The PN-Tree: A Parallel and Distributed Multidimensional Index

M. H. ALI mhali@cs.purdue.edu
Computer Science Dept., Purdue University, West Lafayette, IN 47907, USA

A. A. SAAD
M. A. ISMAIL
Department of Computer Science and Automatic Control, Faculty of Engineering, Alexandria University, Alexandria, Egypt

Abstract. Multidimensional indexing is concerned with the indexing of multi-attributed records, where queries can be applied on some or all of the attributes. Indexing multi-attributed records is referred to by the term *multidimensional indexing* because each record is viewed as a point in a multidimensional space with a number of dimensions that is equal to the number of attributes. The values of the point coordinates along each dimension are equivalent to the values of the corresponding attributes. In this paper, the *PN-tree,* a new index structure for multidimensional spaces, is presented. This index structure is an efficient structure for indexing multidimensional points and is parallel by nature. Moreover, the proposed index structure does not lose its efficiency if it is serially processed or if it is processed using a small number of processors. The *PN-tree* can take advantage of as many processors as the dimensionality of the space. The *PN-tree* makes use of B^+-trees that have been developed and tested over years in many DBMSs. The *PN-tree* is compared to the *Hybrid tree* that is known for its superiority among various index structures. Experimental results show that parallel processing of the *PN-tree* reduces significantly the number of disk accesses involved in the search operation. Even in its serial case, the *PN-tree* outperforms the *Hybrid tree* for large database sizes.

Keywords: multidimensional indexing, multi-attributed records, parallel and distributed processing

1. Introduction

The field of multidimensional indexing has been an open research direction for many years. It has been motivated by the emergence of new applications. Such applications require an index structure that is capable of handling multi-attributed records. These index structures should handle the record’s multiple attributes equally without favoring any attributes over the others, unless explicitly required. Existing records can be queried by all or some of the attributes. For example, image databases may store hundreds of thousands or millions of images. A suitable index structure for image databases is one that is capable of indexing the images by their features. The extraction of features relies either on an expert system to extract the features of these images or on manual extraction. Once the image features are extracted, it is the responsibility of the indexing mechanism to build a set of indexes over these features to speed up the processing of the images.

Indexing multi-attributed records is referred to by the term *multidimensional indexing* because each record is considered a point in a multidimensional space with a number of dimensions that is equal to the number of attributes. The value of the point coordinates along each dimension is proportional to the value of the corresponding attributes. The main objective is to minimize the number of disk accesses required to process various types of queries. In the ideal case, only disk pages that belong to the query result should be retrieved. Therefore, if possible, nearby points in the multidimensional space should be stored on the same disk page so that fewer disk accesses are needed to retrieve the query result.

Because of the need for a fast response time, an intuitive approach is to process the indexes in parallel. However, existing index structures are not easy parallelizable. A straightforward parallelization may not improve significantly the performance of existing structures. Moreover, performance improvement saturates as the number of parallel processors increases, which means that parallelism can take advantage of only a small number of parallel processors. In this paper, the *Projected Node* tree or *PN-tree* is presented. *PN-tree* is parallel in nature but can also be executed efficiently using a single processor. This paper is concerned with multidimensional points or zero sized objects. Non-zero sized objects are beyond the scope of this work. The proposed index structure is intended to be suitable for dynamic databases where insertions and deletions are allowed along with the search operation. Various types of queries such as exact queries, range queries and nearest-neighbor queries are applicable to multidimensional spaces. In this paper, we focus on the performance of the range query.

The B-tree and its variations have been widely used to index one-dimensional spaces. Almost every DBMS includes an implementation of the B^+-tree along with its optimization techniques. Reusing well established technologies gives an added value to newly proposed structures. This paper makes use of the B^+-tree to index multidimensional spaces. The *PN-tree* is built using multiple B^+-trees. Hence, the realization of the *PN-tree* in any DBMS is relatively straightforward.

The remainder of the paper is organized as follows. Section 2 gives a brief description of related work. Section 3 introduces the proposed structure along with its insertion, deletion and splitting algorithms. The proposed structure is subjected to various experiments and the results are both analyzed and presented in Section 4. Comparative studies are presented in Section 5. Section 6 concludes the paper.

2. Related work

In this section, the basic concepts of multidimensional indexing are presented. A classification of existing multidimensional index structures along with a brief description of various structures is given in Section 2.1. Previous work on parallel multidimensional index structures is presented in Section 2.2. Since we compare the proposed structure with the *Hybrid tree*, more detail about the *Hybrid tree* is given in Section 2.3.

2.1. Classification of multidimensional index structures

Multidimensional index structures can be classified into: (1) structures designed for storing multidimensional points only. These methods are called *Point Access Methods* (PAMs), and (2) structures designed for spatial or non-zero objects. These methods are called *Spatial Access Methods* (SAMs). This classification has been used by many researchers [3, 9, 24, 25]. We describe briefly various multidimensional index structures in the following sections. For a more detailed survey, the reader may refer to [9].

2.1.1 Point access methods (PAMs)

PAMs can be divided further into hierarchical and non-hierarchical structures. The k-d tree [4], which is a main memory structure, is a generalization to the hierarchical binary search tree for multidimensional points. The k-d tree is the basis for several disk based structures such as the local split decision (LSD) tree [11], the k-d-b tree [21], and SP-GIST[1]. The hB-tree [19] and the Buddy tree [24] are introduced to avoid the propagation of downward splits in the k-d-b tree.

Non-hierarchical structures, e.g., the Grid files [20], represent records as multidimensional points. The underlying multidimensional space is formed by the cartesian product of the domain keys. Then, the space is divided into a grid, where each grid cell is stored in a disk page. Many variations and enhancements over grid files exist, e.g., the EXCELL method [27], the two-level Grid file [12], the multilayer Grid file [26], the twin Grid file [13], and the BANG (Balanced And Nested Grid) file [8].

2.1.2 Spatial access methods (SAMs)

Major techniques for handling non-point or non zero-sized objects can be categorized into the following classes:

(1) *Object Mapping/Transformation*: This approach maps the bounding rectangles of objects from a k-dimensional space into points in a 2k-dimensional space. Alternatively, the multidimensional space can be linearized using a space-filling curve and is indexed using any basic one-dimensional data structure for point data. This is the case for the UB-tree [2].

(2) *Object Duplication/Clippin*g: The object clipping method decomposes an object into smaller yet simpler pieces such that each piece is included entirely in one subspace. The object identifier is duplicated in all these subspaces.

(3) *Object Boundin*g: Under this mechanism, the space is partitioned into subspaces such that each object is included entirely in one subspace.

