

# ClusterTree: Integration of Cluster Representation and Nearest-Neighbor Search for Large Data Sets with High Dimensions

Dantong Yu and Aidong Zhang, *Member, IEEE*

**Abstract**—In this paper, we introduce the ClusterTree, a new indexing approach to representing clusters generated by any existing clustering approach. A cluster is decomposed into several subclusters and represented as the union of the subclusters. The subclusters can be further decomposed, which isolates the most related groups within the clusters. A ClusterTree is a hierarchy of clusters and subclusters which incorporates the cluster representation into the index structure to achieve effective and efficient retrieval. Our cluster representation is highly adaptive to any kind of cluster. It is well accepted that most existing indexing techniques degrade rapidly as the dimensions increase. The ClusterTree provides a practical solution to index clustered data sets and supports the retrieval of the nearest-neighbors effectively without having to linearly scan the high-dimensional data set. We also discuss an approach to dynamically reconstruct the ClusterTree when new data is added. We present the detailed analysis of this approach and justify it extensively by experiments.

**Index Terms**—Indexing, cluster representation, nearest-neighbor search, high-dimensional data sets.



## 1 INTRODUCTION

A **N** index structure organizes the whole data set to support efficient queries. Recently, many applications require efficient access and manipulation of large-scale multidimensional data sets. For example, many features extracted from image data sets are high-dimensional vectors [20], [28]. Also, in bioinformatics, gene expression data extracted from the DNA microarray images form large-scale multidimensional data sets. The high dimensions and enormous size of these data sets pose very challenging problems in indexing of the data sets for efficient querying. The design of indices to support high-dimensional data access has become an active research area.

Many approaches have been proposed to index multidimensional data sets. These approaches can efficiently support nearest-neighbor search for relatively low dimensional data sets [15], [3], [22]. Recently, most studies in index design [6], [5], [30], [17], [4] focus on high-dimensional data sets. Although most of these indexing strategies can insert data points dynamically, their performance might be affected by the insertion order of the new data points. The problem with a dynamic index structure is that the newly-inserted data points might cause that the structure no longer efficiently manages the whole data set. It can greatly increase the amount of data accessed for a query. When the dimensions increase and the data set is very large,

the efficiency for queries is a major consideration. To build an efficient index for a large data set with high dimensions, the overall data distributions or patterns should be considered to reduce the affects of arbitrary insertions. In some studies [12], [16], effective “packing” algorithms are developed which optimize the existing dynamic index by considering the data distribution.

Clustering is an analysis technique for discovering interesting data distributions and patterns in the underlying data set. Given a set of  $n$  data points in a  $d$ -dimensional metric space, a clustering approach assigns the data points to  $k$  groups ( $k \ll n$ ) based on the calculation of the degree of similarity<sup>1</sup> between data points such that the data points within a group are more similar to each other than the data points in different groups. In this approach, each group is a cluster. Supervised clustering approaches normally require a  $k$ , which is the number of groups, a priori, but not for unsupervised clustering approaches. Many excellent clustering algorithms were developed for discovering data patterns, but little research exists to incorporate them into index structures and similarity searches. Most index structures based on partition split a data set independent of its distribution patterns. These index structures have either a high degree of overlapping between bounding regions at high dimensions or inefficient space utilization. We observed that cluster structures of the data set can help build an index structure for high-dimensional data sets which supports efficient queries [25]. This motivates us to take into consideration the cluster information for indexing large scale data sets with high dimensions, and design

• The authors are with the Department of Computer Science and Engineering, State University of New York at Buffalo, Buffalo, NY 14260. E-mail: {dtyu, azhang}@cse.buffalo.edu.

Manuscript received 28 July 2000; revised 2 Mar. 2001; accepted 1 Oct. 2001. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112626.

1. The specific definition of similarity for the data sets may be application dependent.

similarity search algorithms which can choose an efficient searching order based on the structure of the data pattern.

In this paper, we present a novel dynamic indexing approach which provides a compact cluster representation to facilitate efficient querying. The indexing structure, termed ClusterTree, is a hierarchy of clusters and sub-clusters which incorporates the cluster representation into the index structure to achieve efficient retrieval. Our cluster representation is highly adaptive to any kind of cluster and can detect a new trend in the data distribution. The data points which are spatially close to one another are naturally grouped together in the ClusterTree. The ClusterTree provides a practical solution to index clustered data sets and supports the retrieval of the nearest-neighbors effectively and efficiently without having to linearly search the high-dimensional data set. Our goal is to minimize the response time to a user's query. The ClusterTree is one of the few that works towards building an efficient index structure by utilizing cluster information for high-dimensional data sets. We have conducted massive experiments to evaluate the performance of the given approach and report the comprehensive results. Our approach outperforms the SR-Tree and Pyramid-Tree in both synthetic and real data sets. The given approach can also be easily implemented in multiple disks system (RAID) [31].

The rest of the paper is organized as follows: Section 2 summarizes recent work on index structure design. Section 3 introduces our cluster representation approach called ClusterTree. Section 4 presents the query processing using the ClusterTree. Section 5 discusses the efficient dynamic reorganization of the ClusterTree. Section 6 presents the experimental results. Section 7 gives the conclusion.

## 2 RELATED WORK

Existing multidimensional tree-like indexing approaches can be classified into two categories: data and space partitioning.

### 2.1 Data Partitioning

A data partitioning approach partitions a data set and builds a hierarchy of bounding regions. Some examples of this type of index structure are the R-tree and its variants, the R\*-tree and R<sup>+</sup>-tree [15], [3], which support the nearest-neighbor search efficiently for low-dimensional data sets.

The R-Tree is a height-balanced tree with index records in its nodes. There are two kinds of nodes: internal and leaf nodes. The internal nodes contain pointers to their children, and the leaf nodes contain pointers to data objects. All the nodes have minimum bounding rectangles (MBR) as a page region. Each internal R-tree node contains a set of at most  $M$  children and at least  $\lceil \frac{M}{2} \rceil$  children. One disadvantage of R-trees is that the bounding boxes (rectangles) associated with different nodes may overlap. Therefore, when we search an R-tree, instead of following one path, we might follow multiple paths down the tree, although some branches do not contain relevant data.

The R\*-tree is an R-tree variant which has the following properties:

- Forced reinsert reduces the overlapping between the minimum bounding rectangles (MBR) of neighboring nodes.

- Storage utilization is improved.
- The volume of the MBRs is reduced.

When a point is inserted into a node, if there is no MBR in this node containing the point, the point will be inserted into a node with the least enlargement of the overlapping. Furthermore, splitting of a node can be avoided by forced reinsert. The performance of the R\*-tree is improved by about 70 percent over the R-tree.

The X-Tree [6] is an R\*-Tree-based index structure [3] which avoids the degeneration of the directory in high-dimensions using a special split algorithm and variable sized directory nodes. The X-Tree outperforms the R-Tree and the R\*-Tree significantly.

The SS-Tree [30] is a similarity indexing strategy for high-dimensional data sets. In contrast to the R-Tree, it uses hyperspheres as region units. Queries on the SS-Tree are very efficient because it only needs to calculate similarity between a region and the query point. For the insert operation, the SS-Tree is traversed by choosing the child nodes whose centroids are closest to the insert data point first. A *lazy* recalculation of the radius and centroid is designed to improve the insertion efficiency. When the chosen node for insertion overflows, the SS-Tree uses a *forced reinsert* similar to the R\*-Tree. It removes 30 percent of the children with the greatest distance from the centroid and reinserts them back into the SS-Tree using the same insert algorithm. If the children of the chosen node have already been reinserted, the node must be split. The split algorithm simply chooses the dimension with the highest coordinate variance between the centroid and the children's centroids, and finds a splitting location which minimizes the sum of the variances on each side of the split plane. Compared with the R\*-Tree, the SS-Tree has higher fanout because the hyperspheres for regions require half the storage space of the hyperrectangles. The SS-Tree outperforms the R-Tree and its variants on insertion and query as the dimensions increase.

The SS<sup>+</sup>-Tree [19] is a variant of the SS-Tree with a modified splitting heuristic to optimize the bounding shape for each node. Instead of finding a splitting plane having the maximum variances on both sides, the SS<sup>+</sup>-Tree uses the  $k$ -means cluster algorithm to divide the overflow node into two children. Therefore, the splitting method in the SS<sup>+</sup>-Tree reflects the data clustering and leads to less variance within the siblings when compared with the SS-Tree. To decrease the volume of the bounding shape for each node, the SS<sup>+</sup>-Tree uses the golden ratio method [29] to approximate the optimized data centroid for the node.

The SR-Tree [17] is a new index structure which combines the bounding spheres and rectangles for the shapes of node regions to reduce the blank area. The region for a node in the SR-Tree is represented by the intersection of a bounding sphere and a bounding rectangle. Thus, the overlapping area between two sibling nodes is reduced, particularly for high dimensions. The SR-Tree takes the advantages of both rectangles and spheres, and enhances the query performance remarkably. However, the storage required for the SR-Tree is larger than the SS-Tree because the nodes in the SR-Tree need to store the bounding rectangles and bounding spheres.

Consequently, the SR-Tree requires more CPU time and more disk accesses than the SS-Tree for insertions.

Another example of data partitioning is the M-Tree [10], which organizes and searches the data based on a distance function, where the distance metric is not limited by  $L_p$  norm. M-Tree considers relative (dis)similarity between the data objects.

## 2.2 Space Partitioning

Space partitioning approaches divide a data space into disjoint subspaces. A hierarchy of subspaces can be generated from the process of dividing.

The K-D-B Tree [22] is one of the earliest multidimensional index structures based on space partitioning. It partitions a  $d$ -dimensional data space into disjoint subspaces by  $(d-1)$ -dimensional hyperplanes which are alternately perpendicular to one of the dimension axes. Therefore, the subspaces are represented by hyperrectangles based on comparison with one element of a single domain. The K-D-B Tree forces splitting to maintain a balanced structure. That is, an overflow node will be split. Splitting can be propagated to the descendants because the descendants of the node may intersect with the splitting plane. This recursive splitting might cause sparse or empty nodes, thus optimized storage utilization cannot be guaranteed. Complete partitioning in the K-D-B tree generates many page regions and becomes exponential as the dimensions increase.

The Pyramid-Tree [5] is based on a special partitioning strategy which focuses on high-dimensional data sets. The basic idea is to divide the data space first into  $2d$  pyramids, each sharing the center point as its peak (the tip point of a pyramid). Each pyramid is then sliced into slices parallel to the base of the pyramid, and each slice forms a data page. The range query under this index structure can be efficiently processed for both low- and high-dimensional data sets, and is not affected by the so-called *curse of dimensionality* when the data distribution is uniform. The Pyramid-Tree outperforms the X-Tree [6] by a factor of up to 800 in terms of the response time for queries. However, the partitioning of the Pyramid-Tree cannot ensure that the data points in one data page are always neighbors. The slices close to the center contain the data points which are close to each other, while the slices close to the base of the pyramid may contain the data points which are not similar. The queries touching the boundary of the data space cannot be handled efficiently. Since the slices do not have simple shapes like hypersphere or hyperrectangle, it is very hard to estimate the distance between a query point and the slices, thus the  $k$ -nearest-neighbor search might not be supported efficiently. Also, the performance of the Pyramid-Tree is affected by data distributions. The extended Pyramid-Tree chooses the center of the data distribution as the top of the pyramids for better performance. But, when the data distribution is arbitrary, it is very hard to locate the center of the data space.

The Hybrid-Tree [9] combines positive aspects of both K-D-B Tree and R-Tree index structures into a single data structure to achieve good performance scalable to high dimensions. Recently, DBIN [4] was developed, which is one of a few indexing algorithms that can incorporate cluster information in the nearest-neighbor search.

## 3 CLUSTERTREE AND ITS CONSTRUCTION

In this section, we will first introduce the hierarchical structure of the *ClusterTree*. We will then present an approach to decomposing a cluster into subclusters and the algorithm to generate a ClusterTree by decomposing clusters recursively.

### 3.1 The ClusterTree

A ClusterTree is a hierarchical representation of the clusters of a data set. A ClusterTree organizes the data based on their different levels of clustering information, from coarse to fine, providing an efficient index structure of the data. Each nonleaf node in the ClusterTree is defined as:

$$\begin{aligned} \text{Node:} & \quad [Node\_id, \gamma, (Entry_1, Entry_2, \dots, Entry_\gamma)] \quad (m_{node} \leq \gamma \leq M_{node}), \\ \text{Entry}_i: & \quad (SC_i, BS_i, SN_i), \end{aligned}$$

where  $Node\_id$  is the node identifier,  $\gamma$  is the number of the entries in the node, and  $m_{node}$  and  $M_{node}$  define the minimum and maximum number of entries in the node. An entry is created for each subcluster of the cluster which the current nonleaf node represents. In entry  $Entry_i$ ,  $SC_i$  is a pointer to the  $i$ th subcluster,  $BS_i$  is the bounding sphere for the subcluster and  $SN_i$  is the number of data points in the  $i$ th subcluster. The bounding sphere is represented by  $(c, r)$ , where  $c$  is the center and  $r$  is the radius.

Each leaf node contains pointers to the data points and has the structure:

$$\begin{aligned} \text{Leaf:} & \quad [Leaf\_id, \gamma, (Entry_1, Entry_2, \dots, Entry_\gamma)] \\ & \quad (m_{leaf} \leq \gamma \leq M_{leaf}), \end{aligned}$$

where  $\gamma$  is the number of data points contained in the leaf node, and  $m_{leaf}$  and  $M_{leaf}$  are the minimum and maximum number of entries.  $Entry_i$  contains the address of the data point residing in the secondary storage. If we store all the data points belonging to the same leaf node in linear array-like style, then for each leaf node, only the first and last entries need to be saved. All data points that belong to the leaf node are saved between the first and last data points. Furthermore, I/O time can be greatly reduced by bulk-loading disk blocks which contain the sequentially stored data points. If the data points are randomly saved in the storage media, their addresses will be saved in the leaf node. This provides the flexibility of building index structure without physically moving the data points. We define the *level* of a node in a ClusterTree as the length of the path from the root to this node, beginning with 0. It is desirable that a ClusterTree is balanced so that the levels of the leaf nodes are approximately equal.

### 3.2 Cluster and its Representation

Let the input data set consist of  $d$ -dimensional points which are already clustered. For the input data without cluster information, we can apply the cluster algorithms in [14], [21], [26], [27] to detect clusters. For each cluster, we calculate the following parameters: The number of data points, the centroid  $c$ , and the volume of the minimum bounding sphere  $S$ . The centroid  $c = \langle c_1, c_2, \dots, c_d \rangle$  can be calculated by:

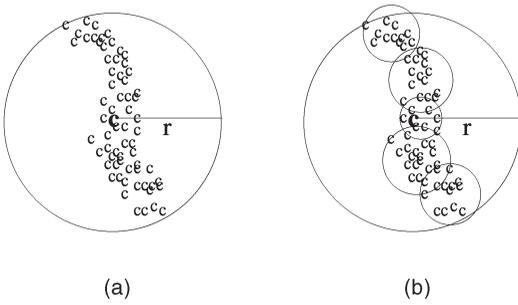


Fig. 1. Decomposition of cluster  $C$ . (a) Original cluster  $C$ . (b) Decomposed cluster  $C$ .

$$c_i = \frac{\sum_{j=1}^N o_{ji}}{N}, 1 \leq i \leq d, \quad (1)$$

where  $N$  is the number of the data points in the cluster and  $o_{ji}$  is the  $i$ th value of data point  $o_j$  in the cluster. Thus, each cluster is represented by a hypersphere  $S$ .

