An Efficient Technique for Nearest-Neighbor Query Processing on the SPY-TEC

Dong-Ho Lee and Hyoung-Joo Kim, *Member*, *IEEE*

Abstract—The SPY-TEC (Spherical Pyramid-Technique) was proposed as a new indexing method for high-dimensional data spaces using a special partitioning strategy that divides a d-dimensional data space into $2d$ spherical pyramids. In the SPY-TEC, an efficient algorithm for processing hyperspherical range queries was introduced with a special partitioning strategy. However, the technique for processing k-nearest-neighbor queries, which are frequently used in similarity search, was not proposed. In this paper, we propose an efficient algorithm for processing nearest-neighbor queries on the SPY-TEC by extending the incremental nearest-neighbor algorithm. We also introduce a metric that can be used to guide an ordered best-first traversal when finding nearest neighbors on the SPY-TEC. Finally, we show that our technique significantly outperforms the related techniques in processing k-nearest-neighbor queries by comparing it to the R*-tree, the X-tree, and the sequential scan through extensive experiments.

Index Terms—Similarity search, high-dimensional index technique, nearest-neighbor query, incremental nearest-neighbor algorithm, SPY-TEC.

1 Introduction

Feature-based similarity search has become an important search paradigm for various database applications such as multimedia retrieval, data mining, decision support, and statistical and medical applications. The technique used is to map the data items as points into a high-dimensional feature space. The feature space is usually indexed using a multidimensional index structure. Similarity search then corresponds to a hyperspherical range search, which returns all objects within a threshold level of similarity to the query objects, and a k-nearest-neighbor search that returns the k most similar objects to the query object. One of the most popular applications using this technique is a content-based image indexing and retrieval system [3], [6], [15], [4] which extracts several features (such as color, texture, shape, etc.) from images, indexes the images based on those features, and supports similarity queries based on them. To support efficient similarity search in such a system, robust techniques to index high-dimensional feature spaces need to be developed because the feature vectors used are high-dimensional [3], [11].

Initially, traditional multidimensional data structures (e.g., R-tree [1], kd-tree [13], and grid files [14]), which were designed for indexing low-dimensional (two or three-dimensional) spatial data, were used for indexing high-dimensional feature vectors. However, recent research activities [27], [25], [21] reported the result that basically none of the querying and indexing techniques which provide good results on low-dimensional data also perform sufficiently well on high-dimensional data for larger queries. Many researchers have called this problem the *"curse of dimensionality"* [11], and many database-related projects have tried to tackle it [25].

As a result of these research efforts, a variety of new index structures [7], [8], [17], [19], [28], cost models [5], [22], [27], and query processing techniques [29] have been proposed. However, most of the high-dimensional index structures are extensions of the R-tree [1] or the kd-tree [13] adapted to the requirements of high-dimensional indexing. Thus, all of these index structures are limited with respect to data space partitioning and suffer from specific drawbacks of the R-tree or the kd-tree [25].

For example, most of the R-tree-based index structures, such as the TV-tree [17], X-tree [28], SS-tree [8], and SR-tree [19], tend to have low fanouts and a high degree of overlap between bounding regions in higher dimensions. These degrade the performance of query processing in high-dimensional data spaces. Although the X-tree uses a modified R-tree node splitting algorithm to reduce overlap among the index nodes, it has the overhead of performing disk management operations to create and maintain variable sized nodes (so-called supernodes) produced by this modified splitting algorithm. Also, most of the kd-tree-based index structures, such as the k-d-B-tree [16], hB-tree [9], and LSDh-tree [2], suffer from such problems as no guaranteed utilization (e.g., k-d-B-tree) or require storage of redundant information (e.g., hB-tree). In addition to the above drawbacks, these index structures have the well-known drawbacks of multidimensional index structures, such as high costs for insert and delete operations and a poor support of concurrency control and recovery [10].

To overcome these drawbacks, in our earlier work, we proposed a new special space partitioning strategy, the SPY-TEC [10], which is optimized for similarity search in high-dimensional spaces, and proposed the algorithms for processing hyperspherical range queries on the data space partitioned by this strategy. The SPY-TEC first partitions the d-dimensional space into $2d$ spherical pyramids having the

• *The authors are with OOPSLA Laboratory, Department of Computer Engineering, Seoul National University, Shilim-Dong Gwanak-Gu, Seoul 151-742, Korea. E-Mail: {dhlee, hjk}@oopsla.snu.ac.kr.*

Manuscript received 12 Oct. 2000; revised 26 Sept. 2001; accepted 12 Feb. 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112980.

center point of the space as their apex and the curved $(d-1)$-dimensional surface as their bases and then cuts each spherical pyramid into several spherical slices. By using this partitioning strategy of the SPY-TEC, we were able to transform the given d-dimensional data space into a one-dimensional value. Thus, we could use a B^+-tree to store and access data items, and take advantage of all of the benefits of a B^+-tree, such as fast insert, update, and delete operations, and good concurrency control and recovery. However, we could not propose an algorithm for processing nearest-neighbor queries efficiently on the SPY-TEC.

In this paper, we propose the incremental nearest-neighbor algorithm on the SPY-TEC. We also introduce a new metric that can be used to guide an ordered best-first traversal when finding nearest neighbors on the SPY-TEC.

The rest of this paper is organized as follows: Section 2 discusses major algorithms related to nearest-neighbor queries. Section 3 briefly reviews the structure of the SPY-TEC. Section 4 describes the incremental nearest-neighbor algorithm on the SPY-TEC. Section 5 presents the results of an empirical study comparing our technique with the R*-tree, the X-tree, and the sequential scan. Finally, we conclude our work and describe our future plans in Section 6.

2 Related Work

There are numerous algorithms for answering nearest-neighbor or k-nearest-neighbor queries that are motivated by the importance of these queries in fields, including geographical information systems (GIS), document retrieval, pattern recognition, and learning theory [12]. Many of the above algorithms require specialized search structures, but some employ commonly used spatial structures. For example, algorithms exist for the k-d tree, quadtree-related structures, the R-tree, and others. Of these algorithms, there are two major approaches that provide a basis for our work. One was published by Roussopoulos, et al. [20] and we call it the *KNN algorithm* because it was intended for general nearest neighbor or k-nearest-neighbor queries. The other algorithm was published by Hjaltason and Samet [12]. We call it the *INN algorithm* because it used the incremental nearest-neighbor approach. Due to their importance for our work, these algorithms are presented in detail.

In the *KNN algorithm*, the authors proposed an approach for a nearest-neighbor search in the R-tree. The key idea of their work is to maintain a global list (*ActiveBranchList*) of the candidate k nearest neighbors as the R-tree is traversed in a depth-first manner. The authors introduced two important distance functions, MINDIST and MINMAXDIST for ordering nodes that will be visited. MINDIST is the distance from the query point q to the closest point on the boundary of a bounding rectangle r of node n, while MINMAXDIST is the minimum value of all the maximum distances between q and a face (or vertex) of r containing an object o [20]. Fig. 1 shows two examples of the calculation of MINDIST and MINMAXDIST, which are shown with a solid and a broken line, respectively. With these distance functions, the authors proposed three strategies for upward and downward pruning. In some sense, the two orderings represent the optimistic (MINDIST) and the pessimistic (MINMAXDIST) ordering choices because experiments reported in [20] showed that ordering the *ActiveBranchList* using MINDIST consistently performed better than using MINMAXDIST [12]. Since MINDIST represents the minimum distance from a query object q to a bounding rectangle r, it is the most optimistic ordering choice possible. Thus, it provides a means of pruning nodes from the search, given that a bound on the maximum distance is available. On the other hand, MINMAXDIST is an upper bound on the distance of the object o nearest to q. Therefore, it should be clear that MINMAXDIST by itself does not help in pruning the search, as objects closer to q could be found in elements of n at positions with higher MINMAXDIST values [12].