The Quad tree [17, 23] handles point and non-point objects. It handles non-point objects via object clipping or duplication. The R-tree [10] handles point and non-point objects using a rectangular *Minimum Bounding Region* (MBR). These MBRs may overlap. The R*-tree [3] is another variation of the R-tree. The R*-tree modifies the insertion and splitting algorithms of the R-tree using various criteria.

The TV-tree [18] uses the least number of attributes possible to discriminate among objects. More attributes are introduced gradually to discriminate among objects. The SS-tree [29] is a variation of the R*-tree that uses spheres as bounding regions instead of rectangles. Both spheres and rectangles are combined in the SR-tree [16] to define the minimum bounding regions. The X-tree [5] is a variation of the R-tree that postpones node splitting by introducing supernodes that are larger than the usual block size. Node splitting occurs if minimum overlap is guaranteed.

2.2. Parallel R-trees

Kamel and Faloutsos propose a parallel architecture for the R tree [15]. They propose three different approaches to distribute the data among multiple disks to facilitate and speed up parallel data access. The first approach distributes the data points among *d* disks either in a round robin fashion or by partitioning the space into *d* partitions. An independent R-tree is built over each disk. The R-trees operate in parallel to answer the query. The second approach uses one R-tree whose nodes are ‘supernodes’. Each supernode consists of *d* pages that are striped over the *d* disks. The third approach, referred to as the MX-R-tree or the multiplexed R-tree [7], is a single R-tree with each node spanning one page. The root is kept in main memory while other nodes are distributed over different disks. Nodes are assigned to disks based on a *proximity* measure. Nodes that are likely to qualify under the same query are separated into different disks. This heuristic increases parallelism as both nodes can be retrieved at the same time. Experimental results show the superiority of the MX-R-tree over the other approaches.

2.3. The hybrid tree

In the indexing methods discussed in the previous sections, it is noticed that the space is divided into subspaces. With respect to the way in which the space is divided into subspaces, these indexing methods can be divided into *Bounding Region* (BR)-based and *Space Partitioning* (SP)-based index structures. A BR-based index structure consists of *bounding regions* (BRs) that are arranged in a spatial hierarchy. On the other hand, an SP-based index structure consists of a space that is recursively partitioned into mutually disjoint subspaces.

The *Hybrid tree*, proposed by Chakrabarti and Mehrotra in 1999 [6], combines positive aspects of BR-based data structures and SP-based data structures. The *Hybrid tree* chooses a single dimension to split a node. However, single dimension splits, like those in the k-d-b tree, may necessitate costly cascading splits. The *Hybrid tree* avoids the cascading splits problem by relaxing the above constraint, i.e., the indexed subspaces may overlap. This is similar to the BR-based data structures. The overlap is permitted for the sake of not violating any constraints placed over the storage utilization. The space partitioning in a *Hybrid tree* is represented using a k-d tree. The k-d tree is modified to allow for overlapping splits by storing two split positions: the first split position represents the right (or higher) boundary of the left (or lower side) partition (denoted by lsp or left side partition) while the second split position represents the left boundary of the right partition (denoted by rsp or right side partition). Moreover, the dead space in each node is eliminated by storing few bits that represent the coordinates of the live space. These bits help reduce the overhead of empty nodes.

3. The PN-tree

In this section, we present the *PN-tree*, a new disk-based multidimensional index structure along with its insertion, node splitting, searching, and deletion algorithms. The details of how to parallelize and distribute the *PN-tree* over multiple disks are presented at the end of this section.

3.1. The basic idea

The basic idea behind the *PN-tree* can be summarized as follows (refer to Figure 1 for illustration):

- Points are clustered into multidimensional nodes.
- Each node is projected over each dimension.
- Projections over each dimension are indexed separately using one-dimensional index structures.

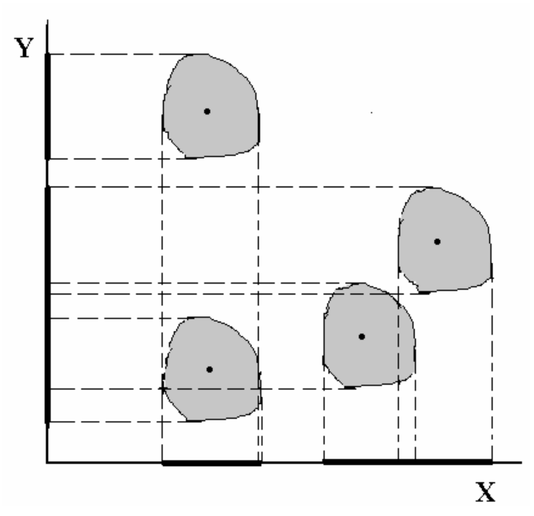

Figure 1. Points are clustered into nodes then these nodes are projected over each dimension.

First, points are clustered multidimensionally into multidimensional nodes so as to cluster nearby points in the multidimensional space into the same node and consequently to store them on the same disk page. Clustering speeds the processing of range queries.

Second, we project the nodes over each of the n-dimensions in an attempt to obtain *n* different views of the data space. Then, the nodes that are formed from the previous step are projected over each dimension. As a result, there will be a set of one-dimensional intervals over each dimension representing the node projections.

Finally, projections over each dimension can be considered as a view of the multidimensional space that is suitable for indexing. Using these projections, the original space can be reconstructed to some extent.

3.2. The index structure

We introduce the following three structures that will cooperate together and work in harmony to yield the *PN-tree*:

(1) **A set of buckets.** Each bucket stores one node that contains a set of points. The size of the bucket is equal to a disk page size. Each bucket is identified by its disk block number.

(2) **A set of one-dimensional index structures.** This set of one-dimensional index structures indexes the projections of nodes over different dimensions. These index structures store the line segments that represent node projections. Building these one-dimensional index structures is detailed in Section 3.7.

(3) **A simple multidimensional index structure.** This multidimensional index structure stores the centroids of the nodes. Node centroids are used by the insertion algorithm to select the closest node to the point to be inserted. The overhead involved in the storage of node centroids is not high. This is because the number of cluster nodes is much less than the number of points.

3.3. The insertion algorithm

The insertion algorithm and its associated node splitting algorithm are responsible for clustering nearby points into the same node. Efficient insertion and splitting algorithms eliminate the need for other clustering techniques. The insertion algorithm can be summarized as follows:

- Choose the node whose centroid is closest to the point to be inserted.
- Insert the point into the chosen node.
- If an overflow occurs then split the chosen node.