### 3.2.1 Splitting a Cluster

Due to the arbitrary distribution of the data set, the data points within a cluster may have arbitrary shape. The sphere used to represent the cluster must be big enough to cover all the data points in the cluster. Within the sphere, there may be some empty regions which contain no data. When the dimensions go higher, the empty regions will occupy most of the space bounded by the sphere. Also, two bounding hyperspheres of two different clusters may overlap, even though the two clusters do not intersect. For the data points which belong to only one cluster, the bounding sphere of the other cluster may contain them, which may lead us to make wrong decision of the clustering information about these data points. Thus, the overlapping between two hyperspheres might cause inaccuracy.

For example, to cover the cluster  $C$  in Fig. 1a, a sphere  $S$  centered at  $c$  should at least have radius  $r$ . Here, we define the density of the cluster as:

$$\begin{aligned} \text{Density}_c &= \frac{\text{number of points in } C}{\text{volume of } S} \\ &= \frac{\text{number of points in } C}{\frac{2\pi^{d/2}r^d}{d\Gamma(\frac{d}{2})}}. \end{aligned}$$

The gamma function  $\Gamma(x)$  is defined as:

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt,$$

where  $\Gamma(x+1) = x\Gamma(x)$  and  $\Gamma(1) = 1$ . When the number of dimensions is high ( $> 20$ ) and  $r > 1$ , the volume of a hypersphere can be huge. The density will be close to 0. Thus, we simplify the volume of  $S$  as:  $\text{vol}(S) = r^{\log d}$ . When the density of a cluster falls below a preselected threshold or the number of the data points in the cluster is larger than a preselected threshold, the cluster will be decomposed into several smaller clusters. We call them *subclusters*. The subclusters can be further decomposed to obtain more compact representation. We can split a big cluster by expanding a  $k$ -medoid method [21] (see below). Fig. 1b shows how a cluster can be decomposed.

### 3.2.2 $k$ -medoid Algorithm to Split a Cluster

The selection of  $k$ -medoids is crucial in the  $k$ -medoid algorithm. An efficient and effective algorithm for selecting the  $k$ -medoids must preserve these conditions:

- *Representativeness.*  $k$  medoids must well represent the whole data set.
- *Minimum distance.* The distance of each data point to its medoid should be as small as possible.
- *Balance.* The partition of the data set by these medoids should be balanced.
- *Efficiency.* The quadratic and more expensive algorithm for selecting medoids should be avoided.

The optimal algorithm for searching the best medoids is to examine all candidate combinations of  $k$  data points from the cluster which contains  $n$  data points ( $k \ll n$ ). Given each candidate set of  $k$ -medoids, for each data point in the cluster, calculate the distance between the data point to its closest medoids in the candidate set. The time complexity for examining all the  $k$ -combinations in the cluster and choosing the closest medoid for each data point is:

$$\binom{n}{k} \cdot k \cdot (n - k).$$

Some improvement has been made, such as PAM in [18], but it needs to search for a minimum on the graph  $G_{n,k}$ , where  $G_{n,k}$  represents the searching model. For large values of  $n$  and  $k$ , examining all nodes in the graph is time consuming and accounts for the inefficiency of PAM to deal with very large clusters.

We now present an approach to split a cluster using  $k$  medoids. We measure the performance of splitting a cluster by the inverse of the total volume of all subclusters generated by that splitting, i.e.,

$$\frac{1}{\sum_1^k \text{vol}(\text{Cluster}_i)},$$

where  $\text{vol}(\text{Cluster}_i)$  calculates the volume of  $\text{Cluster}_i$  and equals the volume of the subcluster's bounding sphere. That is, the cluster partition based on the subclusters with the minimum volumes will be better because it always has smaller overlapping and provides a compact representation for the cluster.

**$k$ -medoids determination.** The greedy method presented in [13], [1] can be used to select the good medoids. We first randomly select  $k \times \lambda$  points from the cluster and put them into a sample set. We refine the sample set to form a candidate medoid set  $\mathcal{A}$  in which every medoid should be as far away as possible from each other. The optimal selection strategy is an *NP* problem for nonfixed  $k$ . We use a heuristic algorithm to solve the problem. First, we randomly pick a data point  $o$  and put it into the set  $\mathcal{A}$  of the candidate medoids. Then, we calculate the distance of each data point in the sample set to the data point  $o$ , choose the data point with the maximum distance, put it again into the candidate set  $\mathcal{A}$ , and recompute the distance from each point in the sample set to its nearest medoid. We continue to do this until the  $4 \times k$  points have been chosen for  $\mathcal{A}$ .

The hill-climbing algorithm in [21] is used to choose the best  $k$ -medoids to decrease the total volumes of all

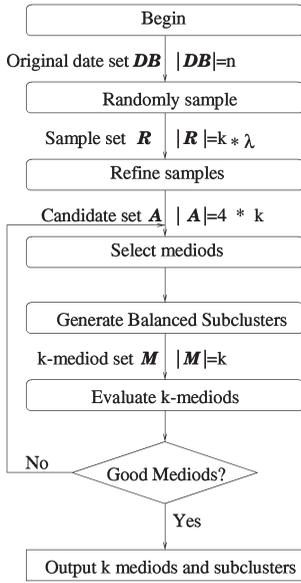


Fig. 2. Decomposition of a single cluster into  $k$  subclusters.

subclusters, and thus improve the performance. Using the same method of choosing the candidate set  $\mathcal{A}$ , we can efficiently generate an initial  $k$ -mediod set. To refine this  $k$ -mediod set, we first pick the subcluster with the lowest density or the minimum number of data points. A subcluster, ( $SC$ ), may have very low density or be very small, because:

1. the subcluster is formed by outliers,
2.  $SC$  is part of a natural subcluster  $SC'$  and the majority of data points in  $SC'$  have been assigned to another subcluster  $SC''$ , or
3.  $SC$  is likely distorted by outliers or the data points which may falsely be assigned to  $SC$ .

We choose a new mediod from the remaining points in the candidate set to replace the bad mediod, and repeat the evaluation procedure until the performance does not change. After  $k$ -mediods are chosen, we assign each data point in the cluster to the closest mediod. This partition of the data points in a cluster will generate  $k$  subclusters. But this partition might generate unbalanced structure, which greatly affects the performance of the ClusterTree. In the following section, we present a balancing algorithm of redistributing data points to prevent this happening. Fig. 2 describes the procedure of generating  $k$  subclusters.

### 3.2.3 Generating Balanced Subclusters for a Cluster

The above approach does not necessarily generate a balanced partition for the cluster. Nonbalanced partitions and the inferred index structure are not efficient in terms of time and space. The worst case of the unbalanced partitions would be among all of the partitions (subclusters), only one subcluster is significant because most of the data points are aggregated into it and all of the other subclusters are trivial. Such unbalanced partitioning may occur repetitively in the further splittings. As a result, the height of the splitting is linearly comparable to the size of the cluster. The performance of a search on the unbalanced index structure is highly unpredictable. The extreme case would be a linear

search through the whole data set. Fig. 3a describes the unbalanced partitions and the generated index structure.

It is unlikely to always satisfy all of the criteria listed in Section 3.2.2 for a good  $k$ -mediod partition because the minimum distance criterion may conflict with the balancing requirement. Thus, we describe a heuristic algorithm (Fig. 4, Algorithm 3.2.3) that balances these two criteria. The input to the algorithm is a  $k$ -mediod set. The output is  $k$  nearly balanced subclusters. We do not generate the subclusters with exactly the same size because it would destroy the structure of the subclusters. In Algorithm 3.2.3, Step 1 assigns each data point in cluster  $C$  to its closest subcluster. Step 2 checks whether there is a subcluster which includes more than half of the data points. We know that there is at most one subcluster whose size can exceed  $\frac{n}{2}$ , where  $n$  is the number of the data points in cluster  $C$ . At Step 3, we shrink the size of the biggest subcluster  $SC_i$  by picking the data points which fall far away from their mediod  $m_i$  and reassigning them to other subclusters. One method is to sort all of the data points by their distances to the mediod, and then pick  $|SC_i| - \frac{n}{2}$  data points with the longest distance. The general sorting algorithm takes about  $O(n \log n)$  time. If the data set is huge, sorting is time consuming. We use algorithm SELECT<sup>3</sup> in [11] to decrease the size of  $SC_i$ . Given  $i$  and an array  $A$ , SELECT returns the  $i$ th smallest elements of the array  $A$ . It was proved in [11] that the average time complexity of SELECT is linear. Let  $D(SC_i)$  be the array of the distances between the data points in  $SC_i$  and the mediod  $m_i$ , the median distance  $d_{med}$  of  $D(SC_i)$  is calculated as:  $d_{med} = \text{SELECT}(D(SC_i), \lfloor \frac{n}{2} \rfloor)$ . Then, the data points are divided into two groups:  $G_1$  and  $G_2$ , where  $G_1 = \{o | d(o, m_i) \geq d_{med}\}$  and  $G_2 = SC_i - G_1$ . Let  $SC_i = G_2$ . We shrink the size of  $SC_i$  to be smaller than  $\frac{n}{2}$ .

In Algorithm 3.2.3, the function *ChooseOther Mediod* is used to pick a new subcluster for a data point. The selected subcluster should be as close as possible to the data point. Also, the subcluster selected for insertion needs the least enlargement to include the data point. After insertion, the number of data points should still be below  $\frac{n}{2}$ . Thus, Algorithm 3.2.3 is guaranteed to generate  $k$  subclusters whose sizes are all smaller than  $\frac{n}{2}$ .

### 3.3 Algorithm for Building ClusterTree

We now present an algorithm to build a height-balanced ClusterTree. Algorithm 3.3 gives the details of generating a ClusterTree. In Algorithm 3.3, *pageSize* is determined by the disk block size ( $M$ ) and the size of a data point ( $\xi$ ), where  $\xi = \text{dimensions} \times \text{sizeof}(\text{element of a data point})$ . *pageSize* can be calculated by the formula:  $\text{pageSize} = \frac{M}{\xi}$ . Stack is used to store the nodes which need to be processed. While the stack is not empty, the node on top of the stack will be popped. The child nodes will be created for each of the entries belonging to the popped node. If some of the child nodes need to be further split, then they will be pushed into the stack. When the stack is empty, which means that all of the nodes are processed, the procedure of creating the ClusterTree is finished.

**Height of ClusterTree.** A balanced partitioning algorithm was introduced in Section 3.2.3. Using it at Step 3-c' in Algorithm 3.3, (see Fig. 5) we can generate  $k$  roughly

2. We use  $|A|$  to denote the number of elements in set  $A$  (cardinality).  
3. See Appendix A.

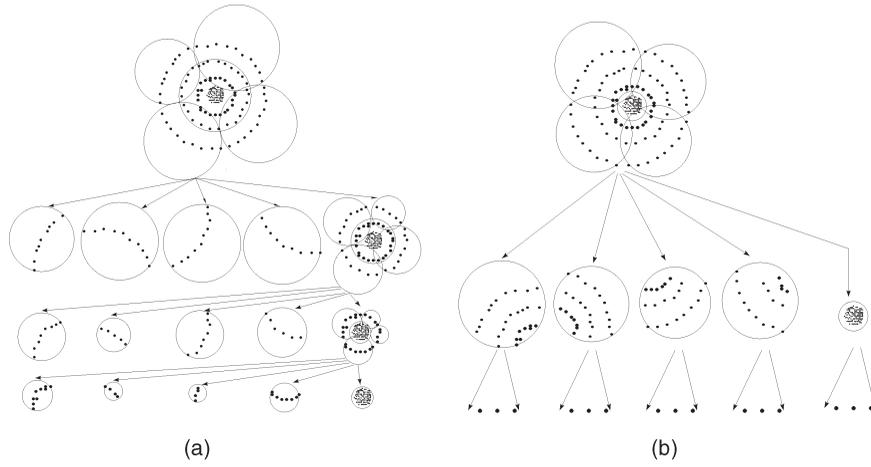


Fig. 3. (a) An unbalanced tree can cause a linear search over the whole data set. (b) A balanced partition for the cluster.

balanced subclusters whose sizes are no more than half of the parent cluster's. After the partitioning algorithm is applied  $O(\log N)$  times, where  $N$  is the size of the whole data set, the number of data points in each leaf node becomes equal to or less than the constant *pageSize*. Thus, the maximum height of the ClusterTree is  $O(\log N)$ .

The medoid selection method in Section 3.2.2 theoretically leads to balanced splitting [1]. In practice, the splitting method might still create trivial leaf nodes which contain few data points. We can apply a simple postprocessing technique to merge all of the leaf node whose disk block utilization is no more than  $\frac{1}{k} \times \text{PageCapacity}$ . So, several trivial leaf nodes may be merged into one leaf node which is inserted to the parent node.

## 4 SIMILARITY QUERIES

Similarity queries can be classified into two categories: range queries and  $p$ -nearest-neighbor ( $p$ -NN) queries. The definitions for these two categories are given in [24]:

**Definition 1.** (*Range query*). Given a query point  $\mathbf{q}$  and a range parameter  $\epsilon$ . The result set  $NN_\epsilon(\mathbf{q})$  for the range query in data

set  $DB$  is defined as:  $NN_\epsilon(\mathbf{q}) = \{\mathbf{o} | d(\mathbf{q}, \mathbf{o}) \leq \epsilon \text{ and } \mathbf{o} \in DB\}$ , where  $d$  is a distance measurement.

**Definition 2.** ( *$p$ -NN query*). For a query point  $\mathbf{q}$  and a query parameter  $p$ , the query returns the smallest set  $NN^p(\mathbf{q}) \subseteq DB$  that contains  $p$  data points from the data set, and for which the following condition holds:

$$\forall \mathbf{o} \in NN^p(\mathbf{q}), \forall \mathbf{o}' \in (DB - NN^p(\mathbf{q})) : d(\mathbf{o}, \mathbf{q}) \leq d(\mathbf{o}', \mathbf{q}).$$

### 4.1 Range Query

A range query is represented as a sphere which is centered at  $\mathbf{q}$  with radius  $\epsilon$ . In the ClusterTree structure, the range query should be performed in the related clusters which intersect the query. There is a high probability that the query will intersect several clusters, so these clusters can be selected simultaneously. Other clusters which do not intersect with the query sphere will not be considered. We perform a recursive search in each cluster. Fig. 6 shows how to do a query. At the first level, the query sphere intersects with three clusters **A**, **B**, and **C**. So these three clusters will be searched. Cluster **A** has four subclusters, of these, only

#### Algorithm 3.2.3: Generate Subcluster

Input:  $k$ -medoid set  $\mathcal{M}$  for a cluster  $\mathcal{C}$ ,  $|\mathcal{C}| = n$

Output:  $k$  semi-balanced subclusters  $\{SC_1, SC_2, \dots, SC_k\}$  for  $\mathcal{C}$

1. **for** each  $\mathbf{o} \in \mathcal{C}$ ,  $1 \leq j \leq n$ 
  - a. calculate its close medoid  $\mathbf{m}_j$
  - b. increase the number of points in  $SC_j$ , i.e.  $|SC_j| = |SC_j| + 1$
2. Get the maximum subcluster  $SC_l$ , where for any  $1 \leq i \leq k$ ,  $|SC_i| \leq |SC_l|$   
**if**  $|SC_l| \leq \frac{n}{2}$  **then return**; **else goto** step 3.
3. Decrease the number of data points in  $SC_l$  as follows:
  - a.  $d_{med} = \text{SELECT}(D(SC_l), \lfloor \frac{n}{2} \rfloor)$ .
  - b. **for** each data point  $\mathbf{o} \in SC_l$   
**if**  $d(\mathbf{o}, \mathbf{m}_l) > d_{med}$   
 $x = \text{ChooseOtherMediod}(\mathcal{M}, \mathbf{o})$ ;  
Put  $\mathbf{o}$  in  $SC_x$ , and  $|SC_x| = |SC_x| + 1$ .