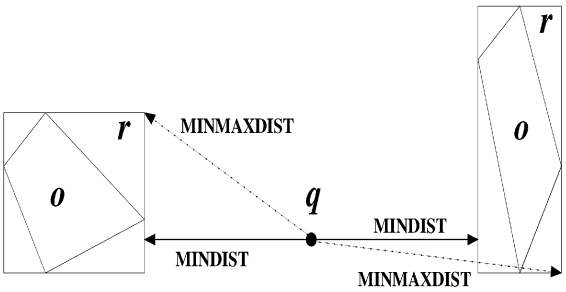


Fig. 1. An example of MINDIST and MINMAXDIST.

In the *INN algorithm*, the authors proposed the incremental nearest-neighbor algorithm that employs what may be termed best-first traversal. When finding k nearest neighbors to the query object using the *KNN algorithm*, k is known prior to the invocation of the algorithm. Thus, if the $(k+1)$th neighbor is needed, the k-nearest-neighbor algorithm needs to be reinvoked for $(k+1)$ neighbors from scratch. To resolve this problem, the authors of the *INN algorithm* proposed the concept of *distance browsing* which is to obtain the neighbors incrementally (i.e., one by one) as they are needed. This operation means browsing through the database on the basis of distance. They showed through various experiments that the *INN algorithm* significantly outperforms the *KNN algorithm* for distance browsing queries and also usually outperforms the *KNN algorithm* when applied to the k-nearest-neighbor problem for the R-tree. They also showed that the two pruning strategies proposed in [20] are only useful when finding the first nearest neighbor, and the one strategy that does not use MINMAXDIST is sufficient when used in a combination with upward and downward pruning in their algorithm. This implies that MINMAXDIST is not necessary for pruning in the incremental nearest neighbor approach.

In the KNN algorithm, finding a leaf node containing a query object in a spatial index can be done in a depth-first manner by recursively descending the tree structure [12]. With this method, the recursion stack keeps track of what nodes have yet to be visited. Therefore, the order where nodes are visited is the order where the node is visited during tree traversal. However, the INN algorithm picks the node with the least distance in the set of all nodes that have yet to be visited when deciding what node to traverse next on the R-tree. Thus, it uses a priority queue where the distance from the query point is used as a key. Therefore, in the INN algorithm, the order where nodes are visited is independent of the structure of the index.

To the best of our knowledge, the *INN algorithm* is one of the most efficient algorithms for finding the nearest neighbor or k nearest neighbors. However, this algorithm does not provide good results on high-dimensional data either, as we will show in our experimental evaluation. This is not a problem of the *INN algorithm* itself, but a problem of the spatial index structure (R-tree), which does not support efficient indexing or query processing structurally on a high-dimensional data space.

In this paper, we propose a new metric that can be used to guide an ordered best-first traversal when finding nearest neighbors on the SPY-TEC. We also propose an efficient incremental nearest-neighbor algorithm based on this new metric on the SPY-TEC.

3 The SPY-TEC

In [25], Berchtold et al. proposed a special partitioning strategy (Pyramid-Technique) that divides the data space first into $2d$ pyramids and then cuts each pyramid into several slices. They also proposed the algorithms for processing hypercubic range queries on the space partitioned by this strategy. However, the shape of queries used in similarity search is not a hypercube, but a hypersphere [3], [6], [11], [30]. Thus, when processing hyperspherical range queries with the Pyramid-Technique, there is a drawback that exists in all index structures based on the bounding rectangle [10], [11].

The main idea of the SPY-TEC is based on the observation that spherical splits will be better than right-angled splits of the Pyramid-Technique for similarity search. This observation is due to the fact that the shape of the queries used in similarity search is not a hypercube, but a hypersphere. Although we have presented the basic idea and space partitioning strategy of the SPY-TEC in [10], we should explain it again briefly for better understanding of our incremental nearest-neighbor algorithm on the SPY-TEC.

The SPY-TEC is to transform d-dimensional data points into one-dimensional values and then store and access the values using the B^+-tree. Also, we store a d-dimensional point $plus$ the corresponding one-dimensional key as a record in the leaf nodes of the B^+-tree. Therefore, we do not need an inverse mechanism of this transformation. The transformation itself is based on a specific partitioning of the SPY-TEC. To define the transformation, we first explain the data space partitioning strategy of the SPY-TEC.

3.1 Data Space Partitioning

The SPY-TEC partitions the data space in two steps: In the first step, we split the d-dimensional data space into $2d$ spherical pyramids having the center point of the data space (0.5, 0.5, ..., 0.5) as their apex and a $(d-1)$-dimensional curved surface of the data space as their bases. The second step is to divide each of the $2d$ spherical pyramids into several spherical slices, with each slice corresponding to one data page of the B^+-tree. Fig. 2 shows the data space partitioning of the SPY-TEC in a two-dimensional example. First, the two-dimensional data space has been divided into four spherical pyramids resembling fans. Each of these spherical pyramids has the center point of the data space as its apex and one curved line of the data space as its base. In the second step, each of these four spherical pyramids is split again into several data pages which are shaped like the annual rings of a tree. Given a d-dimensional space instead of the two-dimensional space, the base of the spherical pyramid is not a 1-dimensional curved line as in the example, but a $(d-1)$-dimensional spherical surface. As a sphere of dimension d has $2d$ $(d-1)$-dimensional spherical surface as a surface, we obviously obtain $2d$ spherical pyramids [10].

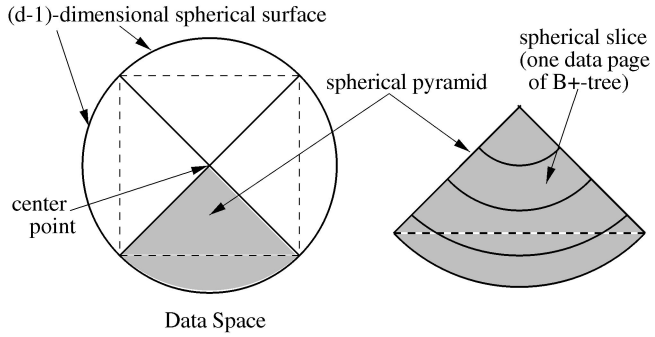


Fig. 2. Partitioning strategy of the SPY-TEC.