Once a new point arrives, the algorithm picks the node with the closest centroid to accommodate the incoming point. The multidimensional index structure that stores the node centroids is probed in this step. A nearest-neighbor search is performed over this structure to pick the nearest centroid. The nearest-neighbor search is based on the branch-and-bound technique [22]. The insertion of incoming points into the nodes with the nearest centroids clusters nearby points into the same node.

The pseudo code of the insertion algorithm is given below. We use four functions to handle the multidimensional index structure responsible for storing node centroids. These four functions will be used while describing other algorithms throughout the paper.

InsertCentroid (NodeId N, Point C). This function stores the centroid *C* along with its corresponding '*NodeId'* in the multidimensional structure.

UpdateCentroid (NodeId N, Point OldC, Point NewC). This function updates the centroid of node *N* from '*OldC'* to '*NewC'*.

DeleteCentroid (NodeId N, Point C). This function deletes the centroid *C* of node *N* from the multidimensional structure.

NodeId GetNearestNeighbor(Point P). This function returns the node whose centroid is closest to *P*.

We index node projections over each dimension using one-dimensional index structures. We use the following functions to handle interval projections.

InsertInterval (integer D, NodeId N, real Start, real End). This function inserts the projection of node *N* over dimension *D.* The projection is specified by the one-dimensional interval with *'Start'* and *'End'* as its start and end points, respectively.

DeleteInterval (integer D, NodeId N, real Start, real End). This function deletes the interval of node *N* whose boundaries are '*Start'* and '*End' from* the one-dimensional index structure built over dimension number '*D'*.

UpdateInterval (integer D, NodeId N, real OldStart, real OldEnd, real NewStart, real NewEnd). This function updates the projections of node *N* from '*OldStart'* and '*OldEnd'* to '*NewStart'* and *'NewEnd'*, respectively, in the one-dimensional index structure built over dimension number '*D'*.

List GetPartialResult(integer D, real Start, real End). This function returns a list of all nodes whose projection over dimension number *'D'* intersects the search range projection over the same dimension.

Algorithm 1: Insertion

```
Given a point P, it is required to insert P in the structure.
Step1: N=GetNearestNeighbor(P), [Get the node whose centroid is closest to P]
       if the structure is empty, create a new node N
Step2: insert P into N
Step3: if an overflow occurs then
            invoke Split to split N into N1 and N2
            DeleteCentroid (N, N.Centroid).
            InsertCentroid (N1, Centroid(N1)).
            InsertCentroid (N2, Centroid(N2)).
            stop
Step4: UpdateCentroid (N, N.Centroid, Centroid(N)) [Updates centroid of node N
                                                    after insertion]
Step5: for i=1to NumberOfDimensions
 begin
```

```
            if $P_i$<N.$\text{min}_i$ then UpdateInterval(i, N, N.$\text{min}_i$, N.$\text{max}_i$, $P_i$, N.$\text{max}_i$)
            if $P_i$>N.$\text{max}_i$ then UpdateInterval(i, N, N.$\text{min}_i$, N.$\text{max}_i$, N.$\text{min}_i$, $P_i$)
      end
End
```

Notes:

N.min_i denotes the lower bound of node N along the ith dimension.

N.max_i denotes the upper bound of node N along the ith dimension.

N.Centroid denotes the old centroid of node N before it is updated.

Centroid(N) this function computes the new centroid of node N.

3.4. The node-splitting algorithm

The splitting algorithms of multidimensional index structures have evolved over time. Once a node overflows, it is split into two new nodes. Various criteria have been used to split overflowing nodes. For example, the R-tree [10] minimizes the area of bounding rectangles. The R*-tree [3] takes into account the minimization of the area of each rectangle, the overlap among rectangles, the margin of each rectangle and the storage utilization constraints. Both the SS-tree [29] and the SR-tree [16] try to minimize the variance among points in each node.

The node-splitting algorithm used by the *Hybrid tree* [6], which is also used in the *PN-tree*, can be summarized as follows:

- Choose the split dimension (the dimension with the largest extent).
- Choose the split position:
    - Select the split position at the middle of the node.
    - If storage utilization is violated, shift the split position in the proper direction to satisfy utilization requirements.

Notice that the following two decisions have to be made when splitting a node: First, which dimension to split along. Second, where to split along the chosen dimension. It was proved by Chakrabarti and Mehrotra that choosing the dimension with the largest extent as the split dimension minimizes the increase in the *Expected Disk Access* (EDA). Also, the split position should be chosen at the middle of the node. The proof is stated in [6]. Storage utilization requirements may be violated if we split at the middle of the node. In this case, we try to make the split as close to the middle of the node as possible provided that utilization constraints are satisfied. In few situations, the utilization constraints may not be satisfied, e.g., if a group of points stand in a straight line that coincides with the split position. Adding this group to either partition causes the other partition to be underutilized. These situations are rare in practice and cause no problem if we allow few nodes to remain underutilized. Minimization of the *EDA* keeps nearby points clustered after node splitting. This means that the node-splitting algorithm works side by side with the insertion algorithm to preserve the clustering of points.

3.5. The search algorithm

In this section, the search algorithm is presented for range queries. The range queries are in the form of a query point and a distance *d*. It is required to find all points within distance *d* from the query point. The search algorithm is summarized as follows (refer to Figure 2 for illustration):

- Project the search range over each dimension.
- For each dimension *i*, include the nodes whose projections intersect the search range projection in a partial result set R_i.
- Intersect the partial result sets R_1, R_2, …, R_k to yield the result set R, where *k* is the dimensionality.
- Retrieve member nodes of R from disk.

As described above, the search range is projected over all dimensions. A node belongs to the search range if its projections over all dimensions intersect the range query projections over the same dimensions. The search algorithm is described in detail in Algorithm 2.

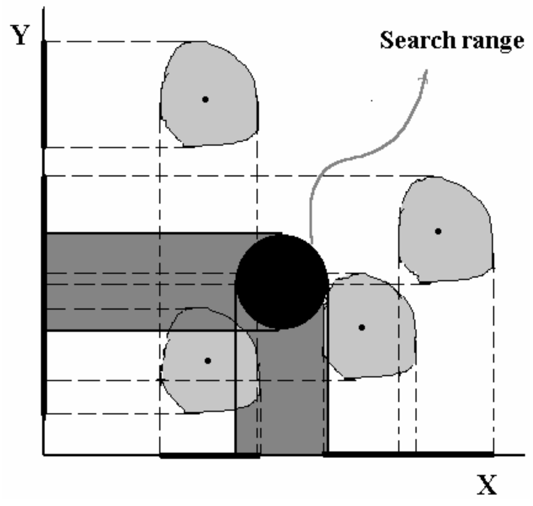

Figure 2. The search operation.