Fig. 4. Algorithm 3.2.3.

**Algorithm 3.3: Build ClusterTree**Input :  $\beta$  clusters  $\{C_1, C_2, \dots, C_\beta\}$ Output: Index structure for the  $\beta$  clusters

1. Generate a root to represent all clusters;  
Create an entry  $E_i$  for each cluster  $C_i$ , add them into the root node;  
Push the root into *Stack*.
2. **if** *Stack* is not empty,  $currentNode = pop(Stack)$  and **goto** step 3;  
**else return**;
3. **for** each entry  $Entry_i$  in  $currentNode$ , where  $1 \leq i \leq \gamma$   
**if**  $Entry_i.SN \leq pageSize$  **then**  
    a. Create a leaf node  $child_i$  for  $Entry_i$ .  
    b. Let the pointer  $SC_i$  in  $Entry_i$  point to  $child_i$ .  
    c. Save data points to disk pages.  
**else**  
    a'. Create a non-leaf node  $child_i$  for  $Entry_i$ .  
    b'. Let the pointer  $SC_i$  in  $Entry_i$  point to  $child_i$ .  
    c'. generate  $k$ -medoids, create  $k$  subclusters  $child_i, 1 \leq i \leq k$ .  
    d'. Create an entry for each subcluster and add them to  $child_i$ .  
    e'. Push  $child_i$  into *Stack*.
4. **goto** step 2.

Fig. 5. Algorithm 3.3.

**A4** intersects with the query sphere. Of the subclusters of **B**, only subcluster **B4** intersects with the query sphere. At the bottom of the ClusterTree, we only need to search subclusters **A4** and **B4**. By checking the data points in the two subclusters, **B4** has three points within the query range. Thus, **B4** will be selected as the most related cluster and the query result set  $NN_\epsilon(\mathbf{q})$  is  $\{r_1, r_2, r_3\}$ .

**4.2  $p$ -Nearest-Neighbors Query****4.2.1 Distance Measurement for Nearest-Neighbor Search**

Given a query point  $\mathbf{q}$  and a ClusterTree, the search will look through the nodes of the ClusterTree to find the  $p$ -Nearest-Neighbors. We provide three distance measurements for ordering the nodes involved in the search: the average distance ( $d_{avr}$ ), the maximum distance ( $d_{max}$ ), and the minimum distance ( $d_{min}$ ) between  $\mathbf{q}$  and the bounding hypersphere of a node, where  $d_{min}$  and  $d_{max}$  give the lower and upper bounds on the actual distance of  $\mathbf{q}$  from data points in the node, and  $d_{avr}$  estimates the average distance of  $\mathbf{q}$  from data points in the node. When we traverse the ClusterTree, the three distance measurements are needed to prune the search paths within the ClusterTree. These measurements are defined as follows ( $Sphere(c, r)$  is the bounding sphere of the node, which is centered at  $\mathbf{c}$  with  $r$  as the radius.):

$$\begin{aligned} d_{avr}(\mathbf{q}, Sphere(\mathbf{c}, r)) &= d(\mathbf{q}, \mathbf{c}), \\ d_{max}(\mathbf{q}, Sphere(\mathbf{c}, r)) &= d(\mathbf{q}, \mathbf{c}) + r, \\ d_{min}(\mathbf{q}, Sphere(\mathbf{c}, r)) &= \begin{cases} d(\mathbf{q}, \mathbf{c}) - r & \text{if } d(\mathbf{q}, \mathbf{c}) > r, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

We define the average distance between an entry in a node and the query point as:

$$d_{avr}(\mathbf{q}, Entry(SC, BS, SN)) = d_{avr}(\mathbf{q}, BS).$$

Similarly, we can define the maximum and minimum distances between an entry and the query.

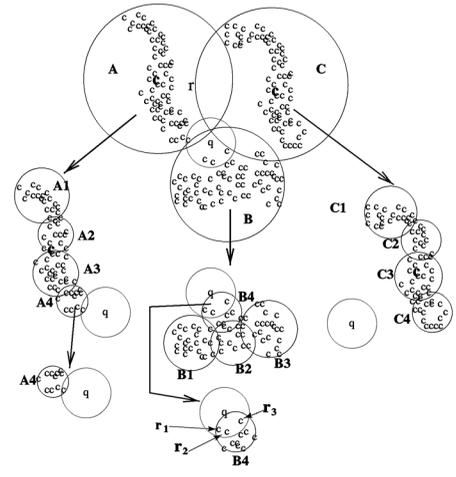


Fig. 6. A range query is performed by recursively traversing the hierarchical structure.

The  $p$ -NN query can be converted to a range query, given that we can find a threshold  $d_p$  such that  $NN^p(\mathbf{q}) \subseteq NN_{d_p}(\mathbf{q})$  and  $\forall d_p' < d_p, NN^p(\mathbf{q}) \not\subseteq NN_{d_p'}(\mathbf{q})$ , where

$$d_p = \max_{\mathbf{o} \in NN^p(\mathbf{q})} \{d(\mathbf{o}, \mathbf{q})\}.$$

If the minimum threshold  $d_p$  is known in advance, then the problem can be solved using a range query, but  $d_p$  is normally unknown in advance. By specifying a threshold  $d_p$  for a range query, we can still perform the  $p$ -nearest-neighbor query. However, the selection of the threshold  $d_p$  is difficult because users may not know the data set in detail and the dissimilarity cannot always be smoothly mapped into a distance parameter  $d_p$  in the Euclidean metric. If  $d_p$  is too small, then not enough data will be retrieved. If  $d_p$  is too big, then many unnecessary nodes in the ClusterTree will be checked. The selection of  $d_p$  plays a crucial role in the efficiency of the query algorithm. The following property can help in adjusting the value of  $d_p$ .

**Property 1.** For a given query  $\mathbf{q}$  and data set  $DB$ , if  $|NN_\epsilon(\mathbf{q})| \geq p$ , then  $NN^p(\mathbf{q}) \subseteq NN_\epsilon(\mathbf{q})$  and  $\epsilon \geq D_p$ , where  $D_p$  is the minimum value of  $d_p$  and  $D_p = \max_{\mathbf{o} \in NN^p(\mathbf{q})} \{d(\mathbf{o}, \mathbf{q})\}$ .

**Proof.** Suppose  $NN^p(\mathbf{q}) \not\subseteq NN_\epsilon(\mathbf{q})$ . There exists a data point  $r$  such that  $r \in NN^p(\mathbf{q})$  and  $r \notin NN_\epsilon(\mathbf{q})$ . Because  $|NN_\epsilon(\mathbf{q})| \geq p$ , there also exists a data point  $r'$  such that  $r' \in NN_\epsilon(\mathbf{q})$  and  $r' \notin NN^p(\mathbf{q})$ . Therefore,

$$d(r', \mathbf{q}) < d(r, \mathbf{q}),$$

which contradicts the definition of the  $p$ -nearest-neighbors.  $\square$

There might be more than one data point sharing the same  $p$ th distance to the query point  $\mathbf{q}$ , so the  $p$ -nearest-neighbor set  $NN^p(\mathbf{q})$  might not be unique. In this case, most implementations nondeterministically pick some data points with the  $p$ th distance value. Property 1 still holds for all of the possible  $p$ -nearest-neighbor sets. We provide the following algorithm to initialize the  $d_p$ .

First, it sorts all entries in the root in increasing order of the maximum distance  $d_{max}$  between an entry and the query  $\mathbf{q}$ . Then, it scans the entries and sums up the number of data points in them until it reaches an entry  $Entry_j$  such that the

total number of data points in the scanned entries exceeds the query parameter  $p$ . Based on Property 1, we know that  $d_{max}(\mathbf{q}, Entry_j) \geq D_p$ . The initial value of  $d_p$  will be set to  $d_{max}(\mathbf{q}, Entry_j)$ . In the rest of this section, we will discuss the  $p$ -nearest-neighbors query based on the range search. Our goal is to minimize the number of accessed nodes.

We start from the root of the ClusterTree with the rough estimate of the threshold  $d_p$  and perform a depth-first search until the leaf level is reached. During the traversal, we prune many nodes based on their distance to the query. Also, while traversing the ClusterTree, based on the information available at each node, we dynamically adjust the value of  $d_p$  so that it approaches  $D_p$ . This further helps in pruning more nodes to improve the efficiency.

We make use of Property 1 to adjust the value of  $d_p$ . Let the current search range be a hypersphere with the center  $\mathbf{q}$  and having radius  $d_p$ , denoted by  $Sphere(\mathbf{q}, d_p)$ . Recall that every internal node  $V$  stores the information  $(SC_i, BS_i, SN_i)$  for each of its children. We prune nodes and decrease the current threshold  $d_p$  according to the following two rules:

1. *Prune.* At an internal node  $V$ , for each child  $child_i$ , consider  $BS_i$ . If  $d_{min}(\mathbf{q}, BS_i) > d_p$ , then we prune the subtree rooted at  $child_i$  because none of the data points corresponding to  $child_i$  lie inside the query sphere.
2. *Adjust.* For each child  $child_i$  in an internal node  $V$ , if  $d_{max}(\mathbf{q}, BS_i) \leq d_p$  and the number of the data points that belong to this entry is greater than  $p$ , then  $d_p$  can be replaced by  $d_{max}(\mathbf{q}, BS)$ , because  $NN^p(\mathbf{q}) \subseteq NN_{d_{max}(\mathbf{q}, BS)}(\mathbf{q})$ , according to Property 1. If the node is a leaf and its data points are within  $Sphere(\mathbf{q}, d_p)$ , then the data points are inserted into a result buffer and  $d_p$  is updated.

#### 4.2.2 Nearest-Neighbor Search Algorithm for ClusterTree

We now present a revised *branch-and-bound* algorithm which finds the nearest-neighbors, discuss the efficiency of the search algorithm based on  $d_{avr}$  and  $d_{min}$  ordering, address the disadvantages of both orderings and, finally, provide a combined search algorithm for nearest-neighbor search to enhance the pruning efficiency.

**Search order among siblings.** The order of visiting the sibling branches rooted at the current node can affect the efficiency of pruning. If the branch close to the query is picked first, the threshold  $d_p$  can be decreased. When the search later comes to another sibling branch which is further than the previous one, this branch can be pruned immediately. Therefore, the search ordering should reflect the probability that the nodes contain the query's results, and the nodes with high possibility should be searched first. Usually, the possibility for a node containing the query results is estimated by the minimum distance  $d_{min}$  or average distance  $d_{avr}$ . The search algorithm using minimum distance based ordering is called *minimum distance based search*(MDS), and the one using average distance based ordering is called *average distance based search*(ADS).

There are other factors affecting the accuracy of using the distances  $d_{min}$  or  $d_{avr}$  to estimate the possibility for a node to contain the query results, such as the layout of a node or the

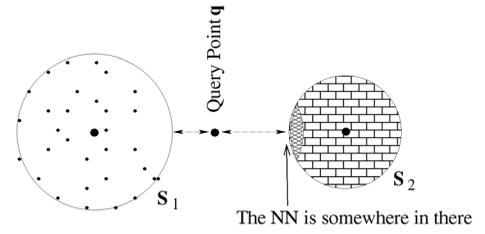


Fig. 7. Both  $d_{avr}$  and  $d_{min}$  cannot give good ordering.

data distribution within the node. When the data layout in a node is compact, there is a high probability (confidence) that some data points are located on the border region close to the query point.  $d_{min}$  can give a good estimate of the distance between the query point and its nearest points within the node.  $d_{min}$  shows better performance in [23] because when the dimensions are low, R-tree is very efficient and nodes are relatively compact. But when the node is sparse, the  $d_{min}$  cannot give a good estimation (see example in [23]). For these types of nodes, the center of the bounding sphere has a relatively higher concentration of data points compared with the surface of the bounding sphere. Only when the query is close to the centroid, will it have a high chance of obtaining more neighboring points from this node. Therefore, the average distance  $d_{avr}$  has better estimation of the possibility than  $d_{min}$  in this case.

Even in a single index structure, neither  $d_{min}$  nor  $d_{avr}$  has uniformly good estimation on the search order because the nodes in an index structure do not have the same data distribution. In Fig. 7,  $d_{min}(\mathbf{q}, S_1) < d_{min}(\mathbf{q}, S_2)$ , and  $d_{avr}(\mathbf{q}, S_1) < d_{avr}(\mathbf{q}, S_2)$ . Both MDS and ADS will choose  $S_1$  first, which does not contain the nearest-neighbors. Therefore, we define an optimized distance  $d_{opt}$  to determine the search order:

$$d_{opt}(\mathbf{q}, Sphere(\mathbf{c}, r)) = w_1 \cdot d_{min}(\mathbf{q}, Sphere(\mathbf{c}, r)) + w_2 \cdot d_{avr}(\mathbf{q}, Sphere(\mathbf{c}, r)),$$

where  $\rho$  is the density of  $Sphere(\mathbf{c}, r)$ ,  $w_1 = \frac{\rho}{\rho+1}$ , and  $w_2 = \frac{1}{\rho+1}$ . Here  $w_1 + w_2 = 1$ . We use  $\rho$  to represent the compactness of the nodes. As we can see, when the density  $\rho$  increases, the weight of  $d_{min}$  increases, so  $d_{min}$  plays an important role in determining the possibility. When the density decreases,  $d_{avr}$  is chosen for the distance between a query point and the node. The search algorithm which uses the optimized distance  $d_{opt}$  based ordering is called *optimized distance based search*(ODS). In Fig. 7,  $S_1$  has a very low density and  $S_2$  has a much higher density than  $S_1$ .  $d_{opt}(\mathbf{q}, S_1) \approx d_{avr}(\mathbf{q}, S_1)$ ,

$$d_{opt}(\mathbf{q}, S_2) \approx d_{min}(\mathbf{q}, S_2), d_{min}(\mathbf{q}, S_2) \ll d_{avr}(\mathbf{q}, S_1),$$

therefore,  $d_{opt}(\mathbf{q}, S_1) > d_{opt}(\mathbf{q}, S_2)$  and  $S_2$  will be chosen first for the nearest-neighbor search. Later, in the experiment section, we will compare these three distance metrics.