Numbering the spherical pyramids is based on the following observation: All points v located in the ith spherical pyramid sp_i have the common property that the distance in the ith coordinate from the center point is either smaller than the distance of all other coordinates if $i < d$, or larger if $i \geq d$ [25]. Therefore, given a point v, we have to find the dimension i having the maximum deviation $|0.5 - v_i|$ from the center to determine the spherical pyramid containing the point v. If v_i is greater than or equal to 0.5, then the spherical pyramid containing the point v is sp_{i+d}. If it is smaller than 0.5, the spherical pyramid containing the point v is sp_i.

Fig. 3a shows the process of numbering the spherical pyramids in a two-dimensional example. As depicted in Fig. 3a, the value of $|0.5 - v_1|$ of a point v in a two-dimensional space is greater than the value of $|0.5 - v_0|$. Thus, the dimension having the maximum deviation $|0.5 - v_i|$ from the center is d_1 and the value of v_1 is smaller than 0.5. Therefore, the point v belongs to the spherical pyramid sp_1. For example, consider another point $v' = (0.8, 0.4)$. The dimension having the maximum deviation from the center for each dimension of v' is

$$d_0(0.3 = |0.5 - v'_0| > |0.5 - v'_1| = 0.1).$$

Also, the value of v'_0 is greater than 0.5. Therefore, the point v' belongs to the spherical pyramid $sp_{(0+2)}$. Although the formal expression of this procedure was presented in [10], we redefine it formally for better understanding of the partitioning strategy of the SPY-TEC.

Definition 1 (Spherical Pyramid of a Point v). *A d-dimensional point v is defined to be located in a spherical pyramid sp_i.*

$$i = \begin{cases} j_{max} & \text{if } v_{j_{max}} < 0.5 \\ (j_{max} + d) & \text{if } v_{j_{max}} \geq 0.5 \end{cases}$$

$$j_{max} = (j | (\forall k, 0 \leq (j,k) < d, j \neq k : |0.5 - v_j| \geq |0.5 - v_k|)).$$

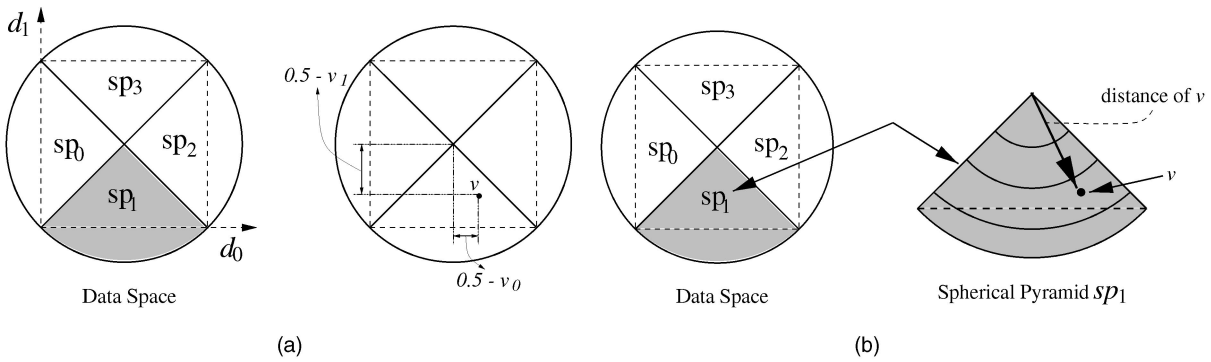


Fig. 3. The SPY-TEC. (a) Numbering of spherical pyramids. (b) Distance of a point within its spherical pyramid.

In Definition 1, j_{max} is the dimension having the maximum deviation $|0.5 - v_i|$ from the center for each dimension of a d-dimensional point v and i is the number of the spherical pyramid containing v.

In order to transform d-dimensional data into a one-dimensional value, we have to determine the location of a point v within its spherical pyramid. The Pyramid-Technique uses the height of the point within the pyramid as the location of the point. However, we use the distance from the point to the center point of the data space as the location of the point. Fig. 3b shows the process of determining the distance of the point v as the location within its spherical pyramid. We assume that the distance function is the Euclidean distance which is frequently used for similarity measurement in content-based image retrieval [3], [11]. More formally, see Definition 2.

Definition 2 (Distance of a point v). *Given a d-dimensional point v, the distance d_v of the point v is defined as*

$$d_v = \sqrt{\sum_{i=0}^{d-1}(0.5 - v_i)^2}.$$

According to Definitions 1 and 2, we are able to transform a d-dimensional point v into a one-dimensional value $(i \cdot \lceil\sqrt{d}\rceil + d_v)$. In this one-dimensional value, i is the number of the spherical pyramid containing the point v, d is the dimension of the point v, and d_v is the distance from the point v to the apex of its spherical pyramid. More formally, see Definition 3.

Definition 3 (Spherical Pyramid Value of a Point v). *Given a d-dimensional point v, let i be the number of the spherical pyramid containing v according to Definition 1, and d_v be the distance of v according to Definition 2. Then, the spherical pyramid value spv_v of v is defined as*

$$spv_v = (i \cdot \lceil\sqrt{d}\rceil + d_v).$$

Note that i is an integer in the range $[0,\ 2d]$, d_v is a real number in the range $[0,\ 0.5\sqrt{d}]$ and $\lceil\sqrt{d}\rceil$ is the smallest integer not less than or equal to $\sqrt{d}$. Therefore, every point within a spherical pyramid sp_i has a value in the interval $[i \cdot \lceil\sqrt{d}\rceil,\ (i \cdot \lceil\sqrt{d}\rceil + 0.5\sqrt{d})]$. In order to make the sets of spherical pyramid values covered by any two spherical pyramids sp_i and sp_j be disjoint, we multiply the spherical pyramid number i by $\lceil\sqrt{d}\rceil$. Without this multiplication of i by $\lceil\sqrt{d}\rceil$, the interval of every point within a spherical pyramid sp_i would be $[i,\ (i + 0.5\sqrt{d})]$. Thus, there might be intersections in the sets of spherical pyramid values covered by any two spherical pyramids sp_i and sp_j when the dimension is higher than four. Note that this transformation is not injective. That is, two points v and v' may have the same spherical pyramid value, but, as mentioned above, we do not need an inverse transformation because we store a d-dimensional point $plus$ the corresponding one-dimensional key as a record in the leaf nodes of the B^+-tree. Therefore, the SPY-TEC does not require a bijective transformation [10].

Note further that, in Figs. 2 and 3b, the outer slice of a spherical pyramid seems to be divided into a number of small pieces in the corner of the data space. However, the slice of a spherical pyramid is not physically divided into small pieces. The slice is the smallest unit that corresponds to one data page of the B^+-tree. The points of these small pieces are not stored in different data pages. That is, the points inside one slice are physically stored in one data page of the B^+-tree. The points inside one outer slice are also stored in one data page of the B^+-tree.

3.2 Index Creation

It is a very simple task to build an index using the SPY-TEC. Given a d-dimensional point v, we first determine the spherical pyramid value spv_v of the point and then insert the point into a B^+-tree using spv_v as a key. Finally, we store the point v and spv_v in the corresponding data page of the B^+-tree. Update and delete operations can be done similarly.