One problem exists and is illustrated in Figure 3. This problem is referred to as the *false alarm* problem. The projections of a node over all dimensions may intersect the projections of the query range. However, the node itself does not intersect the query range and the corresponding disk page records do not belong to the query result. A large number of false alarms degrade the performance. Fortunately, these false alarms are not frequent under usual conditions. False alarms are considered a key factor in analyzing the performance of the structure as we describe in the performance evaluation section.

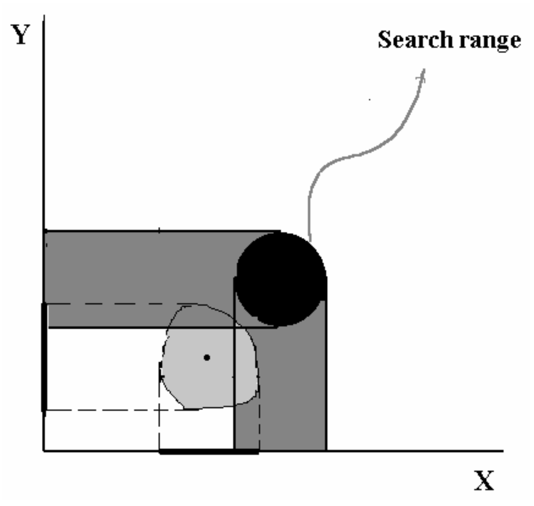

Figure 3. The false alarm problem.

Algorithm 2: RangeQuerySearch

Given a query point QP and a search range radius R. It is required to retrieve all points that belong to the specified search range

```
    Step1: [get partial results from each dimension]
           for i=1 to NoOfDimensions
                  $List_i$= GetPartialResult(i,$QP_i$-R, $QP_i$+R)
    Step2: [Intersect partial result sets]
           d=NoOfDimensions
           while d>1 do
                  begin
                    for i=1 to (d div 2)
                         $List_i$= Intersect($List_i$,$List_{i+(d+1) div 2}$)
                    d= (d+1) div 2
                  end
    Step3: retrieve nodes in $List_1$ from disk
End
```

3.6. The deletion algorithm

The deletion algorithm is straightforward and is summarized in the following steps:

- Search for the point to be deleted
- If not found then error
  else
    - Remove the point from the enclosing node
    - Adjust the node centroid
    - Adjust node projections if necessary
    - If the node underflows, then remove the whole node and reinsert its orphaned points

Notice that a node underflows if it contains less than a specified percentage of its full capacity. Experimental results of [3, 29] show that setting this percentage to 40% gives the best results. This means that a minimum utilization of 40% is guaranteed. However, a node containing 40% of its maximum capacity is rare. Node utilization is usually much higher using real data sets. In the *PN-tree*, the typical storage utilization is around 75%. The algorithm is described in detail in Algorithms 3 and 4.

```
Algorithm 3: Deletion
        Given a point P, it is required to delete this point from the structure
        Step1: N=Search(P)   [get the node N containing P]
        Step2: remove P from N
        Step3: if N.NoOfPoints<m  then Invoke RemoveNode(N) (Algorithm 4) and stop
        Step4: UpdateCentroid(N, N.Centroid, Centroid(N))  [update the centroid of node N]
        Step5: [Update the projections of N over all dimensions]
         for i=1 to NoOfDimensions
                if $P_i$=N.$min_i$ then
                  begin
                          Calculate LB, the new lower bound of N along the $i^{th}$ dimension
                          UpdateInterval(i, N, N.$min_i$, N.$max_i$, LB, N.$max_i$)
                  end
                else $P_i$=N.$max_i$ then
                  begin
                          Calculate UB, the new upper bound of N along the $i^{th}$ dimension
                          UpdateInterval(i, N, N.$min_i$, N.$max_i$, N.$min_i$, UB)
                  end
End

Algorithm 4: RemoveNode
        Given an underflowing node N, it is required to remove N from the structure. The remaining
points are reinserted
        Step1: DeleteCentroid (N, N.Centroid)  [delete the centroid of node N]
        Step2: [remove the projections of node N over all dimensions]
        Step3: for i=1 to NoOfDimensions
                DeleteInterval(i, N, N.$min_i$,N.$max_i$)
        Step4: [reinsert orphaned points]
         for i=1 to N.NoOfPoint
                Insert(N[i])
End
```

Modifying the coordinates of a point, say *P*, is performed by deleting *P* then reinserting it after being modified. This algorithm allows the structure to reorganize and redistribute the modified points into nodes.

3.7. Building one-dimensional indexes

The *PN-tree* uses one-dimensional indexes to store the projections of each node over the various dimensions. Any one-dimensional index structure that is already developed and tested can be used to accomplish this task. Simply, the required one-dimensional index structure should support the indexing of rectilinear line segments, i.e. segments that are parallel to the x-axis. Various techniques have been proposed in the literature to index line segments [14, 23]. In this section, the B^+-tree is proposed and is incorporated in our structure. The B^+-tree is chosen because it is a well developed technology that has come to a level of robustness and maturity over the years. The choice of the B^+-tree makes it relatively easy to incorporate the *PN-tree* into commercial DBMSs.

The projections of data nodes over various dimensions are stored so that they can be used in the processing of queries. The nodes whose projections intersect the search range projections over all dimensions should be retrieved. From Figure 4, for two line segments not to intersect, the following condition must hold: $E_1<S_2$ or $E_2<S_1$, where S_1 and S_2 are the start points of the two line segments and $E_1$and E_2 are the end points of the two line segments, respectively. This condition can be stated as follows: "for two line segments not to intersect, the first line segment must end before the second line segment begins or the second line segment must end before the first line segment begins."

We propose to build two B^+-trees over each dimension: One B^+-tree for the start values of the projection intervals and the other B^+-tree for the end values. The B^+-tree consists of two parts: the index part and the sequence set. The index part is used to speed up the search process while the sequence set is used for sequential processing of the stored data.

The B^+-tree that stores the start values of the intervals is searched for the first value that is greater than the end value of the search range projection. Then, the sequence set is traced to exclude all the intervals that follow. Analogous steps are performed on the other B^+-tree that stores the end values of the intervals. The first value that is less than the start value of the search range projection is reached. It is noted here that the B^+-tree that stores the start values of the intervals has a sequence set that is sorted in an ascending order. On the other hand, the B^+-tree that stores the end values of the intervals has a sequence set that is sorted in a descending order. By following the two B^+-trees, the intervals that begin after the end of the search range projection and those that end before the search range interval starts are removed. The remaining intervals are those that intersect the search range projection.