**Traversing order in the whole ClusterTree.** Our  $p$ -NN algorithm uses the depth-first search instead of the breadth-first search because it has better pruning capability. When the search goes down from the root to the leaf, the radii of the bound spheres will become smaller and the precision of bounding spheres will increase gradually. The depth-first search can quickly sink to the bottom of the ClusterTree. As

**Algorithm 4.2.2: Depth-First Search for  $p$ -NN***Input:* A query point  $\mathbf{q}$  and  $p$ *Output:* Sorted array storing the  $NN^p(\mathbf{q})$ 

1. a. Sort the entries of the *root* in decreasing order of distance  $d_{opt}$  between the entries and the query  $\mathbf{q}$ .  
b. Push all of the entries onto *Stack* in decreasing order of  $d_{opt}$ .  
c. Set  $d_p = InitialThreshold(root, \mathbf{q})$ .
2. **if** *Stack* is not empty, *currentEntry* = pop(*Stack*) and **goto** step 3; **else return**;
3. a. **if**  $d_{min}(\mathbf{q}, currentEntry) > d_p$  **then goto** step 2.  
b. *currentNode* = *currentEntry.SC*  
c. **if** the type of *currentNode* is leaf  
**then**  
**for** each data point  $\mathbf{o}$  in *currentNode*  
 $dist = d(\mathbf{o}, \mathbf{q})$ ;  
**if**  $dist < d_p$  and *bufferSize* <  $p$   
**then**  
Insert  $\mathbf{o}$  into buffer  
*bufferSize* = *bufferSize* + 1.  
**else if**  $dist < d_p$  and *bufferSize* =  $p$   
Delete the point from Buffer with the maximum distance to  $\mathbf{q}$   
Insert  $\mathbf{o}$  into Buffer  
**if** *bufferSize* =  $p$  **then**  
 $d_p$  will be assigned as the distance between the query  $\mathbf{q}$  and the  $p$ -th neighbor.  
**else if** the type of *currentNode* is internal node  
**for** each entry  $E_i$  in *currentNode*,  
**if**  $d_{max}(\mathbf{q}, E_i) \leq d_p$  and  $p \leq |SN_i|$  **then**  $d_p = d_{max}(\mathbf{q}, E_i)$ .  
Sort the entries of *currentNode* in order of  $d_{opt}$  between the entries and the query  $\mathbf{q}$ .  
Push all of the entries onto *Stack* in decreasing order of  $d_{opt}$ .
4. **goto** step 2.

Fig. 8. Algorithm 4.2.2.

soon as the first  $k$  data points are retrieved, we have more precise knowledge about the distance threshold  $d_p$ . Therefore,  $d_p$  can be adjusted optimally in the depth-first search.

Compared to the branch-and-bound algorithm proposed in [23], our depth-first search combines *downward pruning* and *upward pruning* into a simple one, named *sibling pruning*. It uses a *stack* to capture the kernel concept of the depth-first search. Because the threshold  $d_p$  can be changed optimally in the depth-first search, sibling pruning can still be as effective as the two prunings in [23]. The algorithm is listed in Fig. 8.

Algorithm 4.2.2 gives the steps of the depth-first search. *Stack* is used to save the entries in a node, where the entries correspond to the branches in the ClusterTree containing potential nearest-neighbors. By saving the entries of a node, the algorithm can prune many branches without loading the children of the node. The function *InitialThreshold* in Algorithm 4.2.2 is used to set the initial value of the threshold  $d_p$ .

## 5 DYNAMIC INSERTION

An index structure must handle the dynamic insertion of new data points, which may require that the underlying clusters be adjusted. Most clustering approaches cannot add new data points efficiently, which greatly limits the flexibility of the clustering approaches. Dynamically inserting any kind of data point into the ClusterTree should be supported because the data sets may need to be updated.

However, statically organized structures are normally more optimized than dynamically organized ones and, consequently, support queries more efficiently.

When a new data point is inserted into the cluster, it may perturb the whole structure of the index built on top of the cluster. We divide the new data points into three categories according to the degree of their perturbation on the clusters:

- *Cluster points*: are either the duplicates of or very close to some data points in a cluster within a given threshold, and can be safely inserted into the leaf nodes without changing the structure of the cluster and its subclusters.
- *Close-by points*: are the data points which are neighbors to some points in the clusters within a given threshold. They can be inserted into some clusters with slight changes in the shape of the cluster or its subclusters within the given threshold.
- *Random points*: are the data points which are either far away from all of the clusters and cannot be bounded by any bounding sphere of the ClusterTree, or might be included in the bounding spheres of the clusters at each level, but they do not have any neighboring cluster points within a given threshold. In the first case, the random points can be collected into a set, and later saved into a new ClusterTree. Thus, we only consider the second case because they affect the original ClusterTree.

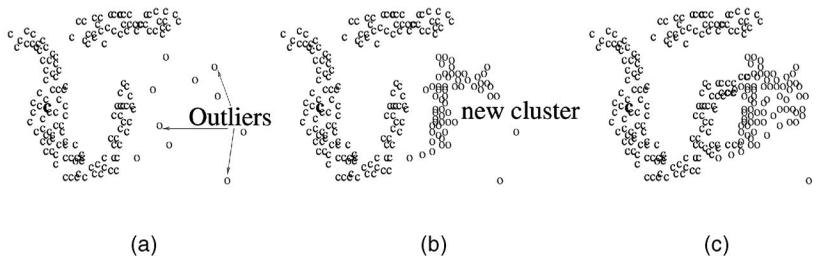


Fig. 9. (a) Clusters with random points. (b) A new cluster generated from random points. (c) The merge of the clusters.

To minimize the impact of insertions on the index structure, we design different strategies for each type of new data points. A query of the new data point on the tree can collect the neighbor information of the data point and classify it into one of the three categories. Thus, we can know what type of data points it is. The cluster points will be directly inserted into the leaf node. The close-by points will be inserted into the leaf node which contains its nearest-neighbors. The radius and the centroid of the leaf node then needs to be adjusted. If a new data point does not have any neighboring point within a given threshold, it will be classified as a random point. A single random point may be treated as noise and ignored. But when the random points accumulate, they might form some pattern of a new cluster, or they might cause the merge of some clusters. Thus, they may change the distribution pattern of the data set and outdate the original index structure. This will seriously affect the benefit of the clustered index structure and lead to the reorganization of the clusters. Fig. 9 shows how a newly generated cluster will gradually merge with an existing cluster after more insertions have been made.

The insertions of the random points are extremely hard because of the following reasons: 1) the insertion order of the random points can be arbitrary, and the cluster algorithm cannot predict where the insertions will be and 2) the lack of knowledge on what comes next may lead to a wrong decision. If the ClusterTree inserts the random points as soon as they come, the nodes will be expanded randomly, which makes the cluster representation inefficient and inaccurate.

We design a *delayed insertion* approach, which stores a single random point without changing the radius and centroid of any node in the ClusterTree and reorganizes the newly inserted data points when the amount of random points reaches a certain threshold. This reorganization can be time consuming when many nodes and data points are involved, which happens when the whole ClusterTree needs to be reorganized. The reorganization first starts with the nodes involved at the lower level (close to the leaves) and propagates to the upper levels (close to the root) of the ClusterTree until the tree is well organized.

### 5.1 Insertion of Random Points

Here is how a random point is inserted into the index structure. Starting from the root, if an entry of a node contains the data point, the corresponding child of that node will be checked to see whether its children's bounding spheres still contain the random point. If so, the child's children will be again checked recursively until the random

point cannot go down the ClusterTree. Due to the overlapping between bounding spheres, one random point might end up in several bounding spheres. The most-related bounding sphere would be selected, based on the cluster information. We define the *maximum inclusion depth* of a point  $\mathbf{o}$  in a ClusterTree as:

**Definition 3.** Let  $\mathbf{o}$  be a point and  $T$  be a ClusterTree. We define the *maximum inclusion depth* of  $\mathbf{o}$  in  $T$  as the level of the node  $V$  whose bounding sphere contains  $\mathbf{o}$ , such that for any other node  $V'$  in  $T$ , if its bounding sphere also contains  $\mathbf{o}$ , then  $level_{V'} \leq level_V$ . We denote the maximum inclusion depth of  $\mathbf{o}$  as  $L_{\mathbf{o}}$ .

If no bounding sphere  $S$  in ClusterTree  $T$  contains  $\mathbf{o}$ , then we define  $L_{\mathbf{o}} = -1$ . We will collect these types of data points for creating another ClusterTree.

Each node in the ClusterTree has a random data space to store the newly inserted random points. If the random data space is full, a new disk block will be allocated and linked to the random data space. A random point  $\mathbf{o}$  will be stored in the random data space of the node whose level is  $L_{\mathbf{o}}$ . For example, in Fig. 10, the bounding spheres of clusters  $B$  and  $C$  both include  $\mathbf{p}$ , and the subclusters  $B3$  and  $B4$  of  $B$  also include  $\mathbf{p}$ . But, none of the subclusters of  $C$  includes  $\mathbf{p}$ . Here,  $L_p = 2$ . Thus,  $B3$  is selected, and  $\mathbf{p}$  will be stored in the random data space of the node which includes  $B3$ .

### 5.2 Reorganizing Subtrees

When many random data points are inserted into the ClusterTree and the amount of random data points reach a threshold, the ClusterTree needs to be reorganized. The threshold for node <sub>$i$</sub>  is defined as:

$$\tau = \frac{\text{Number of Random Points in node}_i}{\text{Number of Random Points} + \text{Number of Cluster Points in node}_i}$$

We call  $\tau$  the *reorganizing fraction parameter*. The selection of  $\tau$  defines the dynamic property of the ClusterTree. If  $\tau$  is small, the ClusterTree will be updated in response to the insertion of a small number of random data points. Let  $k$  be the number of subclusters for each node, then based on experiments, we observe that when the percentage of the random points reaches  $\frac{1}{k} \times 100\%$  of all of the cluster data points belonging to the current internal node, the original  $k$  subclusters of the node may not effectively represent the data points in this node. Thus, we set  $\tau = \frac{1}{k} \times 100\%$  as the threshold to reorganize the current node.

The reorganization of an internal node is also based on the  $k$ -medoid method. Both the cluster and random data points in the node are treated as a subcluster and the  $k$ -medoid method

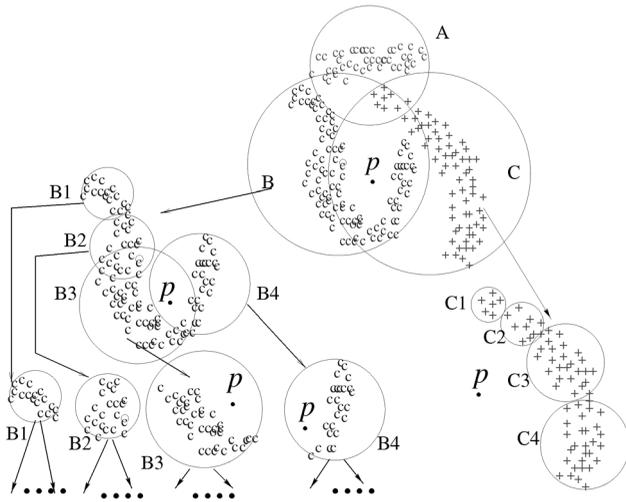


Fig. 10. The selection of the most relevant (sub) clusters.

will be applied to split the subcluster recursively until a new subtree is established. When the new subtree is built, the height of the ClusterTree may increase. We design a *force reinsertion* to keep the ClusterTree balanced. Fig. 11 shows the procedure of the force reinsertion. When the height of the newly reorganized subtree is increased, it will move up the siblings of its original parents. The child node with the smallest number of data points is removed from the children list of node  $A$ , and its data points are reinserted back to  $A$ . When  $\tau$  for  $A$  is reached, it should also be reorganized, and optimization will be propagated to the upper levels until the ClusterTree does not have any nodes where the threshold  $\tau$  has been reached.

The process on the leaf nodes will be slightly different from the process on the internal nodes. When a disk block cannot store all of the data points belonging to leaf node  $L$ , a 2-medoid splitting method will be applied on the leaf node. The two sibling leaf nodes  $L_1$  and  $L_2$  are generated from this splitting. If  $L$ 's parent node is not full,  $L_1$  and  $L_2$  will be inserted into the parent node. Otherwise, we apply the similar technique on the parent node, as described in the beginning of Section 5.2.

Our approach is not sensitive to the insertion order of the new data points, and can accommodate a large number of new data points and new clusters. By inserting the new data points into the fairly close node, the index structure can coarsely partition the new data points, which greatly benefits the (re)clustering on the new data points later on. When a query checks a node  $V$  with random points, all of its children need to be searched first. If the search range  $Sphere(q, d_p)$  is included in one of  $V$ 's children, we can prune the random points because none of the random points will be a candidate. Otherwise, we need to linearly search the random points to determine whether they can be candidates.

## 6 EXPERIMENTS

We have conducted comprehensive experiments to evaluate the performance of the ClusterTree. We also compared the ClusterTree with three other index strategies: DBIN [4], the

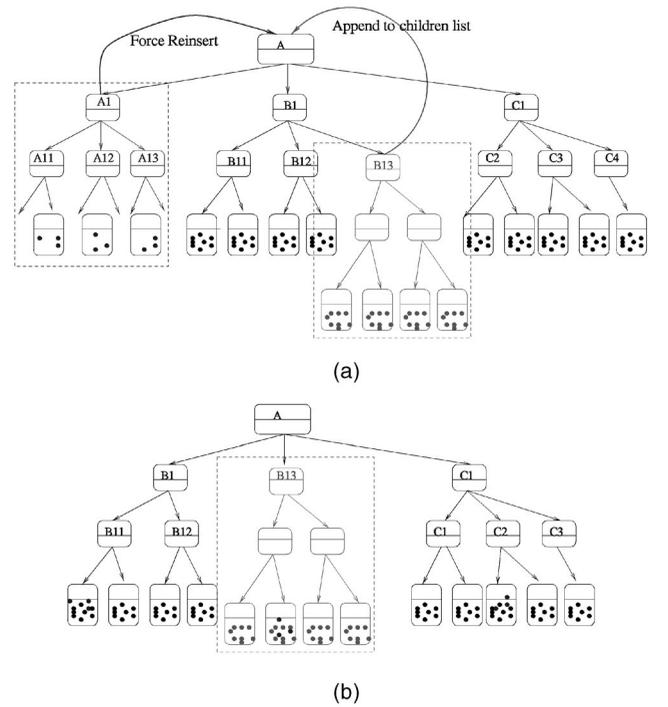


Fig. 11. (a) Before force reinsertions. (b) After force reinsertions.

Pyramid-Tree [5] and SR-Tree [17]. Our experiments were performed on a Sun ULTRA 60 with 512 Megabytes memory. The experimental results presented below demonstrate the originality of the ClusterTree.

### 6.1 Data Sets for Experiments

**Synthetic data set.** We first designed our own synthetic data set generator for performing experiments. The data generator allows control over the structure and the size of the data sets. The user can specify the number of data points, number of clusters, number of dimensions, number of points in each cluster, range of values for each attribute, and the underlying probability distribution of the data points in each cluster. The generator also allows the user to specify the number of noise data points. The noise data points are distributed according to a random distribution across the data space. The clusters are either hyperrectangles (a uniform distribution) or spheres (following  $d$ -dimensional and independent normal distributions). The density of clusters is much higher than the average density of the space around the cluster. The generation of hyperrectangle clusters follows closely the method described in [32]. For the spherically shaped clusters, the user has the flexibility to choose the variance in each dimension. Due to the properties of normal distributions, the maximum distance between a point in the cluster and the center is unbounded. So, a data point that belongs to cluster  $A$  may be very close to some data points in another cluster  $B$ . By creating overlaps among the clusters, we increase the distribution complexity of the synthetic data sets.