The spherical pyramid values of points that all belong to the same spherical pyramid lies in the interval given by the minimum and maximum key values of the data pages. Thus, a single B^+-tree data page corresponds to a spherical slice of a spherical pyramid as shown in Fig. 2 (right). The page regions of the R-tree are (minimum) bounding rectangles, whereas the page regions of the SPY-TEC are spherical slices. Thus, in the rest of the paper, we call the spherical slice the *bounding slice (BS)*.

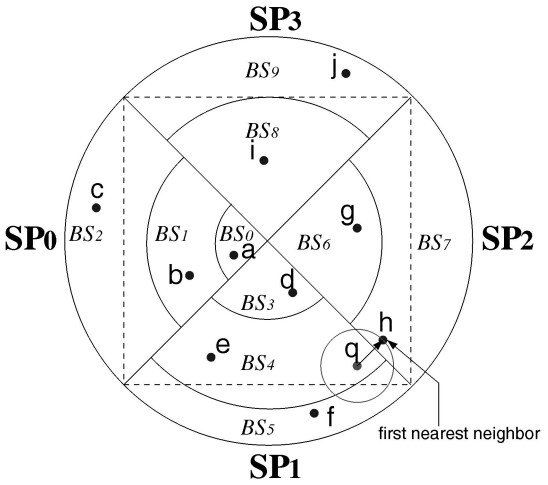


Fig. 4. An example of the SPY-TEC for a set of 10 points.

4 Incremental Nearest-Neighbor Algorithm on the SPY-TEC

In this section, we present the incremental nearest-neighbor algorithm that extends the original *INN algorithm* [12] in order to adapt to the structure of the SPY-TEC. As mentioned above, the *INN algorithm* proposed in [12] picks the node with the least distance in the set of all nodes that have yet to be visited when deciding what node to traverse next on the R-tree. This means that, instead of using a stack or a plain queue to keep track of the nodes to be visited, it uses a priority queue where the distance from the query point is used as a key. In our algorithm, we also use a priority queue where the distance from the query point to the nodes or objects is used as a key.

This section is organized as follows: In Section 4.1, we introduce a new distance metric that can be used to guide an ordered best-first traversal when finding nearest neighbors on the SPY-TEC. Then, we present the incremental nearest-neighbor algorithm on the SPY-TEC in Section 4.2. Finally, in Section 4.3, we give a concrete example of the execution of the algorithm on the SPY-TEC.

4.1 Metrics for Nearest-Neighbor Search

As mentioned in Section 2, about the two distance metrics (MINDIST and MINMAXDIST) proposed in the *KNN algorithm*, only MINDIST is used for the incremental nearest-neighbor approach [12]. In our approach also, MINMAXDIST is not necessary for pruning. Therefore, we need the minimum possible distance from the query object to a node in the SPY-TEC. Fig. 4 shows an example of the SPY-TEC in a two-dimensional data space. For the sake of simplicity, we assume that each bounding slice contains one object. In Fig. 4, the query point falls within a bounding slice BS_4 in the spherical pyramid sp_1. As with most nearest-neighbor algorithms, we must first visit the page (BS_4 in this example) containing the query point. Then, we visit the next page with the second smallest minimum distance from the query point. To do so, we must calculate the minimum possible distance from the query point to a spherical pyramid or a bounding slice. We first describe the process of calculating the minimum distance between the query point and a spherical pyramid, and then discuss the process of calculating the minimum distance between the query point and a bounding slice.

Lemma 1, which follows, measures the minimum distance MINDIST(q, sp_i) from the query point q to a spherical pyramid sp_i. For the sake of simplicity, we focus on the description of the case only for spherical pyramids sp_i, where $i < d$. However, this lemma can be extended to all spherical pyramids in a straight-forward manner [10].

Lemma 1 (Minimum Distance from a Query Point to a Spherical Pyramid). *Given a query point*

$$(q = [q_0, q_1, \ldots, q_{d-1}]),$$

let sp_j $(j < d)$ be the spherical pyramid containing a query point and sp_i be the spherical pyramid that will be examined for the minimum possible distance from q. The minimum distance from q to sp_i, MINDIST(q, sp_i), is defined as

$$\text{MINDIST}(q, sp_i) = \begin{cases} 0 & \text{if } i = j \\ d_q & \text{if } |i-j| = d \\ \dfrac{|q_j - q_i|}{\sqrt{2}} & \text{if } i < d \\ \dfrac{|q_j + q_i - 1|}{\sqrt{2}} & \text{if } i > d. \end{cases}$$

Proof. Given a point $([q_0, q_1, \ldots, q_{d-1}])$ and a hyperplane $(k_0x_0 + k_1x_1 + \ldots + k_{d-1}x_{d-1} + C = 0)$, the distance from the point to the hyperplane in Euclidean geometry is defined as

$$Distance = \frac{|k_0q_0 + k_1q_1 + \ldots + k_{d-1}q_{d-1} + C|}{\sqrt{{k_0}^2 + {k_1}^2 + \ldots + {k_{d-1}}^2}}. \quad (1)$$

We are able to prove the case $(i > d)$ and the case $(i < d)$ using this formula.

1. If $i = j$, sp_i is the spherical pyramid containing the query point q. Therefore, MINDIST$(q, sp_i) = 0$, which is less than or equal to the distance of q from any point in sp_i.
2. If $|i - j| = d$, sp_i is the spherical pyramid on the opposite side of sp_j. Therefore, the minimum distance of q from sp_i is the distance from q to the apex of sp_i (the center of the data space). Thus, according to the notation of Definition 2, MINDIST(q, sp_i) = d_q.
3. In (1), the index k_n and the constant C have discrete values [-1, 0, 1] because of unit space. If $i < d$, the equation for the closest side plane of a spherical pyramid adjacent to the query point is $k_jx_j + k_ix_i = 0$ as depicted in the 2-dimensional example of Fig. 5. This formula can be extended to a d-dimensional data space in a straight-forward way. Given a d-dimensional space instead of the two-dimensional space, the side plane of a spherical pyramid is not a one-dimensional line as in the example of Fig. 5, but a $(d-1)$-dimensional hyperplane, and the equation for this $(d-1)$-dimensional hyperplane has the common property that all indices except k_i and k_j are 0. In this case, $k_j = 1$ and $k_i = -1$

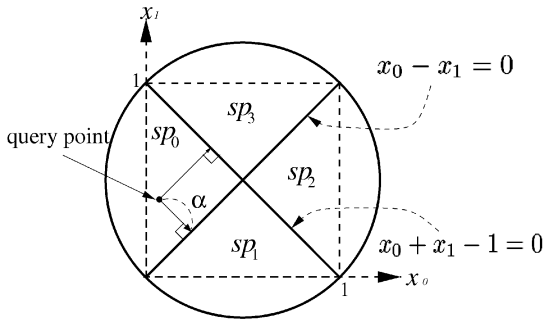


Fig. 5. The minimum distance from the query point to a spherical pyramid.

because $i < d$. Thus, the minimum distance from the query point to the closest side plane of an adjacent spherical pyramid sp_i is $|q_j - q_i|/\sqrt{2}$. Therefore, MINDIST$(q, sp_i) = |q_j - q_i|/\sqrt{2}$.