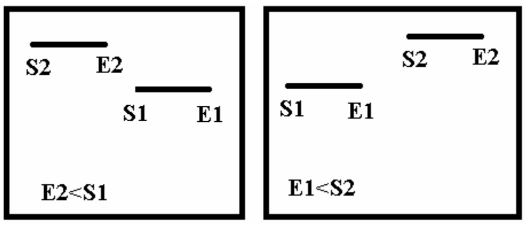

Figure 4. The condition for two line segments not to intersect.

3.8. Parallelizing the proposed structure

One of the major advantages of the proposed *PN-tree* is that it is parallel by nature. It consists of a set of one-dimensional indexes built over the various dimensions. Each dimension is indexed separately using its own one-dimensional structure. Thus, the processing of the *PN-tree* can be distributed among many processors. Each processor deals with one of the dimensions. This means that the *PN-tree* can take advantage of as many processors as the dimensionality of the space. This does not mean that it is an obligation to use as many processors as the dimensionality. For example, if there are n-dimensions and n/2 processors, then each processor will be responsible for indexing two dimensions.

The indexing process can be enhanced and parallelism can be increased if the records that belong to the query result are retrieved in parallel. This can be achieved if the data is declustered over multiple disks. The problem of distributing the data over multiple disks is discussed in the next subsection.

3.9. Declustering the index over multiple disks

The declustering method used here, termed the *proximity index (PI)*, was first suggested by Faloutsos and Kamel in the context of the MX-R-tree [7]. The *Proximity Index* ('PI') assigns the new nodes resulting from the splitting of an overflowing node to the disk with the 'least similar' nodes (i.e., least likely to qualify in the same range query). The formulas to compute the proximity index of a node N_0 to a disk that contains a set of rectangles N_1, ..., N_k is based on the *proximity measur*e: This measure compares two rectangles and assesses the probability that they will be retrieved by the same query. The *proximity index* of a new node N_0 and a disk D (that contains the sibling nodes N_1, ..., N_k) is the proximity of the most 'proximal' node to N_0.

4. Experimental Results

In Section 4.1, various comparison parameters that affect the performance of multidimensional structures are stated. Then, the effect of each parameter is tested and is detailed in the subsequent sections. Data sets are synthetically composed in a multidimensional space such that the point coordinate values over each dimension range from zero to one. The points' coordinate values are approximated to six decimal digits. The setting of each experiment is detailed in its own section.

4.1. Comparison parameters

The basic parameters that are involved in shaping the performance curves are summarized as follows:

1- The page size. The size of the disk page affects the performance of index structures. This effect is detailed in Section 4.2. As the page size increases, more points can be saved in the same disk page. This reduces the number of nodes or equivalently the number of disk pages. As a result, the height of the tree for hierarchical structures is reduced. This decrease in tree height reduces the number of disk accesses. However, in the *PN-tree,* other factors play an important role in increasing or decreasing the disk access.

2- The dimensionality. The number of dimensions in the space is an important factor when evaluating the performance of index structures. Many index structures do not perform well as the dimensionality increases. The performance of our structure under the variation in dimensionalities is described in Section 4.3.

3- The number of processors. The number of processors cooperating to index the data is a factor that affects the performance. This issue is addressed in Section 4.4.

4- The number of inserted points. Finally, the performance of the *PN-tree* is tested under various numbers of inserted points. This performance study is described in Section 4.5.

All tests are run on a Pentium II, 233 MHz, IBM machine with 64 MB RAM. All programs are written in C++ and run under Red Hat Linux Release 6.2.

4.2. The effect of the page size

A simulation experiment over 100,000 uniformly distributed points in a 16-dimensional space is carried out. The size of the disk page is varied and the performance of the *PN-tree* is evaluated under various page sizes. The experiment is conducted for both the serial and the parallel cases of the *PN-tree*. The experiment is carried out for both a 5% and a 10% search ranges, i.e., a search range with a radius that is equal to either 5% or 10% of the whole space. The results are illustrated in Figures 5 and 6 for the serial and parallel cases, respectively.

The effect of the disk page is controlled by two factors. First, as the size of the disk page increases, nodes with larger sizes are obtained. The overlap among such nodes increases specially for higher dimensionalities. Such overlap deteriorates the performance. Second, as the size of the disk page decreases, a large number of nodes are obtained. Such a large number of nodes require one-dimensional indexes with many tree levels. The increase in tree height results in an increase in the number of disk accesses.

The effect of these two factors is depicted in Figures 5 and 6. Figure 5 illustrates the performance of the serial case whereas Figure 6 illustrates the parallel case where maximum parallelism is assumed, i.e., the number of processors is equal to the number of dimensions. Notice that as the size of the disk page decreases, the performance improves. This trend continues till a page size of 1KB is reached. For disk pages smaller than 1KB, a large number of nodes emerges, and hence the one-dimensional index structures increase in height and the performance deteriorates.

A page size of 1KB is considered the best choice for the *PN-tree*. This is why the experiments shown in Figures 5 and 6 are repeated with a focus into the interval between zero and two KB disk pages. The results are illustrated in Figures 7 and 8.

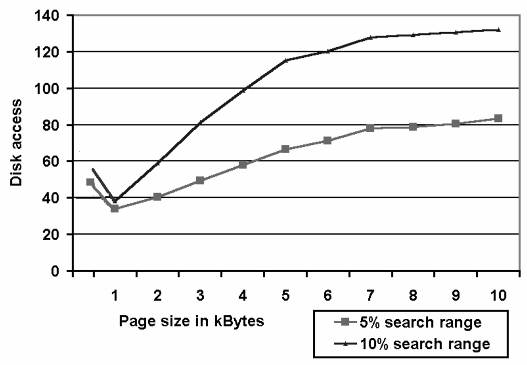

Figure 5. The effect of the page size (the serial case).

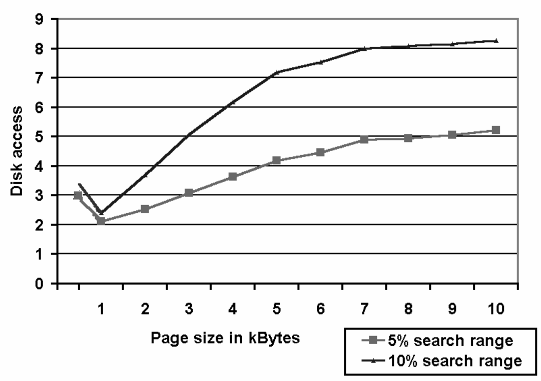

Figure 6. The effect of the page size (the parallel case).