We generated 20 different synthetic data sets to evaluate different aspects of our approach. The data sets have sizes ranging from 100,000 to 1,000,000 data points with five to 50 dimensions each. We added some random data points (noise) to the data sets that total about 10 percent of the data

TABLE 1  
Test Data Sets Used in the Experiments

Dataset	Number of Dimensions	Number of Data Points	Number of Outliers	Number of Clusters	Comments
1	12	100,000	10,000	9	5 hypercube and 4 spherical clusters
2	12	200,000	20,000	9	5 hypercube and 4 spherical clusters
3	12	300,000	30,000	9	5 hypercube and 4 spherical clusters
4	12	400,000	40,000	9	5 hypercube and 4 spherical clusters
5	12	500,000	50,000	9	5 hypercube and 4 spherical clusters
6	12	600,000	60,000	9	5 hypercube and 4 spherical clusters
7	12	700,000	70,000	9	5 hypercube and 4 spherical clusters
8	12	800,000	80,000	9	5 hypercube and 4 spherical clusters
9	12	900,000	90,000	9	5 hypercube and 4 spherical clusters
10	12	1000,000	100,000	9	5 hypercube and 4 spherical clusters
11	5	100,000	10,000	9	5 hypercube and 4 spherical clusters
12	10	100,000	10,000	9	5 hypercube and 4 spherical clusters
⋮	⋮	⋮	⋮	⋮	⋮
20	50	100,000	10,000	9	5 hypercube and 4 spherical clusters

points. Table 1 summarizes the information about the data sets used in our experiments.

**Real data set.** We also conducted our experiments on the GIS data sets generated from the forest-coverage data set [2], which consists of 581K records. Each data record is a 54-dimensional data point, which includes elevation, aspect, slope, etc. The data set was divided into seven clusters by the US Forest Services. We created the data sets with varying dimensions and numbers of data points by projecting and selecting data points from the forest coverage data set. The sizes of the created data sets are 50,000, 100,000, 150,000, ..., 550,000, and 581,000. The dimensions are 5, 10, ..., 50, and 54.

## 6.2 Building the ClusterTree

We measured the performance of constructing the ClusterTree and the SR-Tree for the real data sets under the same conditions. Figs. 12a and 13a show the scalability as the size of the data sets increases from 10,000 to 581,012 data points. The average time complexity of constructing a ClusterTree is  $O(k \cdot N \cdot \log_k N)$ . When  $k = 10$  and  $N < 600,000$ , the time complexity degrades to be a linear function. As shown in Fig. 12a, the running time scales linearly with the size of the

data sets. We follow the suggestion of the SR-Tree and Pyramid Tree, the maximum disk block size is set to 4,096 bytes. The size of an entry in an internal node is  $4 \times d + 20$ , where  $d$  is the number of dimensions. An example of 50 dimensions leads to an effective capacity of 18.6 entries per page for internal nodes, and capacity of 20.48 data points per page for leaf nodes. When the dimensions become smaller,  $k$  becomes bigger. This increases the time complexity for constructing a ClusterTree, generating lot of trivial subclusters. Therefore, when dimension is smaller than 25, we use 2,048 as block size. When dimension is smaller than 10, we choose 1,024 as block size.

Fig. 12a also shows the comparison between the ClusterTree and the SR-Tree, and the speed-up factor of the ClusterTree over the SR-Tree with respect to the CPU and disk I/O time ranges between 2.5 and 8.1. We can also see the increasing scale of building the ClusterTree is much smaller than that of SR-Tree. The reason is that the ClusterTree only needs to calculate the bounding sphere, while the SR-Tree must calculate the minimum bounding hyperrectangles and spheres.

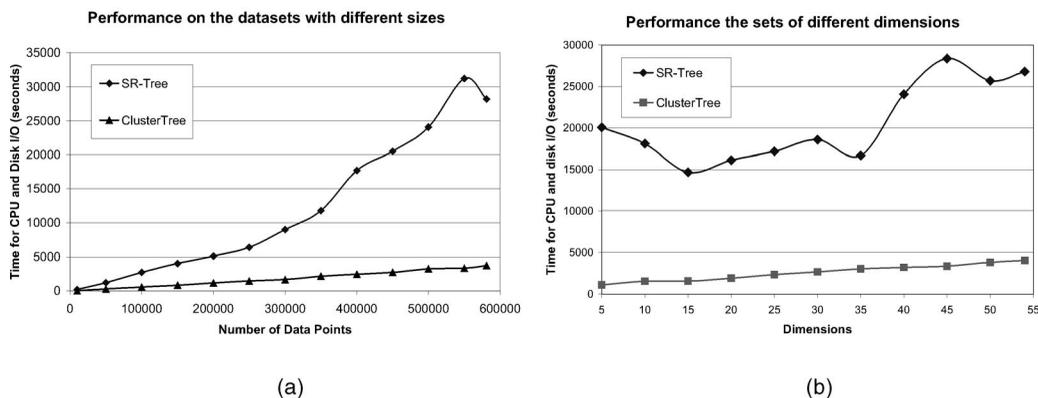


Fig. 12. Performance of constructing the ClusterTree and SR-Tree for different data sizes and dimensions. (a) 40-dimensional data sets. (b) 500,000 data points under different dimensions.

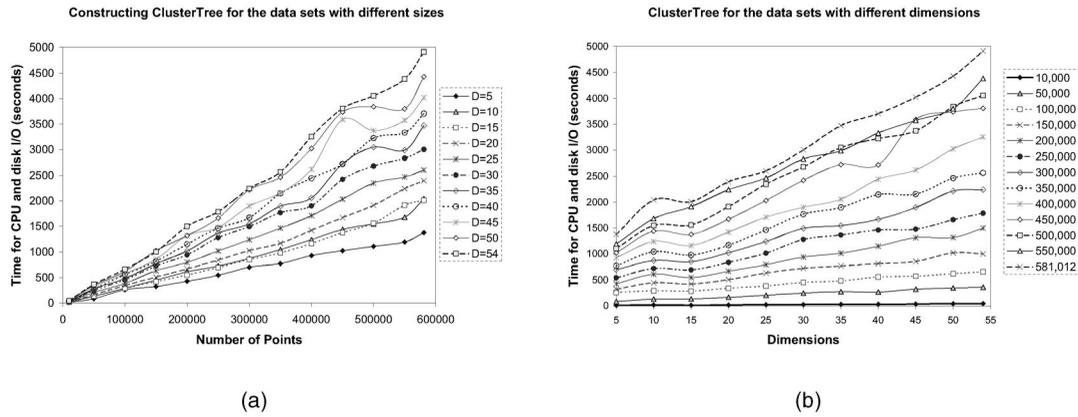


Fig. 13. Performance of constructing the ClusterTree for different data sizes and dimensions.

Fig. 12b shows the scalability as the dimensions of the data sets increase from five to 54. The data sets in Fig. 12b each have 500,000 data points. The ClusterTree exhibits linear behavior with respect to dimensions. Fig. 12b also shows the comparison between the ClusterTree and the SR-Tree with different dimensionalities, and the speed-up factor with respect to the CPU and disk I/O time ranges between four and 18. The construction time of the SR-Tree is highly affected by dimensions, while the ClusterTree is not impacted by dimensions at all. The linear increment comes from the distance computation for increasing dimensions. Fig. 13 gives the overall construction time of the ClusterTree under different dimensions and data set sizes. Each curve in Fig. 13a represents the CPU and disk I/O time of building a ClusterTree for 13 data sets with different sizes and the same number of dimensions  $D$ , labeled by the number of dimensions  $D$ . Each curve in Fig. 13b represents the CPU and disk I/O time of building a ClusterTree for 11 data sets with different dimensions and the same data set size, labeled by the data set size. They both show that the construction time is linear with respect to the dimensions and the data set size.

### 6.3 Performance on Insertion

We randomly generated up to 1000 data points and inserted them into the ClusterTree and SR-Tree. These data points include cluster points, close-by points, and random points. Figs. 14a and 15a show the average CPU and disk access

time for inserting a new data point into the two index structures with different sizes and dimensionalities. The synthetic data sets were used in this experiment. The ClusterTree is faster in insertion because the centroid-based algorithm in the ClusterTree only calculates the distance between the new data points and the centroid. Fig. 14b shows that the total number of nodes accessed is linear to the number of data points when random data points are inserted into the ClusterTree. Fig. 14 also shows that inserting a new data point into the ClusterTree costs no more than 0.02 seconds with 40 node accesses. The dynamic insertion into a ClusterTree does not have a significant impact on the whole index structure. It does not slow down the on-line response time for queries. Fig. 15b shows that the number of node accessed for inserting data points increases slightly as the dimensions increase because the number of nodes in the ClusterTree is mainly determined by the size of the data set, not by the dimensions. This leads to no significant increase in CPU and disk I/O time as shown in Fig. 15a. When the same data point is inserted into the two index structures, the number of nodes accessed in the ClusterTree is very close to that of the SR-Tree. But the ClusterTree is faster than the SR-Tree. This is because the ClusterTree has smaller nodes than the SR-Tree, and it takes much less time for the ClusterTree to process a node.

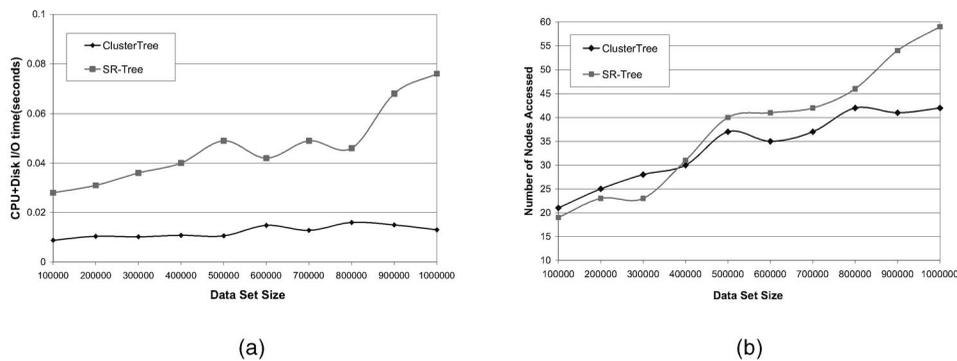


Fig. 14. Insertion cost of ClusterTrees and SR-Trees of data sets with different sizes.

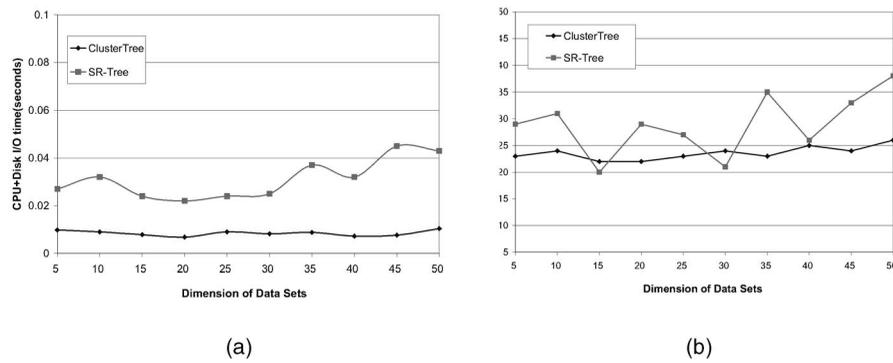


Fig. 15. Insertion cost of ClusterTrees and SR-Trees of data sets with different dimensions.

#### 6.4 Performance on Query

We will now evaluate the performance of the nearest-neighbor search on the ClusterTree. We will not show experiments for range queries because the nearest-neighbor search is essentially a range query with a consistently changing search scope. The effectiveness and efficiency of the nearest-neighbor search is determined by the index structure and search algorithms, which can be measured by the following parameters:

- Accuracy of the retrieval results.
- Retrieval percentage: the percentage of data points searched to obtain the nearest-neighbors versus all of the data points in a data set.
- Speed-up factors of CPU time and the time spent on disk I/O when the ClusterTree is compared with the related approaches.
- The number of nodes (internal nodes and leaves) accessed in the process of searching the nearest-neighbors.

The size and dimensions of the data sets as well as the number of nearest-neighbors required will affect the values of the above parameters. We aim to demonstrate the power of the ClusterTree for the nearest-neighbor search by comparing with the other existing index structures (SR-Tree, Pyramid-Tree). We will also show the efficiency of the proposed nearest-neighbor search algorithms for the ClusterTree by comparing our results with the results of an optimal search which can achieve the best performance. Such an optimal search is described as follows.

Given a query of a  $p$ -nearest-neighbor search, the  $p$ th nearest distance to the query is calculated in advance, then this  $p$ -nearest-neighbor search can be transformed into a range query which uses the  $p$ th distance as the search range.

Obviously this range query can guarantee the best performance for the  $p$ -nearest-neighbor search. We call this range query algorithm an optimal search algorithm (OPT). The selection of a query point affects the performance evaluation. If a data point's nearest-neighbors are within a few nodes, the query performance will be much better than a data point whose nearest-neighbors are randomly distributed in many data nodes. We picked 50 Cluster points, 50 Close-by point, and 50 Random data points. Half of the 50 Cluster points are picked randomly from the data set itself, the other half of the cluster points are randomly generated based on the scope of the leaf nodes. The size of the evaluation set also affects the average performance if it is small. When the size of the evaluation set is more than 100, the average performance does not change significantly. Therefore, both selection of the evaluation set and its size are important to show the average performance of queries accurately. The same query set is applied to the other two index structures.

##### 6.4.1 Accuracy of Retrieval and Fraction of Data Searched

Our approach can accurately perform the  $p$ -nearest-neighbor search. For a given query, it only searches within the most related clusters for the nearest-neighbors. Tables 2 and 3 show the comparison between our approach and DBIN. From the tables, the ClusterTree achieved 100 percent accuracy on searching the nearest-neighbors. DBIN is based on a probabilistic approach, and also achieves very high accuracy. The fraction of data points that are searched in our approach is much lower than that of DBIN because the ClusterTree decomposes a cluster into several subclusters. Instead of searching the entire cluster, it searches the related subclusters. DBIN only performs queries at the cluster level.

TABLE 2  
Accuracy and Fraction Searched for 12-Dimensional  
Data Set with 1,000,000 Data Points

		Number Of nearest neighbors			
		2	5	10	50
DBIN	Accuracy	99.0%	100%	100%	97.1%
	Fraction	12.5%	13.9%	14.2%	16.6%
ClusterTree	Accuracy	100.0%	100%	100%	100%
	Fraction	2.58%	3.53%	4.2%	5.79%

TABLE 3  
Accuracy and Fraction Searched for 30-Dimensional  
Real Data Set with 100,000 Data Points

		Number Of nearest neighbors			
		2	5	10	50
DBIN	Accuracy	94.7%	90.0%	85.0%	78.6%
	Fraction	16.8%	17.2%	17.4%	17.8%
ClusterTree	Accuracy	100.0%	100%	100%	100%
	Fraction	3.58%	5.024%	6.10%	9.04%

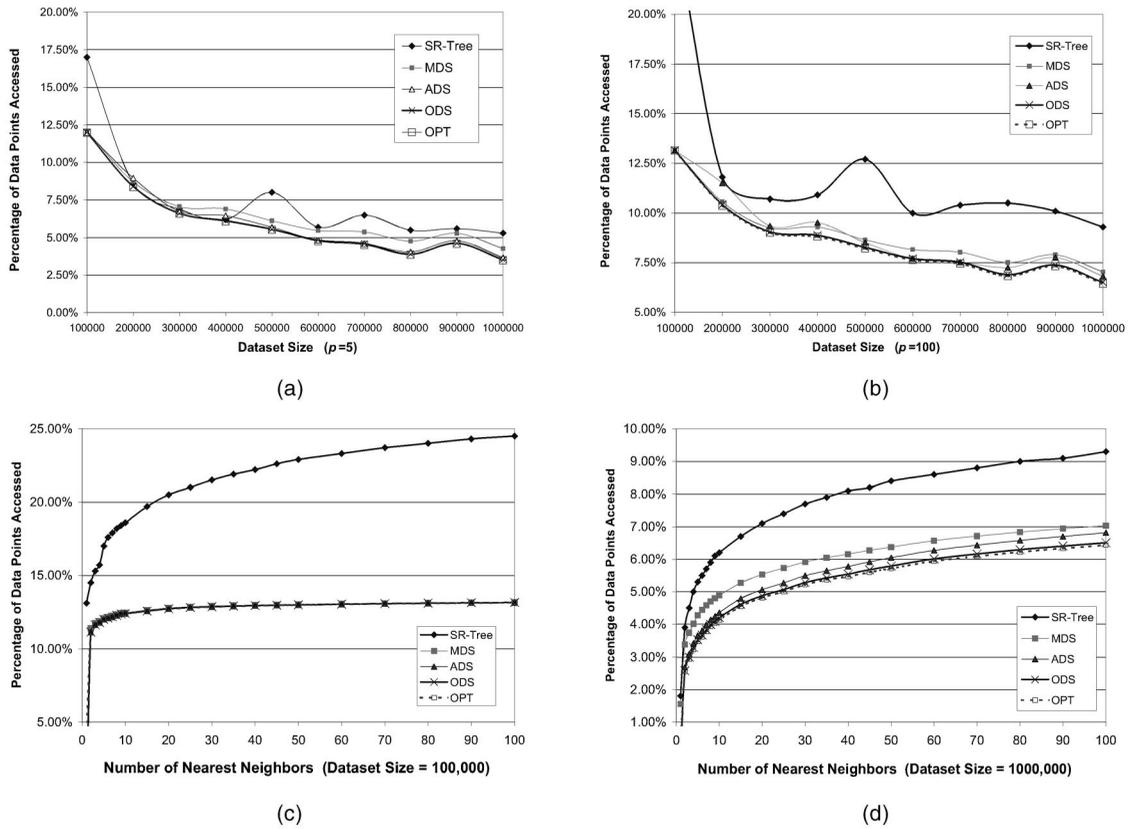


Fig. 16. Retrieval percentage of ClusterTrees and SR-Tree using data sets with the same dimensions.

