows that the SR-tree outperforms the SS-tree and the R*-tree. The performance evaluation tests show that the SR-tree is especially effective for high-dimensional and non-uniform data sets which can be practical in actual image/video similarity indexing. Although the creation cost of the SR-tree is higher than that of the SS-tree, the performance enhancement by the SR-tree would be advantageous to the applications which require such an index structure that are efficient for high-dimensional nearest neighbor queries.

Acknowledgments

The authors would like to express their gratitude to the staff of the Informedia project, a digital video library project at Carnegie Mellon University, for providing them with the data set of real feature vectors. The authors also would like to express their gratitude to Prof Takeo Kanade, Director of the Robotics Institute, Carnegie Mellon University, and JSPS (Japan Society for the Promotion of Science) for giving us the chance to start this research.

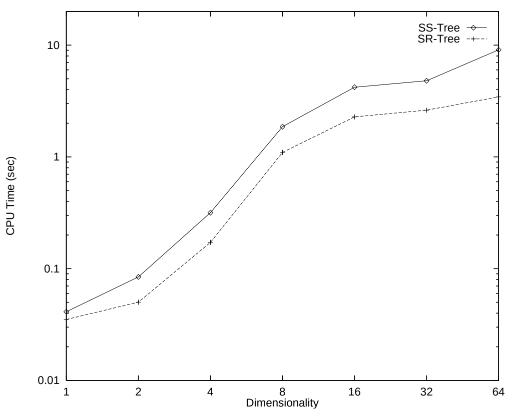
References

[1] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker, “Query by Image and Video Content: the QBIC System,” IEEE Computer, Vol.28, No.9, pp.23–32, Sep. 1995.

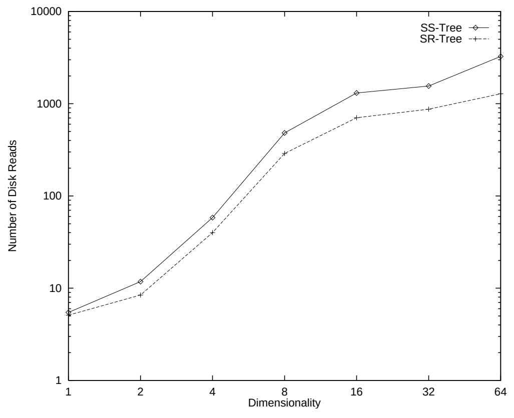
[2] H. D. Wactlar, T. Kanade, M. A. Smith, and S. M. Stevens, “Intelligent Access to Digital Video: Informedia Project,” IEEE Computer, Vol.29, No.5, pp.46–52, May 1996.

[3] D. A. White and R. Jain, “Similarity Indexing with the SS-tree,” Proc. of the 12th Int. Conf. on Data Engineering, New Orleans, USA, pp.516–523, Feb. 1996.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The R*-tree: an Efficient and Robust Access

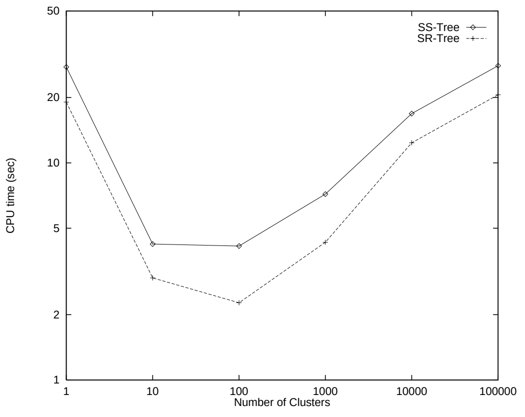


(a) CPU time

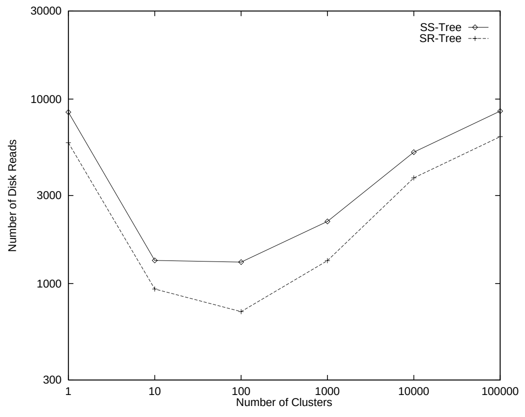


(b) Number of disk reads

Figure 18: Performance of SR-trees and SS-trees with varying dimensionality(cluster data set)



(a) CPU time



(b) Number of disk reads

Figure 19: Performance of SR-trees and SS-trees with varying the distribution(cluster data set)

Method for Points and Rectangles," Proc. ACM SIGMOD, Atlantic City, USA, pp.322–331, May 1990.

[5] J. T. Robinson, "The K-D-B-tree: a Search Structure for Large Multidimensional Dynamic Indexes," Proc. ACM SIGMOD, Ann Arbor, USA, pp.10–18, Apr. 1981.

[6] H. V. Jagadish, "Spatial Search with Polyhedra," Proc. of the 6th Int. Conf. on Data Engineering, Los Angeles, USA, pp.311–319, Feb. 1990.

[7] T. Brinkhoff, H.-P. Kriegel, R. Schneider, B. Seeger, "Multi-Step Processing of Spatial Joins," Proc. ACM SIGMOD, Minneapolis, USA, pp.197–208, May 1994.

[8] A. Guttman, "R-trees: a Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD, Boston, USA, pp.47–57, Jun. 1984.

[9] D. A. White and R. Jain, "Similarity Indexing: Algorithms and Performance," Proc. SPIE Vol.2670, San Diego, USA, pp.62–73, Jan. 1996.

[10] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," Comm. of the ACM, Vol.18, No.9, pp.509–517, Sep. 1975.

[11] R. Sproull, "Refinements to Nearest-Neighbor Searching in k-Dimensional Trees," Algorithmica, Vol.6, No.4, pp.579–589, 1991.

[12] K.-I. Lin, H. V. Jagadish, and C. Faloutsos, "The TV-tree: An Index Structure for High-Dimensional Data," VLDB Journal, Vol. 3, No. 4, pp.517–542, 1994.

[13] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," Proc. of the 22nd VLDB Conf., Bombay, India, pp.28–39, Sep. 1996.

[14] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," Proc. ACM SIGMOD, San Jose, USA, pp.71–79, May 1995.

[15] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-tree: a Dynamic Index for Multi-Dimensional Objects," Proc. of the 13th VLDB Conf., Brighton, England, pp.507–518, Sep. 1987.

[16] D. Greene, "An Implementation and Performance Analysis of Spatial Data Access Methods," Proc. of the 5th Int. Conf. on Data Engineering, Los Angeles, USA, pp.606–615, Feb. 1989.