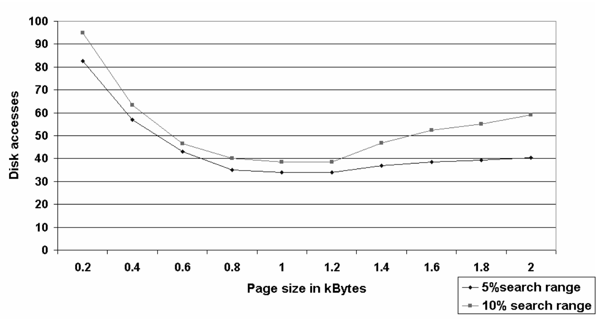

Figure 7. The effect of a page size zoomed around 1KB (the serial case).

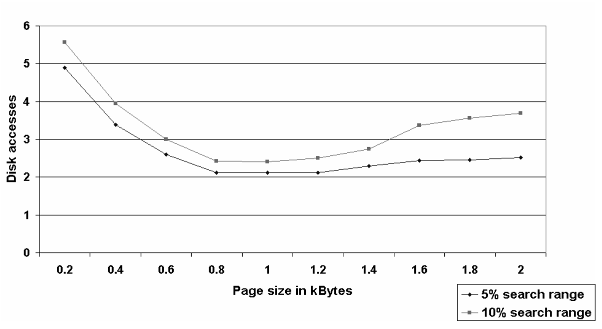

Figure 8. The effect of a page size zoomed around 1KB (the parallel case).

4.3. The effect of dimensionality

The performance of many index structures deteriorates as the dimensionality increases. This phenomenon is known as the *dimensionality curse* [28]. This happens because the overlap among nodes increases in higher dimensionalities. This increased overlap affects adversely the performance of multidimensional structures. For hierarchical structures, many search paths will be active at the same time. This is because, most likely, the search range intersects many overlapping parent nodes.

In the *PN-tree*, the performance behavior with the dimensionality is completely different. Before discussing the performance of both the serial and the parallel cases of the *PN-tree*, it is important to note that the performance is affected by two factors: (1) the disk accesses involved in processing the one-dimensional index structures built over each dimension (2) the disk accesses involved in retrieving from disk the records that result from the intersection of partial result sets over the various dimensions.

The second factor is divided subsequently into the disk accesses involved in retrieving records that belong to the query result, and the disk accesses involved in false alarms. A disk page that is retrieved from disk and that contains no points that belong to the query result is considered a false alarm.

False alarms occur because the projections of a node over all indexed dimensions may intersect the projections of the search range over the same dimensions, while the node itself does not intersect the query range. Refer to Figure 3 for illustration.

The overhead involved in processing the one-dimensional index structures built over each dimension increases as the dimensionality increases. However, the number of disk accesses involved in false alarms decreases with the increase in the dimensionality. This can be explained by considering each new dimension as if it were a new piece of information about the multidimensional space. For each new dimension, a new index is built over this dimension and more information is stored about the space. This increased amount of information results in a reduced number of disk accesses involved in false alarms.

Figure 9 illustrates the performance of our structure in both the serial and the parallel cases for range queries with a radius equal to 5% of the whole space. Figure 10 gives the performance for range queries with a radius equal to 10% of the whole space. The test is performed over 100,000 uniformly distributed points with various dimensionalities. In the parallel case, the number of processors is equal to the dimensionality of the space.

Notice that, for the serial case, the performance improves as the dimensionality increases. This performance improvement saturates as the increase in the dimensionality continues. From Figures 9 and 10, the performance of range queries with 10% radius saturates slower than that of queries with 5% radius. This is because as the search range increases, more false alarms occur.

For the parallel case, the performance is stable with the dimensionality. This is because the increase in the number of disk accesses involved in processing the one-dimensional indexes is circumvented by increasing the number of processors. From the previous analysis, we conclude that the proposed *PN-tree* does not lose its efficiency for high dimensional spaces.

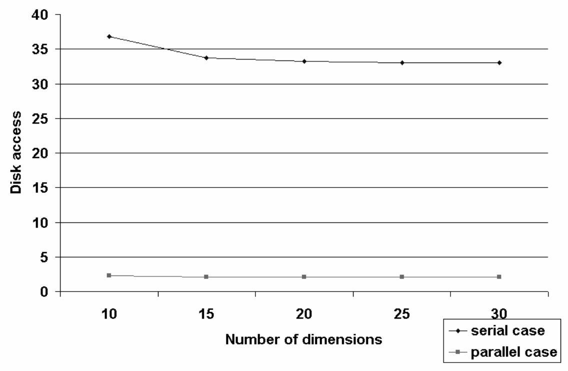

Figure 9. The dimensionality effect for a 5% search range.

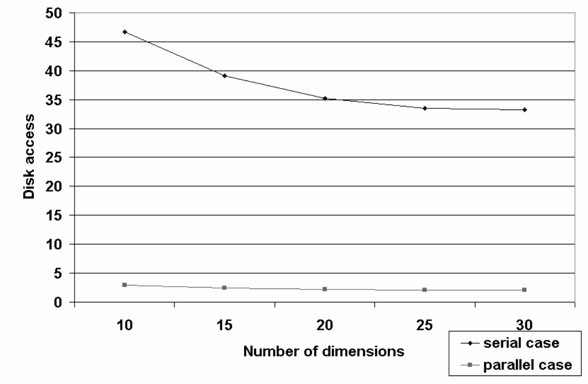

Figure 10. The dimensionality effect for a 10% search range.

4.4. The effect of the number of parallel processors

As described in Section 3.8, the *PN-tree* can be processed using multiple parallel processors. In this section, we study the effect of increasing the number of processors on the performance of the *PN-tree*. We perform the experiment using 100,000 uniformly distributed points in a 16-dimensional space. The number of parallel processors varies from one to 16.

The *PN-tree* can take advantage of as many processors as the dimensionality of the space. The result of the experiment is given in Figure 11.

From Figure 11, notice that the number of disk accesses decreases as the number of processors increases. However, the rate of decrease is not constant. It saturates as the number of processors increases. This saturation occurs because most of the features of the space get captured using the first dimensions. This implies that it is not necessary to use as many processors as the dimensionality of the space. A good performance can be obtained with a much less number of processors. As can be seen from the figure, the difference in the number of disk accesses between ten processors and sixteen processors is insignificant.