4. If $i > d$, the equation for the closest side plane of a spherical pyramid adjacent to the query point is $k_j x_j + k_i x_i - 1 = 0$ (refer to Fig. 5). In this case, $k_j = 1$ and $k_i = 1$ because $i > d$. Thus, the minimum distance from the query point to the closest side plane of an adjacent spherical pyramid sp_i is $|q_j + q_i - 1|/\sqrt{2}$. Therefore, MINDIST$(q, sp_i) = |q_j + q_i - 1|/\sqrt{2}$. □

Calculating the minimum distance from the query point to a bounding slice is more complex than the case of the minimum distance from the query point to a spherical pyramid. However, as depicted in Fig. 6, we can present it easily by classifying it into three cases: the case of bounding slices which belong to the spherical pyramid containing the query point (**Case 1**), the case of bounding slices that belong to the spherical pyramid on the opposite side of the query point (**Case 2**), and the case of bounding slices that belong to a spherical pyramid adjacent to the query point (**Case 3**).

Lemma 2, which follows, measures the minimum distance MINDIST(q, BS_l) from the query point q to a bounding slice BS_l in a spherical pyramid sp_i.

Lemma 2 (Minimum Distance from a Query Point to a Bounding Slice). *Given a query point (q), let sp_j be the spherical pyramid containing a query point, and BS_l be the bounding slice that belongs to a spherical pyramid sp_i. The minimum distance from q to a bounding slice BS_l, MINDIST(q, BS_l), is defined as*

- ***Case 1:*** *($i = j$: the case of BS_l belonging to the spherical pyramid that contains q).*

$$\text{MINDIST}(q, BS_l) = \begin{cases} |d_q - max(BS_l)| & \text{if } d_q > max(BS_l) \\ 0 & \text{if } min(BS_l) \le d_q \le max(BS_l) \\ |d_q - min(BS_l)| & \text{if } d_q < min(BS_l). \end{cases}$$

- ***Case 2:*** *($|i - j| = d$: the case of BS_l belonging to the spherical pyramid on the opposite side of q).*

  Let γ be the distance from the closest side plane of a spherical pyramid adjacent to q and $\theta(\le \pi/4)$ be the angle of a right-angled triangle which consists of two sides, γ and d_q $\left(sin\theta = \frac{\gamma}{d_q}\right)$,

$$\text{MINDIST}(q, BS_l) = \sqrt{{d_q}^2 + min(BS_l)^2 - 2d_q min(BS_l) cos\left(\theta + \frac{\pi}{2}\right)}.$$

- ***Case 3:*** *(otherwise : the case of BS_l belonging to a spherical pyramid adjacent to q).*

  Let δ be the length of the base line in a right-angled triangle which consists of two sides, γ and d_q,

$$\text{MINDIST}(q, BS_l) = \begin{cases} \sqrt{|\delta - max(BS_l)|^2 + \gamma^2} & \text{if } \delta > max(BS_l) \\ \gamma & \text{if } min(BS_l) \le \delta \le max(BS_l) \\ \sqrt{|\delta - min(BS_l)|^2 + \gamma^2} & \text{if } \delta < min(BS_l), \end{cases}$$

where

$$min(BS_l) = \{d_v \mid (\forall v', v, v' \in BS_l : d_v \le d_{v'})\}$$
$$max(BS_l) = \{d_v \mid (\forall v', v, v' \in BS_l : d_v \ge d_{v'})\}.$$

Proof. $min(BS_l)$ is d_v of the point v having the smallest value of the points belonging to BS_l, while $max(BS_l)$ is d_v of the point v having the largest value. We can prove each case by using $min(BS_l)$ and $max(BS_l)$.

1. If $min(BS_l) \le d_q \le max(BS_l)$, then q is inside BS_l. Therefore, MINDIST$(q, BS_l) = 0$ because it is less than or equal to the distance of q from any point inside BS_l. If $d_q > max(BS_l)$, the distances of all of the points in BS_l from the center of the space are less than the distance of q from the center of the space. Therefore, MINDIST(q, BS_l) is the difference between d_q and d_v, where the point v is in BS_l and is farthest from the center of the space. That is, MINDIST$(q, BS_l) = |d_q - max(BS_l)|$. Finally, if $d_q < min(BS_l)$, the distance of q from the center is less than the distances of all of the points in BS_l from the center. Therefore, MINDIST(q, BS_l) is the difference between d_q and d_v, where the point v is in BS_l and is closest to the center. That is, MINDIST$(q, BS_l) = |d_q - min(BS_l)|$. In Fig. 6a, MINDIST$(q, BS_4)$ is 0 because q is inside BS_4. Also, MINDIST(q, BS_3) is $|d_q - max(BS_3)|$ because the distances of all of the points in BS_3 are less than the distance of q. Finally, MINDIST(q, BS_5) is $|d_q - min(BS_5)|$ because the distance of q is less than the distances of all of the points in BS_5.
2. If $|i - j| = d$, sp_i is on the opposite side to the spherical pyramid containing q. In this case, the minimum distance from q to BS_l inside sp_i is the length of the base of a triangle which consists of two sides, such as d_q and $min(BS_l)$, and the angle between them as depicted in Fig. 6b. By using the *cosine rule* [18], we can get the length of the base of a triangle. First, the angle of the apex of a spherical pyramid is $\pi/2$.