Fig. 16 shows the detailed comparison between the ClusterTree and SR-Tree on 10-dimensional data sets with different sizes. The synthetic data sets used are Data Sets 1, 2, ..., 10. Figs. 16a and 16b show the five and 100-nearest-neighbor searches where the size of the data sets goes from 100,000 to 1,000,000. The retrieval percentage for the 5-nearest-neighbor search decreases from 12.5 percent to 3 percent, and the retrieval percentage for the 100-nearest-neighbor search decreases from 13 percent to 6 percent. When the size of the data sets is small, the cluster is very sparse, and the search range might check most of the points. As the size of the data set increases, the clusters become more dense, and the neighboring data points are clustered into the same cluster or subcluster. Therefore, the search region can be reduced. For small data sets, we can afford to search a large part of the data set, and the response time will still be very low. But for a large data set, the retrieval percentage is crucial to the performance. The experiments show that the retrieval percentage decreases as the size of the data sets becomes larger. This can reduce the cost of the CPU time and disk accesses. Fig. 16a shows that ODS has a uniformly lower retrieval percentage than MDS, ADS, and the SR-Tree. The SR-Tree has a relatively high retrieval percentage when the size is 100,000, but its retrieval percentage decreases very fast, and it outperforms MDS and ADS slightly when the size is 200,000 ~ 400,000. MDS and ADS perform better than the SR-Tree when the size of the data sets is bigger than 400,000. The experiments demonstrate that ODS is superior to MDS and ADS. Fig. 16b shows these three search algorithms have better

performance than the SR-Tree. Fig. 16c shows the retrieval percentage for the  $p$ -nearest-neighbor search in a data set with 100,000 points, where  $p = 1, 2, \dots, 100$ . The retrieval percentages for these four search strategies are very close. Fig. 16d shows the results for a data set with 1,000,000 points. The ClusterTree outperforms the SR-Tree for any given value  $p$  in both data sets.

Fig. 17 shows the performance for the nearest-neighbor search in the GIS forest coverage data sets, where the number of dimensions of these data sets is 10. Each curve in Fig. 17a shows the retrieval percentage of the data sets with different number of queries and a fixed data set size, and each curve is labeled by this data set size. When the data set is small, such as 50,000 data points, the retrieval percentage is close to 20 percent. When the data set is large, the retrieval percentage is only about 5 percent. The data sets with 550,000 data points have the lowest retrieval percentage. Fig. 17 also shows that as the size of the data sets increases, the retrieval percentage decreases. Fig. 17b gives an example of 100-NN nearest-neighbor search in 10-dimensional data sets with 50,000 ~ 550,000 data points. The retrieval percentage is 17 percent for the data set with 50,000 data points, and decreases to 6 percent when the size of the data set is 550,000.

Fig. 18a shows the retrieval percentage of the 5-nearest-neighbor search when the dimension is 5, 10, ... 50 and the size of the data sets is 100,000. Data sets 11, 12, ..., 20 used in this experiment are synthetic. The SR-Tree has a retrieval percentage of 2 percent when the dimension is 5, it reaches a percentage of 50 percent when the dimension is 40, while the ClusterTree keeps the retrieval percentage lower than 20

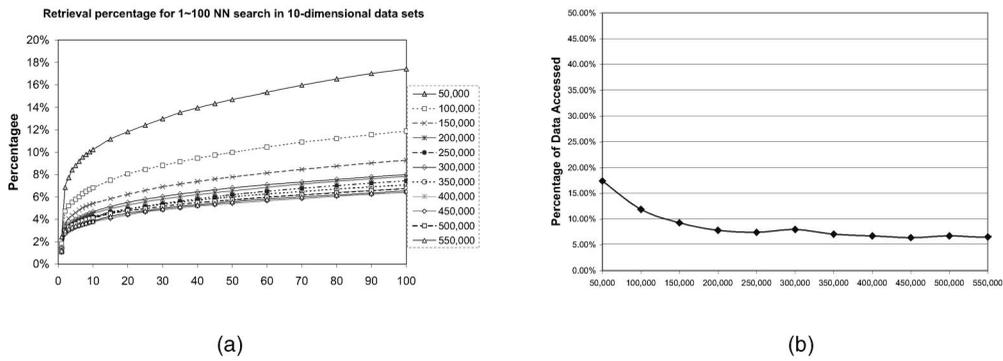


Fig. 17. Retrieval percentage of ClusterTrees for different data sizes and dimensions. (a) Number of queries. (b) Number of data points.

percent. The MDS performs slight better than ADS, and ODS achieves the best performance among these three search methods. ODS almost reaches the performance of the optimal search algorithm (OPT).

Fig. 18b shows the retrieval percentage of the  $p$ -nearest-neighbor search in a 5-dimensional data set with  $p = 1, 2, \dots, 100$ . Table 4 lists the retrieval percentages for the data sets with dimensions of 10 and 40. When the data set is 5-dimensional, ADS performs better than MDS (shown in Fig. 18b), but MDS shows better performance when the data set is 10-dimensional (shown in Table 4). In both data sets, ODS has a lower retrieval percentage than MDS and ADS. As discussed in Section 4.2.2, both  $d_{min}$  and  $d_{avr}$  do not take the data distribution into consideration. Therefore, for different data distributions, neither of them can achieve better performance under all circumstances. ODS takes advantage of the data distribution, therefore, it achieves better performance than ADS and MDS. ODS has almost the same performance as OPT, as shown in Fig. 18 and Table 4. As Table 4 shows, when the data set is 40 dimensional, the SR-Tree searches half of the data set to get the nearest-neighbors. This result derives from the distribution of the distances between the data points within the data set. In [17], [7], it was shown that the minimum distance between any two data points grows drastically as dimensions increase, and the ratio between the minimum distance and maximum distance is more than 60 percent when the number of dimensions is 40. That is, the variation

of distance decreases as the dimensions increase. This also leads to the result that the ratio between  $d_{min}$  and  $d_{avr}$  is close to 1. Therefore, MDS, ADS, ODS and OPT have similar performance when the dimension is 40, as shown in Table 4. With the introduction of clusters and subclusters, the retrieval percentage of the ClusterTree is 30 percent lower than that of the SR-Tree.

#### 6.4.2 CPU and Disk I/O Time

**Performance of nearest-neighbor search on various data set sizes.** In this experiment, we measure the performance behavior with the synthetic data sets of the different sizes in term of CPU time and the time spent on disk I/O. The 10-dimensional synthetic data sets used are data sets 1, 2, ..., 10. Their sizes range from 100,000 to 1,000,000. Previous experiments show that ODS has better performance than ADS and MDS. In this experiment, we only show the results of ODS for the ClusterTree. Fig. 19 shows the total time elapsed when the 5-NN, 30-NN and 100-NN are searched in the data set with varying sizes. As the size of a data set increases, the number of the nodes in the ClusterTree also increases. A search algorithm needs to access more nodes to obtain the nearest-neighbors, which slightly increases the CPU and disk I/O time spent on the search. As shown in Fig. 19, the increment of the total time is very small when the ClusterTree is compared with the SR-Tree and Pyramid-Tree, also the ClusterTree spends less time than the SR-Tree and Pyramid-Tree to perform a nearest-neighbor search.

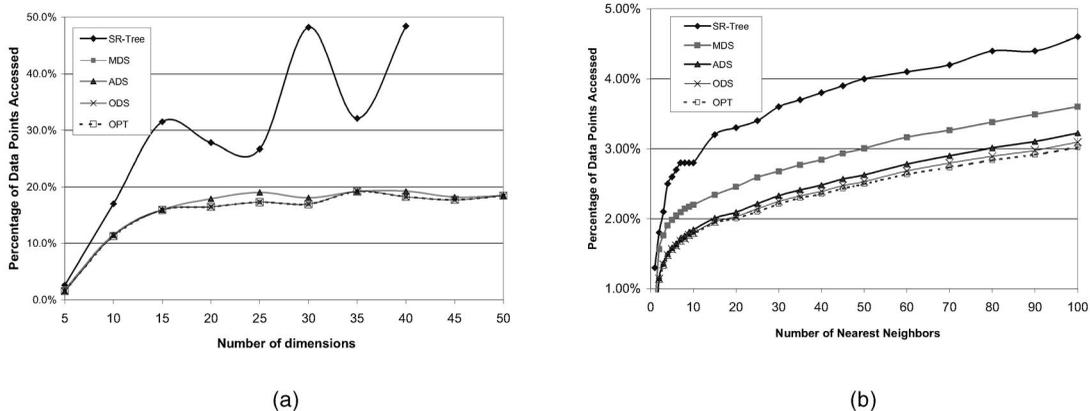


Fig. 18. Retrieval percentage of ClusterTree and SR-Tree for different  $p$  values.

TABLE 4  
Retrieval Percentage of ClutsterTree and SR-Tree for Different  $p$  Values

Value of $p$	Dimension = 10					Dimension = 40				
	SR-Tree	MDS	ADS	ODS	OPT	SR-Tree	MDS	ADS	ODS	OPT
2	16.10%	10.61%	10.35%	10.20%	10.18%	48.40%	17.60%	19.08%	17.59%	17.59%
5	17.00%	11.61%	11.48%	11.30%	11.27%	48.40%	18.25%	19.24%	18.25%	18.25%
10	18.00%	12.10%	12.10%	11.87%	11.84%	48.50%	18.27%	19.26%	18.26%	18.26%
15	18.50%	12.36%	12.42%	12.15%	12.13%	48.50%	18.77%	19.27%	18.77%	18.77%
20	18.80%	12.52%	12.60%	12.34%	12.32%	48.50%	18.86%	19.28%	18.86%	18.86%
25	19.00%	12.64%	12.74%	12.45%	12.42%	48.50%	19.25%	19.28%	19.25%	19.25%
30	19.20%	12.74%	12.86%	12.57%	12.54%	48.60%	19.27%	19.29%	19.27%	19.27%
35	19.30%	12.83%	12.98%	12.68%	12.64%	48.60%	19.28%	19.29%	19.28%	19.28%
40	19.50%	12.90%	13.09%	12.76%	12.74%	48.60%	19.29%	19.29%	19.29%	19.29%
45	19.60%	12.96%	13.15%	12.82%	12.80%	48.60%	19.29%	19.29%	19.29%	19.29%
50	19.60%	13.02%	13.22%	12.88%	12.85%	48.60%	19.29%	19.29%	19.29%	19.29%
60	19.70%	13.09%	13.32%	12.96%	12.94%	48.60%	19.30%	19.30%	19.30%	19.29%
70	19.70%	13.16%	13.41%	13.03%	13.02%	48.60%	19.30%	19.30%	19.30%	19.30%
80	19.80%	13.22%	13.47%	13.10%	13.07%	48.70%	19.30%	19.30%	19.30%	19.30%
90	19.80%	13.26%	13.52%	13.14%	13.12%	48.70%	19.30%	19.30%	19.30%	19.30%
100	20.00%	13.30%	13.57%	13.18%	13.16%	48.70%	19.30%	19.30%	19.30%	19.30%

Based on the experiment results in Fig. 19a, the speed-up factor of the ClusterTree over the SR-Tree starts at 11.4, and slowly decreases to 3.8 as the data set increases. The average speed-up factor is 4.6. The speed-up factor of the ClusterTree to the Pyramid-Tree starts at 1.6, and slowly increases to 2.29 as the size of the data sets increases to 1,000,000. Previous experiments show that the ClusterTree has very low retrieval percentage, therefore, the time spent on reading the data points into main memory from disk and checking whether the data points are the nearest-neighbors is relatively low when compared with SR-Tree.

Table 5 shows the running time of the  $p$  nearest-neighbor search on three data sets with sizes 100,000, 500,000, and 1,000,000, which includes CPU and disk I/O time. All time values are represented in seconds. When the data set is 100,000, the speed-up factor of the ClusterTree over the SR-Tree starts at 9.33 for  $p = 2$ , and increases to 43.4 for  $p = 100$ .

The speed-up factor of the ClusterTree over the Pyramid-Tree starts at 1.5, and increases to 1.9. This experiment shows that the ClusterTree outperforms the competitive structures (SR-Tree and Pyramid Tree) for very small queries as well as large queries. For a large query, the ClusterTree performs even better.

**Performance of nearest-neighbor search on different dimensions.** In this experiment, we tested the effect of *Curse of Dimensionality* on the performance of the index structures.

The synthetic data sets used here have 5, 10, ..., and 50 dimensions. Figs. 20a and 20b show the results of the 5-NN and 100-NN for all of the dimensions. Fig. 20b only figshows part of the result for the SR-Tree. As shown in this figure, the running time of the ClusterTree on a query scales linearly with the dimensions of the data sets. The scale is lower than both SR-Tree and Pyramid-Tree. We observed in Fig. 20a that the speed-up factor of the ClusterTree over the SR-Tree starts at 2.62, and reaches its highest value of 22.03 when the dimension is 40 and  $p = 5$ . Fig. 20b shows that the speed-up factor of the ClusterTree over the SR-Tree starts at 8.16, and reaches its highest value of 87.20 when the dimension is 40 and  $p = 100$ . The speed-up factor of the ClusterTree over the Pyramid-Tree ranges between 1.2 and 3.1, and it scales up as the dimensions increase. That is, the ClusterTree is even faster than the others when the dimensions go higher. The performance curves for the SR-Tree and Pyramid-Tree are not monotonic because the query points are randomly picked and their neighboring information related to the data set can significantly impact the query efficiency. In contrast, the response time of the ClusterTree monotonically increases because the percentage of the data points searched increases with dimensions, as shown in Fig. 18a.