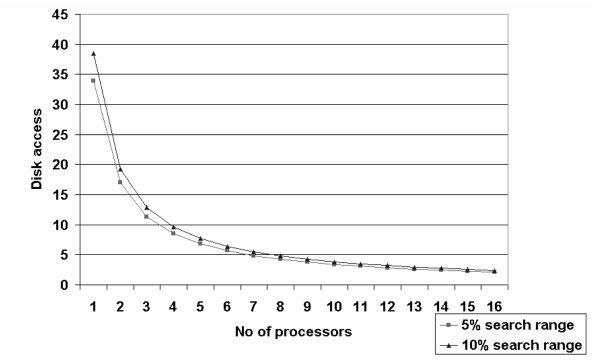

Figure 11. The effect of the number of parallel processors.

4.5. The effect of the number of inserted points

In this section, we study the performance of the *PN-tree* for various data sizes. We use a 16-dimensional space and vary the number of points from 10,000 to 200,000 points with increments of 10,000 points. The experiment is conducted for a 5% and a 10% search ranges. The results are given in Figures 12 and 13, respectively.

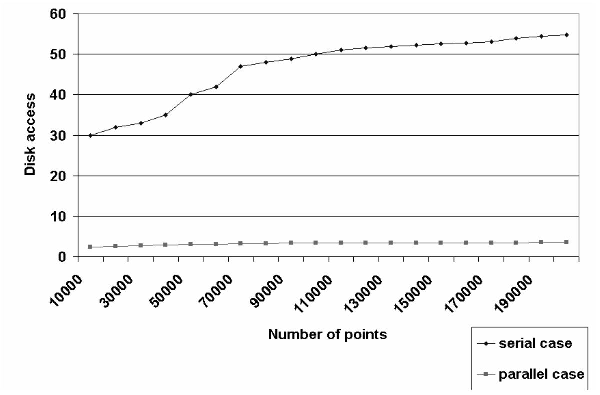

Figure 12. The effect of the number of inserted points (5% search range).

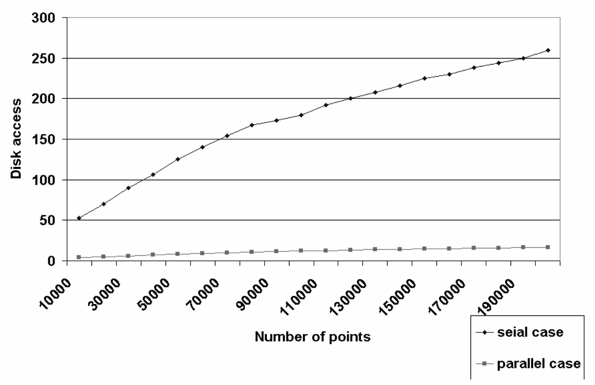

Figure 13. The effect of the number of inserted points (10% search range).

5. Comparative Study

In this section, the performance of the *PN-tree* is compared to the *Hybrid tree*. The *Hybrid tree* is selected because of its superiority over other index structures [6]. The two index structures have some commonalities. The node-splitting algorithm of the *PN-tree* is the same as the one for splitting data nodes in the *Hybrid tree*. Both structures select the split dimension as the one with the largest extent. Also, both structures select the split position so that the node is split into two halves along the split dimension provided that no utilization constraints are violated. Both structures try to minimize the increase in the expected disk access (EDA) as described in Section 3.

We test the performance of the two index structures using real data sets. The real data sets consist of 16-d vectors generated by extracting color histograms from the Corel database of images. These data sets have also been used by Chakrabarti and Mehrotra [6] while testing their *Hybrid tree*.

The experiment is carried out for circular range queries with various radii. Figure 14 gives the result for a radius that is equal to 5% of the whole space. It is noticed that parallel processing of the *PN-tree* improves the performance significantly. Moreover, the performance of the *PN-tree* using parallel processing scales very well with the increase in the number of inserted points.

It can be seen from Figure 14 that serial processing of the *PN-tree* outperforms the *Hybrid tree* when the number of points exceeds 150,000 points. This is because the *PN-tree* has some overhead involved in the processing of the one-dimensional indexes of each dimension. For example, if there are 16 dimensions and two disk accesses are needed to process each one-dimensional index structure, then there will be an overhead of 32 disk accesses involved in processing of indexes. This overhead is not justifiable unless a large number of points is inserted.

Figure 15 gives similar results for range queries with a radius that is equal to 10% of the whole space. Notice that the serial processing of the *PN-tree* outperforms the *Hybrid tree* after 70,000 points are inserted. This means that the *PN-tree* performs better under large search ranges. The reason is that as the radius of the search range increases, upon processing range queries, many search paths will be active at the same time in the *Hybrid tree*. This behavior degrades the search performance from logarithmic to linear.

In conclusion, the *PN-tree* improves the performance significantly with parallel processing and does not lose its efficiency with serial processing. In the serial case, the *PN-tree* outperforms the *Hybrid tree* for a large number of points and for large radii of search ranges.

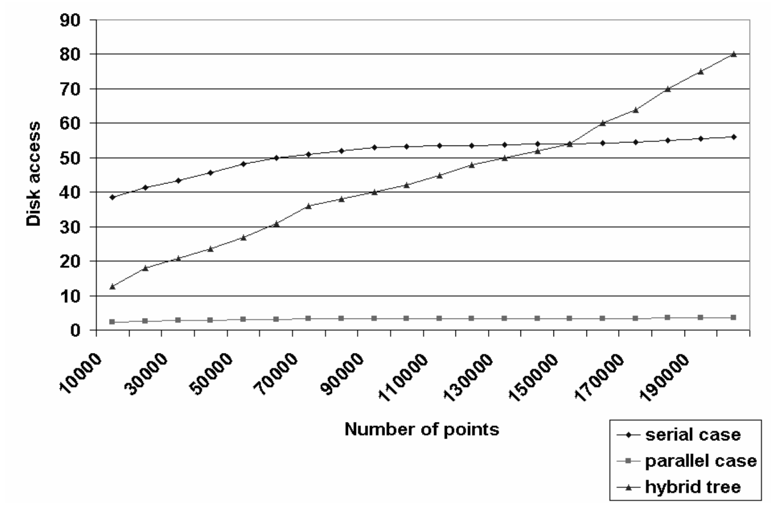

Figure 14. The *PN-tree* versus the Hybrid tree for 5% search range.