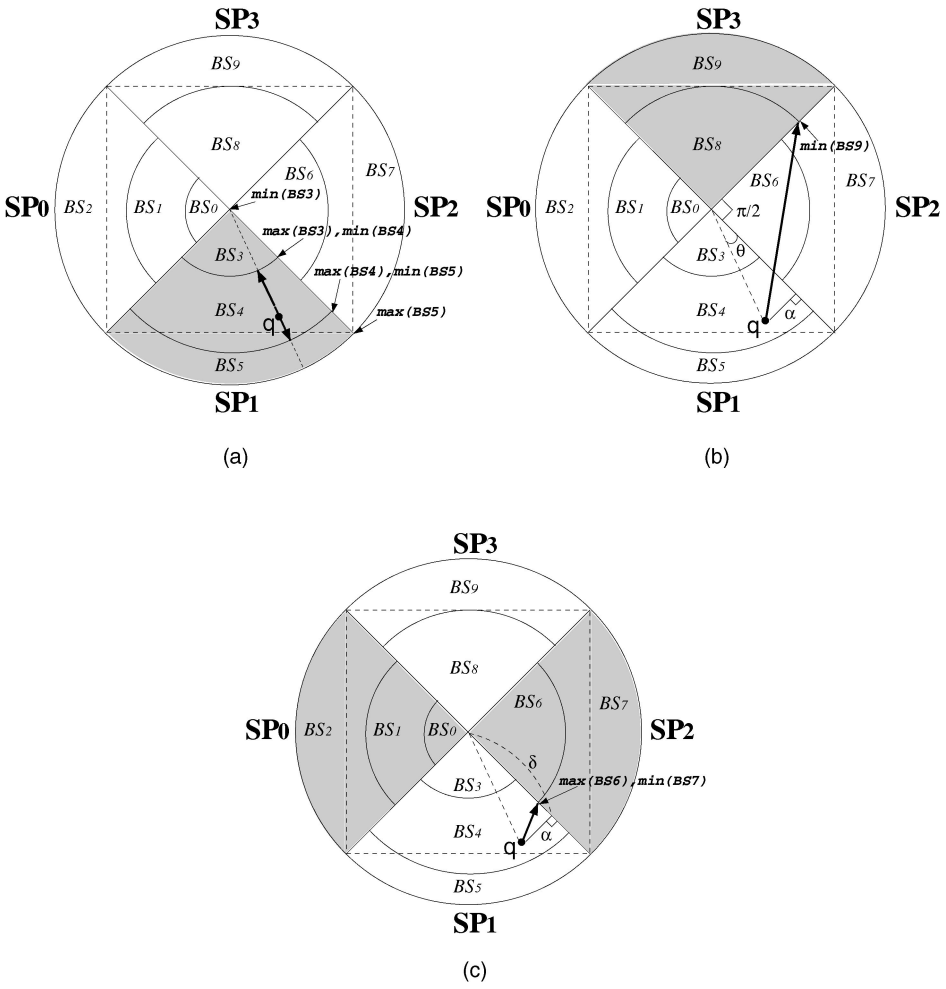


Fig. 6. The minimum distance from the query point to a bounding slice. (a) Case 1, (b) Case 2, and (c) Case 3.

Thus, the angle between d_q and $min(BS_l)$ is $(\theta + \pi/2)$, where $\theta = \arcsin(\gamma/d_q)$. Given the lengths of two sides (b and c) and the angle (A) between them, the cosine rule states: $a^2 = b^2 + c^2 - 2bc \cdot cosA$. Therefore, by the *cosine rule*,

$$\text{MINDIST}(q, BS_l) = \sqrt{{d_q}^2 + min(BS_l)^2 - 2d_q min(BS_l) \cdot cos\left(\theta + \frac{\pi}{2}\right)}.$$

Fig. 6b shows this case in a two-dimensional example. MINDIST(q, BS_8) is d_q because

$$min(BS_8) = 0.$$

3. In this case, sp_i is adjacent to sp_j which contains q. All subcases of this case are similar to those of **Case 1** except that the parameter for classifying each subcase is not d_q, but δ. If

$$min(BS_l) \leq \delta \leq max(BS_l),$$

MINDIST(q, BS_l) is the distance from q to the closest side plane of sp_i. That is, MINDIST$(q, BS_l) = \gamma$. This is similar to the subcase

$$(min(BS_l) \leq d_q \leq max(BS_l))$$

of **Case 1**. If $\delta > max(BS_l)$, MINDIST(q, BS_l) is the length of the hypotenuse in a right-angled triangle which consists of two sides, γ and $|\delta - max(BS_l)|$. Therefore,

$$\text{MINDIST}(q, BS_l) = \sqrt{|\delta - max(BS_l)|^2 + \gamma^2}.$$

Finally, if $\delta < min(BS_l)$, MINDIST(q, BS_l) is the length of the hypotenuse in a right-angled triangle which consists of two sides, γ and $|\delta - min(BS_l)|$. Therefore,

$$\text{MINDIST}(q, BS_l) = \sqrt{|\delta - min(BS_l)|^2 + \gamma^2}.$$

Fig. 6c shows this case in a two-dimensional example. □

4.2 Algorithm Description

In this section, we describe the incremental algorithm for processing nearest neighbor or *k*-nearest-neighbor queries on the SPY-TEC by using Lemmas 1 and 2.

Algorithm 1:
Processing the incremental nearest neighbor query

```
1:  for i = 0 to 2d − 1 do
2:    dist = MINDIST(q, sp_i); /*Using Lemma 1*/
3:    ENQUEUE(queue, sp_i, dist);
4:  end for
5:
6:  while not ISEMPTY(queue) do
7:    Element = DEQUEUE(queue);
8:    if Element is a spherical pyramid then
9:      for each bounding slice in a spherical pyramid do
10:       dist = MINDIST(q, BS_l); /* Using Lemma 2*/
11:       ENQUEUE(queue, BS_l, dist);
12:     end for
13:   else if Element is a bounding slice then
14:     for each object in a bounding slice do
15:       dist = DIST_QUERY_TO_OBJ(q, object);
16:       ENQUEUE(queue, object, dist);
17:     end for
18:   else /* Element is a object */
19:     report element as the next nearest object
20:   end if
21: end while
```

Algorithm 1 shows the algorithm for processing the nearest neighbor query. In lines 1 - 4, the distances of each spherical pyramid from the query point are calculated by using Lemma 1, and then information about each spherical pyramid and its distance are inserted into the priority queue. Since the distance is used as a key in the priority queue, the spherical pyramid closest to the query point is at the head of the queue. The **while**-loop of lines 6 - 21 is the main loop for the algorithm. In line 7, the first element in the head of the queue is dequeued and, according to the type of the element, appropriate operations will be performed. If the type of the element dequeued is a spherical pyramid, as depicted in lines 8 - 12, the distances of each bounding slice in the spherical pyramid from the query point are calculated, and then information of each bounding slice and its distance are inserted into the queue by using Lemma 2. If the type is a bounding slice, as depicted in lines 13 - 17, the distances of each object in the bounding slice from the query point are calculated, and then inserted into the queue. Finally, if the type is an object, it is reported as the next nearest-neighbor object. The first reported object is naturally the nearest neighbor to the query point. Since the element with the smallest distance from the query point is at the head of the queue, the reported object is always the next nearest object to the query point.

It is a very simple task to extend Algorithm 1 for processing *k*-nearest-neighbor queries. If we control the number of reported nearest neighbors in the **while**-loop of Algorithm 1, we can easily process the *k*-nearest-neighbor query.

4.3 Example

As an example, suppose that we want to find the three nearest neighbors to the query point q in the SPY-TEC given in Fig. 4. Below, we show the steps of the algorithm and the contents of the priority queue. Algorithm 1 must compute the distances between q and the spherical pyramids or bounding slices. Table 1 shows these distances (SP means spherical pyramid and BS means bounding slice). When depicting the contents of the priority queue, the spherical pyramids and bounding slices are listed with their distances from the query point q, in order of increasing distance. The objects are denoted in bold letters (e.g., **a**). The algorithm starts by enqueueing $SP_0 \sim SP_3$, after which it executes the following steps:

TABLE 1
Distances of Spherical Pyramids and Bounding Slices from the Query Point q in the SPY-TEC of Fig. 4