Fig. 20c shows the total elapsed time of the three index structures when the number of dimensions is five. We observed that the response time of the SR-Tree linearly

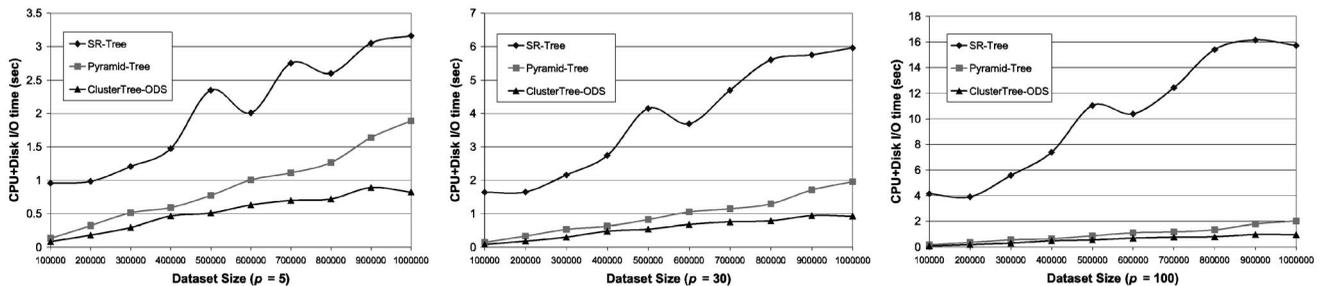


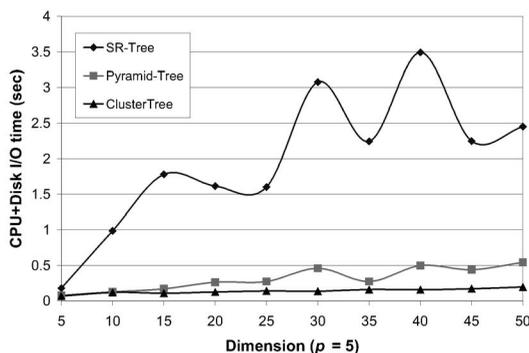
Fig. 19. Performance of SR-Tree, ClusterTree, and Pyramid-Tree on different data set sizes.

TABLE 5  
CPU + Disk I/O Time of ClusterTree, Pyramid-Tree, and SR-Tree for the  $p$ -Nearest-Neighbor Search

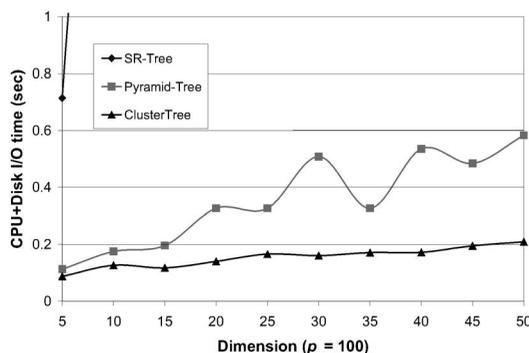
Value of $p$	Dataset size = 100,000			Dataset size = 500,000			Dataset size = 1000,000		
	SR-Tree	Pyramid	ClusterTree	SR-Tree	Pyramid	ClusterTree	SR-Tree	Pyramid	ClusterTree
2	0.784	0.129	0.084	1.95	0.764	0.4865	2.52	1.83	0.7405
5	0.957	0.134	0.0835	2.35	0.772	0.5095	3.16	1.889	0.8215
10	1.111	0.135	0.0875	2.8	0.783	0.5355	3.8	1.924	0.874
15	1.252	0.141	0.0865	3.1	0.791	0.5335	4.36	1.944	0.888
20	1.384	0.144	0.0855	3.45	0.802	0.547	4.88	1.976	0.9035
25	1.512	0.145	0.0845	3.8	0.82	0.5435	5.44	1.967	0.912
30	1.643	0.146	0.0865	4.15	0.823	0.5425	5.96	1.956	0.9235
35	1.781	0.148	0.087	4.55	0.834	0.5445	6.48	1.963	0.9335
40	1.95	0.149	0.0845	4.9	0.836	0.543	7.08	1.959	0.9205
45	2.1	0.15	0.087	5.3	0.838	0.5475	7.6	1.962	0.937
50	2.25	0.172	0.0865	5.75	0.962	0.5515	8.24	2.233	0.9485
60	2.6	0.157	0.085	6.6	0.861	0.5495	9.48	1.993	0.959
70	2.9	0.159	0.0875	7.55	0.861	0.5555	10.84	1.991	0.9715
80	3.3	0.162	0.09	8.65	0.869	0.5575	12.36	1.999	0.92
90	3.75	0.165	0.0885	9.9	0.869	0.563	13.96	2.024	0.9685
100	4.15	0.17	0.0895	11.05	0.879	0.5625	15.72	2.017	0.9735

increases with the query size, while the time of the ClusterTree increases only slightly. Fig. 20d shows the total elapsed time of the ClusterTree and the Pyramid-Tree when the number of dimensions is 50, and we can see that the speed-up factor here is close to 3. It also shows that the time does not increase even when the size of the query increases. As discussed in [8], the effect of high dimensions is to extend query regions. We can define a unit data cube

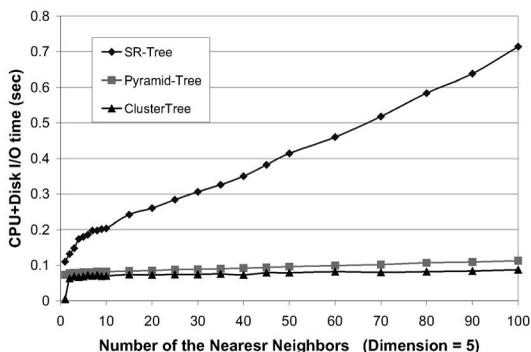
$R[0.0 \dots 1.0]^D$  as: 1)  $D$  is the number of dimensions with  $D \geq 20$  and 2) each component of the data points ranges from 0.0 to 1.0. For any data set in this cube, the distance between the query point and its nearest-neighbor ( $p = 1$ ) reaches a value of 0.5, i.e. the nearest-neighbor sphere has the same diameter as the complete data space. This means that the value of  $p$  has a minor influence on the search scope. Therefore, the total time does not change signifi-



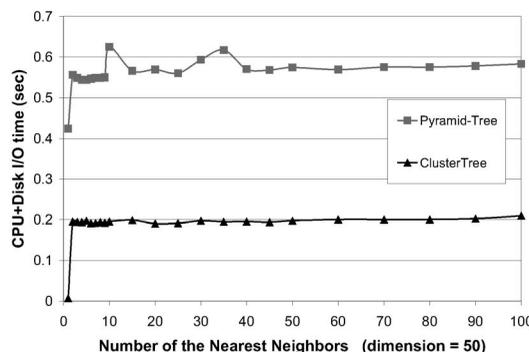
(a)



(b)



(c)



(d)

Fig. 20. Performance of SR-Tree, ClusterTree, and Pyramid-Tree on different dimensions.

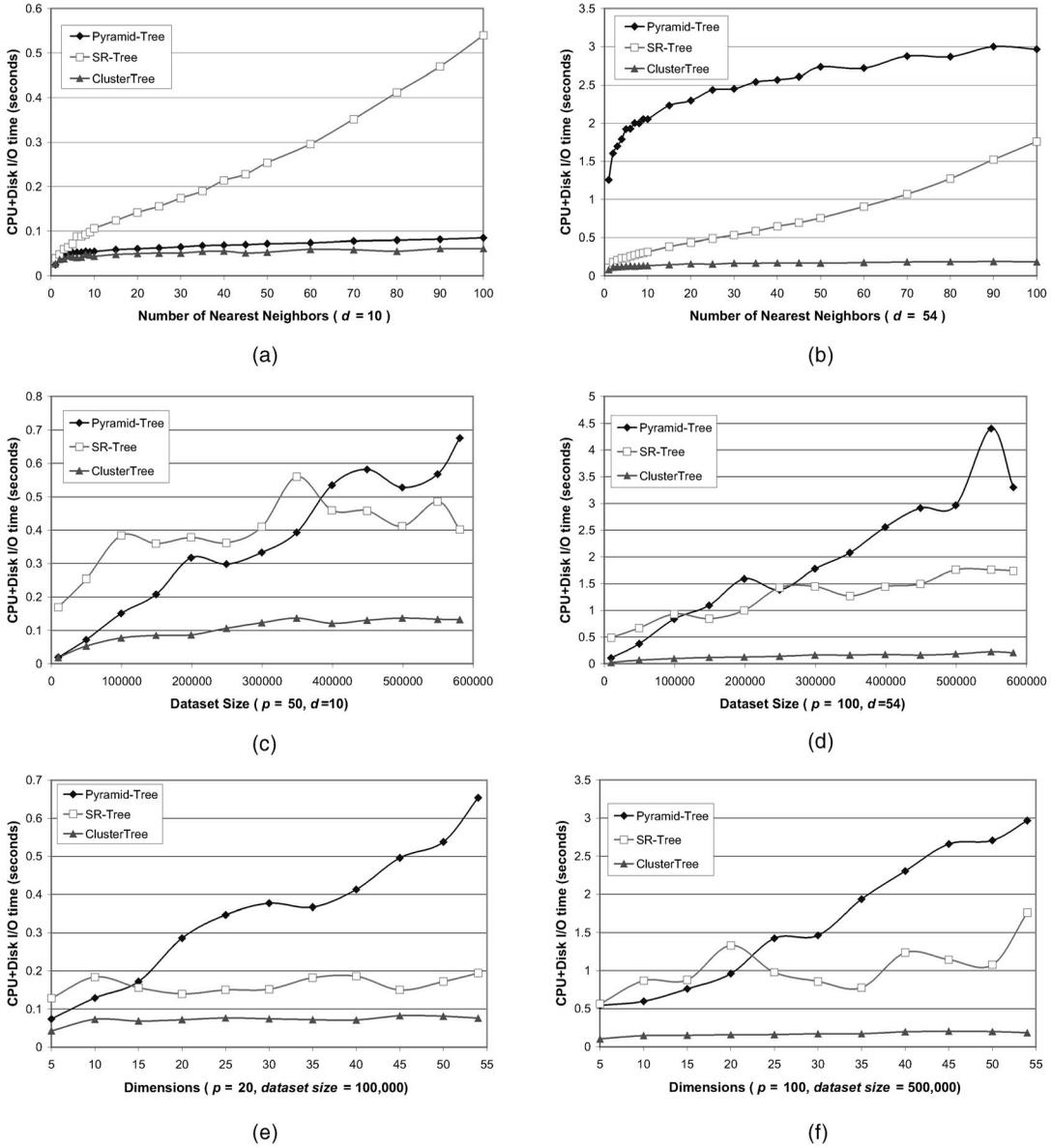


Fig. 21. Performance of Pyramid-Tree, SR-Tree, and ClusterTree on real data sets.

cantly as shown in Fig. 20d. This experiment shows that the ClusterTree runs faster than the Pyramid-Tree and SR-Tree when the dimensions go higher. The major reason is that the ClusterTree considers the clusters in the query, and groups the neighboring points within the same (sub) cluster, then the search region only includes a few nodes close to the query point. Therefore, it demonstrates the effectiveness of including data clustering information in the index structure design.

**Performance Evaluation Using Real Data sets.** In this series of experiments, we use the forest coverage data sets to demonstrate the practical applicability of the ClusterTree. We also compare it with the Pyramid-Tree and the SR-Tree over these real data sets. We vary the number of nearest-neighbors, the size of the data sets and the dimensions, and measure the total query time (CPU + disk I/O time).

Fig. 21a shows the query time for the  $p$ -nearest-neighbor search when the size of the data set is 50,000 and the

number of dimensions is 10. In this experiment, the Pyramid-Tree shows better performance than the SR-Tree, and the ClusterTree shows better performance than the Pyramid-Tree. Fig. 21b shows the results when the size of the data set is 500,000 and dimension is 54. Here, the SR-Tree shows better performance than the Pyramid-Tree. The real data sets are skewed data. The authors of the Pyramid-Tree indicated that the Pyramid-Tree could be affected by the skewness of the data sets. As the number of dimensions increase, the data sets tend to be more skewed. Therefore, the Pyramid-Tree is affected by the skewness of the data sets, and the SR-Tree has better performance than the Pyramid-Tree.

Figs. 21c and 21d show the query time when we fix  $p$  and the number of dimensions  $D$ , and vary the size of the data sets. Fig. 21c shows the performance when  $D = 10$ , where Fig. 21d shows the performance when  $D = 54$ . We observed that the running time of the Pyramid-Tree does not

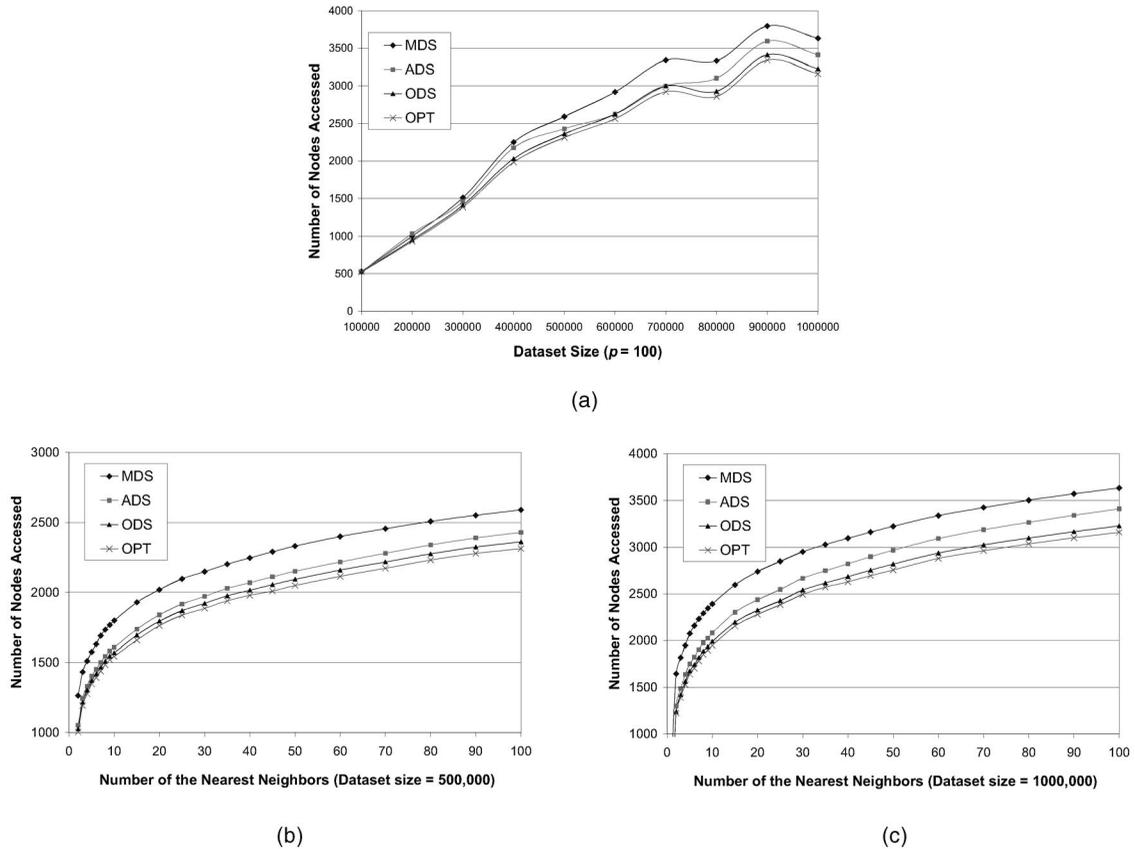


Fig. 22. Performance of ClusterTree for the 12-dimensional data sets with with different size.

monotonically increase when the data set size is 550,000. This is because the Pyramid-Tree ran on a shared computer and its performance was affected by other processes. Figs. 21e and 21f show the query time when we fix  $p$  and the size of the data sets, and vary the dimensions  $D$ . The results presented in Fig. 21 confirm the previous results on the synthetic data sets. The ClusterTree outperforms the competitive index structures, the SR-Tree and the Pyramid-Tree, when we vary the data sets' sizes, dimensions, and number of query points.