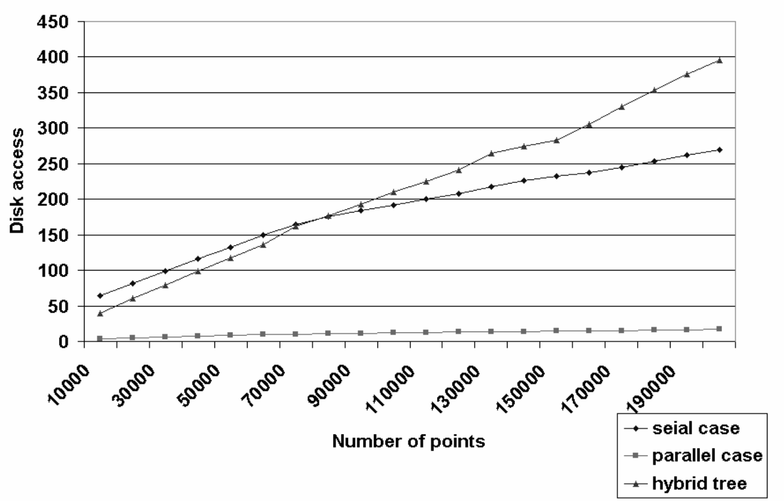

Figure 15. The *PN-tree* versus the Hybrid tree for 10% search range.

6. Conclusions

A new efficient index structure for multidimensional spaces has been presented. The new index structure is parallel and distributed in nature and can make use of as many processors as the dimensionality of the space. Moreover, the *PN-tree* does not lose its efficiency if it is serially processed over one processor.

The *PN-tree* clusters points multidimensionally into nodes by using simple insertion and splitting techniques. The projections of these nodes over each dimension are then indexed using any one-dimensional index structures.

The *PN-tree* scales with both the number of dimensions and the number of inserted points. Moreover, the *PN-tree* can be tuned to use parallel processors to compromise between the cost and the run-time speed of the query.

According to simulation results, the *PN-tree* outperforms already existing techniques for a large number of points and for large search ranges. In summary, the *PN-tree* is an efficient index structure for both serial and parallel indexing of multidimensional data sets.

Acknowledgments

This work was supported by the Department of Computer Science and Automatic Control, at the Faculty of Engineering, Alexandria University, Egypt.

The authors would like to express their appreciation to Prof. Walid Aref, Purdue University, for his fruitful discussion and valuable comments.

References

1. W. Aref and I. Ilyas, “SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees,” Journal of Intelligent Information Systems (JIIS), Volume 17 (215-240), 2001.
2. R. Bayer, “The universal B-Tree for multidimensional Indexing: General Concepts,” in World Wide Computing and its Applications (WWCA, 97), pp. 198-209, 1997.
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 Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 322-331, 1990.
4. J. L. Bentley, “Multidimensional binary search trees used for associative searching,” in the Communications of the ACM (CACM), 18(9), pp. 509-517, 1975.
5. S. Berchtold, D. A. Keim, H.P. Kriegel, “The X-tree: An Index Structure for High-Dimensional Data,” in Proceedings of the International Conference on Very Large Databases (VLDB), pp. 28-39, 1996.
6. K. Chakrabarti and S. Mehrotra, “High Dimensional Feature Indexing Using Hybrid Trees,” in Proceedings of the International Conference on Data Engineering (ICDE), pp. 440-447, 1999.
7. C. Faloutsos, I. Kamel, “High Performance R-trees,” IEEE Data Eng. Bull, 16(3), pp. 28-33, 1993.
8. M. Freeston, “The BANG file: a new kind of grid file,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, 260-269, 1987.
9. V. Gaede, O. Günther, “Multidimensional Access Methods,” ACM Computing Surveys, 30(2), pp. 170-231, 1998.
10. A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 47-57, 1984.
11. A. Henrich, H.W. Six, P. Widmayer, “The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects,” in Proceedings of the International Conference on Very Large Databases (VLDB), pp. 45-53, 1989.
12. K. Hinrichs, “Implementation of the grid file: Design concepts and experience,” BIT Journal, 25(4), pp. 569-592, 1985.
13. A. Hutflesz, H.W. SIX and P. Widmayer, “Twin Grid Files: Space Optimizing Access Schemes,” In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 183-190, 1988.
14. H.V. Jagadish, “On Indexing Line Segments,” in Proceedings of the International Conference on Very Large Databases (VLDB), pp. 614-625, 1990.
15. I. Kamel and C. Faloutsos, “Parallel R-trees,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 195-204, 1992.
16. N. Katayama and S. Satoh, “The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 369-380, 1997.
17. A. Klinger, “Patterns and search statistics,” In Optimizing Methods in Statistics, J.S. Rustagi, Ed., Academic Press, NY, pp. 303-337, 1971.
18. K.I. Lin, H.V. Jagadish, and C. Faloutsos, “The TV-Tree: An Index Structure for High-Dimensional Data,” VLDB Journal, 3(4), pp. 517-542, 1994.
19. D.B. Lomet and B. Salzberg, “The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance,” ACM Transaction on Database System (TODS), 15(4), pp. 625-658, 1990.
20. J. Nievergelt, H. Hinterberger and K. Sevick, “The Grid File: An Adaptable, Symmetric Multikey File Structure”, ACM Transactions On Database Systems (TODS), 9(1), 38-71, 1984.
21. J.T. Robinson, “The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes,” in Proceedings of the SIGMOD International Conference on Management of Data, pp. 10-18, 1981.
22. N. Roussopoulos, S. Kelly, and F. Vincent, “Nearest neighbor queries,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, pp.71-79, 1995.
23. H. Samet, “Spatial Data Structures,” Modern Database Systems: The Object Model, Interoperability, and Beyond, Addison Wesley / ACM Press, 361-385, 1995.
24. B. Seeger and H.P. Kriegel, “The buddy tree: an efficient and robust access method for spatial database systems,” in Proceedings of the International Conference on Very Large Databases (VLDB), 590-601, 1990.

25. T.K. Sellis, N. Roussopoulos, and C. Faloutsos, “Multidimensional access methods: Trees have grown everywhere,” in Proceedings of the International Conference on Very Large Databases (VLDB), 13-14, 1997.
26. H.W. Six and P. Widmayer, “Spatial searching in geometric databases,” in Proceedings of the International Conference on Data Engineering (ICDE), 496-503, 1988.
27. M. Tamminen, “The extendible cell method for closest point problems,” BIT Journal, 22(1), 27-41, 1982.
28. R. Weber, H. J. Schek and S. Blott, “A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces,” in Proceedings of the International Conference on Very Large Databases (VLDB), pp. 194-205, 1998.
29. D. White and R. Jain, “Similarity indexing with the SS-tree,” in Proceedings of International Conference on Data Engineering (ICDE), pp. 516–523, 1996.