SP	Dist.
SP_0	21
SP_1	0
SP_2	4
SP_3	33

BS	Dist.
BS_0	21
BS_1	25
BS_2	29
BS_3	14
BS_4	0
BS_5	2
BS_6	8
BS_7	4
BS_8	33
BS_9	42

OBJ	Dist.
a	23
b	27
c	45
d	16
e	19
f	12
g	35
h	6
i	39
j	47

1. Enqueue $SP_0 \sim SP_3$. Queue :
$$\{[SP_1, 0], [SP_2, 4], [SP_0, 21], [SP_3, 33]\}.$$
2. Dequeue SP_1, enqueue BS_3, BS_4, BS_5. Queue :
$$\{[BS_4, 0], [BS_5, 2], [SP_2, 4], [BS_3, 14], [SP_0, 21], [SP_3, 33]\}.$$
3. Dequeue BS_4, enqueue **e**. Queue :
$$\{[BS_5, 2], [SP_2, 4], [BS_3, 14], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33]\}.$$
4. Dequeue BS_5, enqueue **f**. Queue :
$$\{[SP_2, 4], [\mathbf{f}, 12], [BS_3, 14], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33]\}.$$
5. Dequeue SP_2, enqueue BS_6, BS_7. Queue :
$$\{[BS_7, 4], [BS_6, 8], [\mathbf{f}, 12], [BS_3, 14], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33]\}.$$
6. Dequeue BS_7, enqueue **h**. Queue :
$$\{[\mathbf{h}, 6], [BS_6, 8], [\mathbf{f}, 12], [BS_3, 14], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33]\}.$$
7. Dequeue **h**, report **h** as nearest neighbor. Queue :

$$\{[BS_6, 8], [\mathbf{f}, 12], [BS_3, 14], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33]\}.$$

8. Dequeue BS_6, enqueue **g**. Queue :

$$\{[\mathbf{f}, 12], [BS_3, 14], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33], [\mathbf{g}, 35]\}.$$

9. Dequeue **f**, report **f** as second nearest neighbor. Queue :

$$\{[BS_3, 14], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33], [\mathbf{g}, 35]\}.$$

10. Dequeue BS_3, enqueue **d**. Queue :

$$\{[\mathbf{d}, 16], [\mathbf{e}, 19], [SP_0, 21], [SP_3, 33], [\mathbf{g}, 35]\}.$$

11. Dequeue **d**, report **d** as third nearest neighbor.

Since the elements in the priority queue are sorted in increasing order of distance, sp_1 containing the query point q is at the head of the queue. In line 7 of Algorithm 1, sp_1 is dequeued, and then BS_3, BS_4, and BS_5 in sp_1 are enqueued in increasing order of their distances from the query point. Now, BS_4 is at the head of the queue because it has the smallest distance. BS_4 is dequeued, and then the objects in BS_4 are enqueued. In this example, since we assume that only one object is contained in a bounding slice, the object **e** in BS_4 is enqueued. These operations are repeated until the user finds as many nearest neighbors as desired.

5 Experimental Evaluation

We performed various experiments to show the practical impact of the incremental nearest-neighbor algorithm on the SPY-TEC and compared it to the R*-tree and the X-tree, as well as the sequential scan.

The R*-tree has been chosen for comparison because it is most commonly used in multidimensional indexing applications, and the X-tree was proposed as an indexing structure for high-dimensional data. Thus, we included these techniques in our experiments. Recently, the criticism arose that index-based query processing is generally inefficient in high-dimensional data spaces and that sequential scan processing yields better performance in this case [21], [25]. According to [21], the performance evaluations of high-dimensional nearest-neighbor queries must include a comparison to the sequential scan as a sanity check. Therefore, we also included the sequential scan in our experiments.

For clear comparison, we implemented the incremental nearest-neighbor algorithm on the R*-tree and the X-tree using the algorithm proposed in [12]. We used the hybrid memory/disk-based priority queue proposed in [12], where the contents of the priority queue are stored in the memory or disk according to the distances of the queue elements. For more detail, refer to Section 4.8 in [12].

All experiments were performed on a SUN SPARC 20 workstation with 128 MByte main memory and 10 GByte secondary storage. The block size used for our experiments was 4 KBytes, and all index structures were allowed to use the same amount of cache that hold about 1-8 percent of the database size. For a strict experimental environment, we measured the performance from a "cold start"—that is, execute as if previous tests have not loaded crucial data into system caches. That is, before performing each query, we flushed the operating system buffer cache by doing a recursive read on the dummy file with a big size.

We mainly focused on the experiments using clustered data and real data sets which are meaningful workloads for high-dimensional nearest-neighbor queries. However, we also performed the experiment on uniformly distributed data sets as a sanity check, even though it is not suitable for evaluating the performance of index structures in high dimensionality [21]. The number of block accesses, total search time, and CPU time for each query were recorded. The graphs throughout this section show the results of these experiments.

5.1 Clustered Gaussian Data

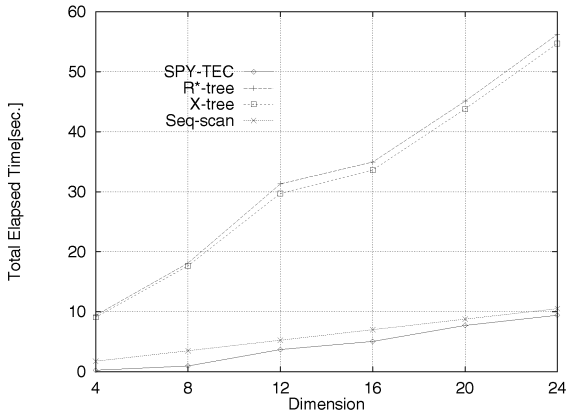
Clustered Gaussian data sets were created for 4-24 dimensions. Each data sets consisted of 20,000 - 100,000 data points that were partitioned into 10 clusters. The points for the clustered data sets were generated in the range [0, 1] for each dimension. In order to create clustered data sets, 10 points were chosen from the uniform distribution and a Gaussian distribution with a standard deviation 0.05 was centered at each point. For each clustered data set, we performed 10-nearest neighbor queries with 100 query points that were taken from within the data set itself. Thus, the result was evaluated as the average of 100 random trials.

In the first experiment on clustered data sets, we measured the performance behavior while we varied data space dimension. For this experiment, we created six files with the dimensionalities 4, 8, 12, 16, 20, and 24. The database size was set to 100,000 points. Thus, each cluster contained 10,000 points.

Fig. 7 shows the total search time and cache configuration of the index structures used in our experiments. Although the search time of all of the index structures increase with growing dimension, the SPY-TEC significantly outperforms the R*-tree, the X-tree, and the sequential scan in all cases. However, the R*-tree and the X-tree are beaten by the sequential scan as might have been expected.

In a four-dimensional space, the SPY-TEC performs 10-nearest-neighbor queries 37.37 times faster than the R*-tree, 36.02 time faster than the X-tree, and 6.96 times faster than the sequential scan. Even in 24-dimensional space, the SPY-TEC performs the queries 5.97 times faster than the R*-tree, 5.80 times faster than the X-tree, and 1.12 times faster than the sequential scan.

In Fig. 8, we show a more detailed comparison of the number of block accesses and CPU time. The performances of the number of block accesses and CPU time show a similar behavior to that of the total search time except that the R*-tree and the X-tree access fewer blocks than does the sequential scan. As depicted in Fig. 8a, the speed-up with respect to the number of block accesses ranges between 1.48 and 5.63 over the R*-tree, between 1.41 and 5.63 over the X-tree, and between 5.33 and 48.05 over the sequential scan. The speed-up in the number of block accesses reaches its highest value with its lowest dimension and is decreasing



(a)

Dimension	4	8	12	16	20	24
Cache size (block)	8	32	72	126	200	288

(b)

Fig. 7. Total search time depending on the dimensionality and cache configuration (Clustered Gaussian Data). (a) Total search time (sec.). (b) Cache configuration.