#### 6.4.3 Number of Nodes Accessed

We also conducted experiments to test the number of the nodes accessed in the nearest-neighbor search when the query size, the dimensions, and the size of data sets change. We aim to test the effectiveness of the search algorithms and compare their performance in term of the number of accessed nodes for queries. Figs. 22 and 23 show the performance of the ClusterTree with ADS, MDS, ODS, and OPT. Fig. 22a shows the number of the nodes accessed versus the size of the 12-dimensional data sets when  $p = 100$ . The size of the data sets ranges from 100,000 to 1,000,000. Figs. 22b and 22c show the results of a  $p$ -nearest-neighbor search for two data sets with 500,000 and 1,000,000 data points, respectively. ODS has a lower number of accessed nodes than MDS and ADS, and only accesses a few more nodes than OPT. The number of accessed nodes in these four algorithms is not sensitive to the size of the query or the size of the data sets. The rate of the increase is much

lower than linear function. When the size of the data set reaches 1,000,000, the number of accessed nodes even decreases as shown in Fig. 22a. Here, ADS performs better than MDS. In an index structure with many empty regions and overlaps, ADS searches among the sibling nodes, which results in the node having the closest centroid to the query point being picked first. The centroid of a node represents the data points within the node, and the query close to the centroid has a higher probability of obtaining more neighboring points. In this case, ADS can perform better than MDS, but this is not always true. In the following experiments, we will show that MDS sometimes performs better.

Fig. 23a shows the number of nodes accessed vs. the dimension for  $p$ -nearest-neighbor search where  $p = 10$ , and the size of each data set is 100,000. Fig. 23b shows the result when  $p = 100$ , and the size of each data set is also 100,000. Fig. 23c shows that the number of accessed nodes increases slowly as query size increases (the number of dimensions is 25). Fig. 23d shows that the number of accessed nodes does not significantly increase when the dimension is 50. Figs. 23a and 23b show that as the dimensions increase, the number of accessed nodes stays at a very low level. In these figures, no more than 1,400 nodes will be accessed while the total number of nodes built for these data sets is more than 10,000.

From Fig. 23, we observe that MDS outperforms ADS in term of the number of the accessed nodes, and it searches up to 100 less nodes than ADS. ODS performs better than

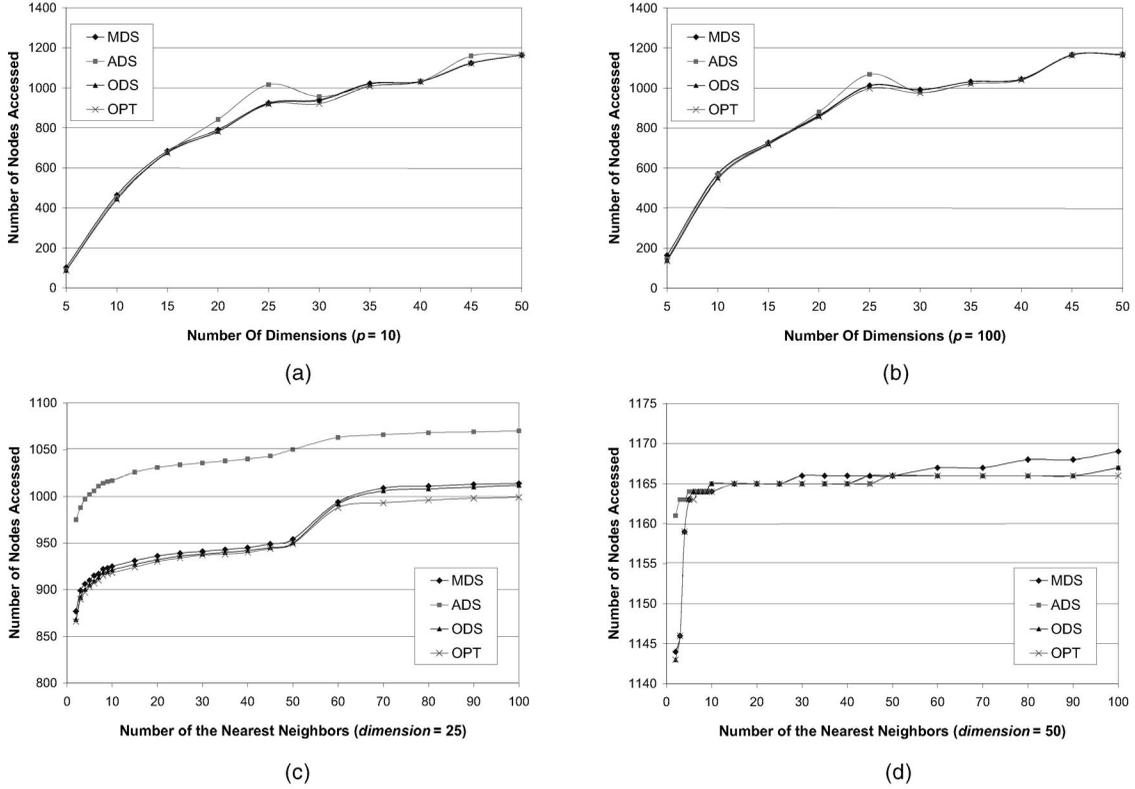


Fig. 23. Performance of SR-Tree on data sets with different dimensions.

MDS and ADS. The superiority of ODS lies in the fact that it uses a successful combination of MDS and ADS in the ClusterTree. MDS fails to make a correct selection when the data points in the node are sparse, and this might result in extra nodes being searched. ADS fails to make a right decision when the data points in the node are compact. Therefore, it might waste time on searching many other nodes which do not have any of the nearest-neighbors before it searches the nodes having the nearest-neighbors.

## 7 CONCLUSION

In this paper, we have presented an indexing approach (termed ClusterTree) to representing clusters in a hierarchical structure. We have also presented an approach to dynamically reconstruct the ClusterTree when new data points are added. By integrating the cluster representation with the index structure, the ClusterTree enhances the performance of the nearest-neighbor search because it can effectively exploit the structure of the data set. The ClusterTree works well for high-dimensional data sets, especially in supporting efficient queries. We conducted comprehensive experiments on both synthetic and real world data sets. The overall results show that the ClusterTree outperforms several of the newest index structures for high-dimensional data sets. The experiments also demonstrated that the clustering and further subclustering approach could greatly reduce the fraction of data that need to be searched to find the nearest-neighbors.

## APPENDIX A

### SELECT ALGORITHM

**Partitioning Algorithm.** Given an array  $A[p..q]$ , the partitioning algorithm divides the array into two regions: the left region which includes the first element and the right region which includes the last element, and puts elements smaller than  $A[p]$  into the left region of the array and elements larger than  $A[p]$  into the right region, where  $A[p]$  is the first element of  $A$ .

The following algorithm rearranges the array  $A[p..r]$  in place.

PARTITION( $A, p, r$ )

- (1)  $x \leftarrow A[p]$
- (2)  $i \leftarrow p - 1$
- (3)  $j \leftarrow r + 1$
- (4) **while** TRUE
- (5)   **do repeat**  $j \leftarrow j - 1$  **until**  $A[j] \leq x$
- (6)    **repeat**  $i \leftarrow i + 1$  **until**  $A[i] \geq x$
- (7)    **if**  $i < j$
- (8)    **then** exchange  $A[i] \leftrightarrow A[j]$
- (9)    **else return**  $j$

SELECT is a divide-and-conquer algorithm for the selection problem. The following code for SELECT returns the  $i$ th smallest element of the array  $A[p..r]$ .

SELECT( $A, p, r, i$ ):

- (1) **if**  $p = r$
- (2)   **then return**  $A[p]$

- (3)  $q \leftarrow \text{PARTITION}(A, p, r)$
- (4)  $k \leftarrow q - p + 1$
- (5) **if**  $i \leq k$
- (6)     **then return**  $\text{SELECT}(A, p, q, i)$
- (7)     **else return**  $\text{SELECT}(A, q + 1, r, i - k)$

The worse-case running time for SELECT is  $\Theta(n^2)$ . The algorithm works well in the average case when  $A$  is a randomized array. To guarantee the best performance, we can toss the elements in  $A$  to make it random. The average time complexity of SELECT is  $O(n)$  in [11].

## ACKNOWLEDGMENTS

The authors would like to thank UCI Repository of machine learning databases for providing them with the forest coverage data set. The authors also thank Stefan Berchtold for posting his source code for the Pyramid-Tree algorithm. The authors also thank Norio Katayama for providing them with the most updated version of SR-Tree source code. Without any of these contributions, it would have been impossible for them to conduct their experiments. This research is supported by US National Science Foundation grants IRI-9733730, EIA-9818289, EIA-9983430, and IRI-9905603.

## REFERENCES

- [1] C.C. Aggarwal, C. Procopiuc, J.L. Wolf, P. Yu, and J.S. Park, "Fast Algorithms for Projected Clustering," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 61-72, 1999.
- [2] S.D. Bay The UCI KDD Archive [http://kdd.ics.uci.edu], Univ. of California, Irvine, Dept. of Information and Computer Science, 1999.
- [3] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, "The R\*-tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM-SIGMOD Int'l Conf. Management of Data*, pp. 322-331, May, 1990.
- [4] K. Bennett, U. Fayyad, and D. Geiger, "Density-Based Indexing for Approximate Nearest-Neighbor Queries," *Proc. Fifth Int'l Conf. KDD*, 1999.
- [5] S. Berchtold, C. Bohm, and H. Kriegel, "The Pyramid-Technique: Towards Breaking the Curse of Dimensionality," *Proc. 1998 ACM SIGMOD Int'l Conf. Management of Data*, pp. 142-153, 1998.
- [6] S. Berchtold, D.A. Keim, and H.-P. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data," *Proc. 22th Int'l Conf. Very Large Data Bases, VLDB '96*, pp. 28-39, 1996.
- [7] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "Nearest-Neighbor" Meaningful?" *Int'l Conf. Database Theory 99*, pp. 217-235, 1999.
- [8] C. Bohm, "A Cost Model for Query Processing in High-Dimensional Data Spaces," *ACM Trans. Database Systems*, vol. 25, no. 2, 2000.
- [9] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces," *Proc. 16th Int'l Conf. Data Eng.*, pp. 440-447, Feb. 2000.
- [10] P. Ciacchia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. 23rd Very Large Databases Conf.*, pp. 426-435, 1997.
- [11] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*. The MIT Press, 1990.
- [12] D.M. Gavril, "R-Tree Index Optimization," *Advances in GIS Research*, T. Waugh and R. Healey, eds., Taylor and Francis, 1994.
- [13] T. Gonzalez, "Clustering to Minimize the Maximum Intercluster Distance," *Theoretical Computer Science*, vol. 38, pp. 311-322, 1985.
- [14] S. Guha, R. Rastogi, and K. Shim, "Cure: An Efficient Clustering Algorithm for Large Databases," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 73-84, 1998.
- [15] A. Guttman, "R-Trees: A Dynamic Index for Geometric Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 47-57, 1984.
- [16] I. Kamel and C. Faloutsos, "On Packing R-Trees," *Proc. Second Int'l Conf. Information and Knowledge Management (CIKM-93)*, pp. 490-499, Nov. 1993.
- [17] N. Katayama and S. Satoh, "The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proc. 1997 ACM SIGMOD Int'l Conf. Management of Data*, pp. 369-380, 1997.
- [18] L. Kaufman and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 1990.
- [19] R. Kurniawati, J.S. Jin, and J.A. Shepherd, "The SS+-Tree: An Improved Index Structure for Similarity Searches in a High-Dimensional Feature Space," *Proc. SPIE Conf. Storage and Retrieval for Image and Video Databases*, pp. 13-24, Feb. 1997.
- [20] B.S. Manjunath and W.Y. Ma, "Texture Features for Browsing and Retrieval of Image Data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 8, pp. 837-842, Aug. 1996.
- [21] R.T. Ng and J. Han, "Efficient and Effective Clustering Methods for Spatial Data Mining," *Proc. 20th Very Large Databases Conf.*, pp. 144-155, 1994.
- [22] J.T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 10-18, Apr. 1981.
- [23] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD*, pp. 71-79, Nov. 1995.
- [24] T. Seidl and H.P. Kriegel, "Optimal Multi-Step k-Nearest-Neighbor Search," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 154-164, 1998.
- [25] G. Sheikholeslami, W. Chang, and A. Zhang, "Semantic Clustering and Querying on Heterogeneous Features for Visual Data," *Proc. Sixth ACM Int'l Multimedia Conf. (ACM Multimedia '98)*, pp. 3-12, Sept. 1998.
- [26] G. Sheikholeslami, S. Chatterjee, and A. Zhang, "WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases," *Proc. 24th VLDB Conf.*, pp. 428-439, Aug. 1998.
- [27] G. Sheikholeslami, S. Chatterjee, and A. Zhang, "WaveCluster: A Wavelet-Based Clustering Approach for Multidimensional Data in Very Large Databases," *The VLDB J.*, vol. 8, no. 4, pp. 289-304, Feb. 2000.
- [28] J.R. Smith and S. Chang, "Transform Features For Texture Classification and Discrimination in Large Image Databases," *Proc. IEEE Int'l Conf. Image Processing*, pp. 407-411, 1994.
- [29] E. Welzl, "Smallest Enclosing Disks (Balls and Ellipsoids)," *Proc. Conf. New Results and New Trends in Computer Science*, pp. 359-370, June, 1991.
- [30] D.A. White, R. Jain, "Similarity Indexing with the SS-Tree," *Proc. 12th Int'l. Conf. Data Eng.*, pp. 516-523, Feb. 1996.
- [31] D. Yu, "Multidimensional Indexing and Management for Large-Scale Databases," PhD dissertation, State Univ. of New York at Buffalo, Feb. 2001.
- [32] M. Zait and H. Messatfa, "A Comparative Study of Clustering Methods," *Future Generation Computer Systems*, vol. 13, pp. 149-159, Nov. 1997.



**Dantong Yu** received the PhD degree in computer science from State University of New York at Buffalo. His research interests include multimedia databases, information retrieval, data mining, and distributed computing. He joined the Brookhaven National Lab in 2001 as part of the DOE Particle Physics Data Grid (PPDG) group. He coordinates the performance monitoring and analysis working group in PPDG.



**Aidong Zhang** received the PhD degree in computer science from Purdue University, West Lafayette, Indiana, in 1994. She was an assistant professor in the Department of Computer Science and Engineering at State University of New York at Buffalo, from 1994 to 1999. She has been an associate professor in the Department of Computer Science and Engineering at State University of New York at Buffalo, since 1999. Her research interests include content-

based image retrieval, geographical information systems, distributed database systems, multimedia database systems, bioinformatics, and data mining. She serves on the editorial boards of *ACM Multimedia Systems*, the *International Journal of Multimedia Tools and Applications*, *International Journal of Distributed and Parallel Databases*, and *ACM SIGMOD DiSC* (Digital Symposium Collection). She has also served on various conference program committees. Dr. Zhang is a recipient of the US National Science Foundation CAREER award. She is a member of the IEEE.

► **For more information on this or any computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.**