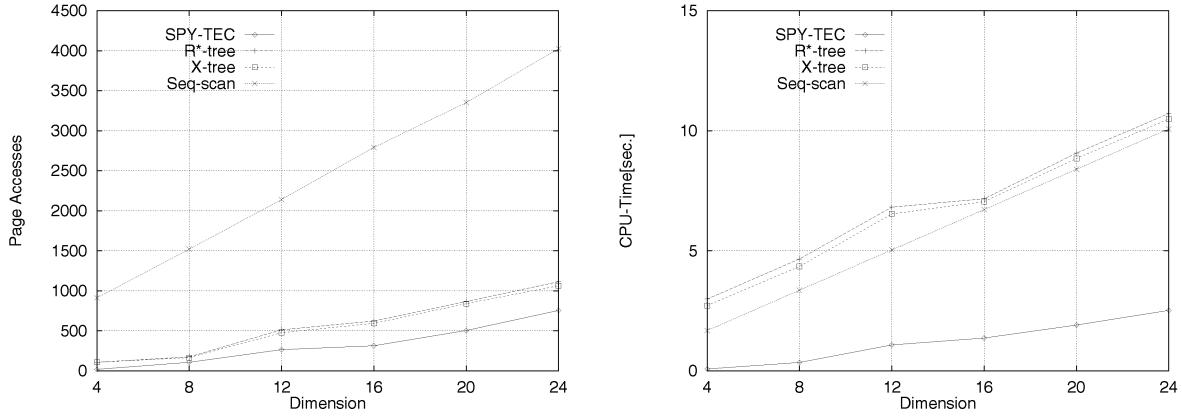
with growing dimensions. In Fig. 8b, the speed-up in CPU time is analogous to that of the number of block accesses, but is higher than it except in the case of the sequential scan. The speed-up in CPU time ranges between 4.25 and 37.38 over the R*-tree, between 4.16 and 34 over the X-tree, and between 4 and 21 over the sequential scan. Although the R*-tree and the X-tree access fewer blocks than does the sequential scan, they use more CPU time. On the other hand, the SPY-TEC always uses less CPU time than other related techniques in all cases. Although the algorithm to process the query in the SPY-TEC seems to be complex when computing the minimum possible distances between the query point and bounding slices, it can compute the distances easily by classifying the cases (in Lemma 2) using simple comparison operations.

In our second experiment on this series, we measured the performance behavior while varying the database size. For this experiment, we varied the database size from 20,000 to 100,000 in a 16-dimensional data space.

Fig. 9 shows the total search time and cache configuration of the index structures used in our experiments. As expected, the SPY-TEC significantly outperforms all the other techniques presented. The speed-up of the SPY-TEC in the total search time seems to be almost constant and ranges between 9.80 and 12.68 over the R*-tree, between 9.44 and 12.65 over the X-tree, and between 1.94 and 2.39 over the sequential scan.

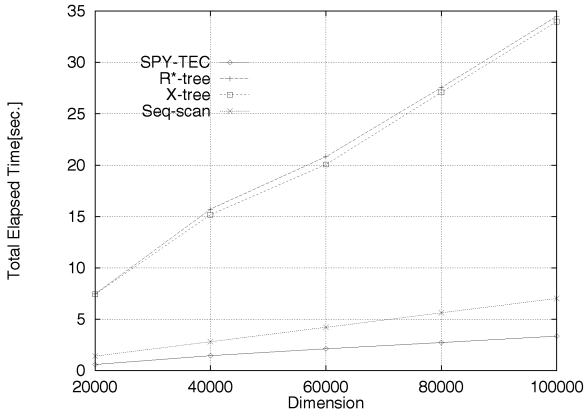
In Fig. 10, we show a more detailed comparison, namely, the number of block accesses and CPU time needed for processing the queries. The performance behavior of the number of block accesses and of CPU time are analogous to that of the total search time except that the R*-tree and the X-tree access fewer blocks than does the sequential scan. In Fig. 10a, the speed-up with respect to the number of block accesses seems to be almost constant and ranges between 1.33 and 1.97 over the R*-tree, between 1.33 and 1.89 over the X-tree, and between 8.16 and 8.89 over the sequential scan. And, as depicted in Fig. 10b, the speed-up in CPU time is higher than that of the number of block accesses except in the case of the sequential scan, but is analogous to it. The speed-up in CPU time seems to be almost constant and ranges between 6.00 and 8.79 over the R*-tree, between 5.94 and 8.46 over the X-tree, and between 5.40 and 6.24 over the sequential scan.

Through the experiments using clustered data, we found that the SPY-TEC significantly outperforms the R*-tree, the X-tree, and the sequential scan in all cases. We also found that, although the R*-tree and the X-tree access fewer blocks than does the sequential scan, they are still beaten by the sequential scan in the total search time. These results may be explained by the fact that the R*-tree tends to have low fanout and a high degree of overlap between bounding regions in higher dimensions, and the X-tree has the overhead of performing disk management operations to create and maintain supernodes produced by the splitting algorithm that minimizes the overlap in the directory



(a) (b)

Fig. 8. Number of block accesses versus CPU time (Clustered Gaussian Data). (a) Block access. (b) CPU time (sec.).

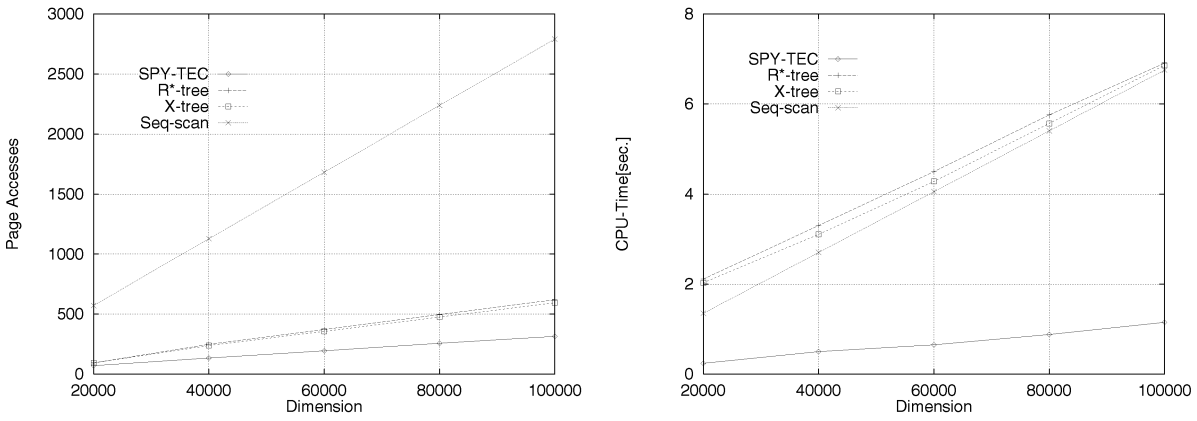


(a)

Dimension	20,000	40,000	60,000	80,000	100,000
Cache size (block)	25	50	75	101	126

(b)

Fig. 9. Total search time depending on the database size and cache configuration (Clustered Gaussian Data). (a) Total search time (sec.). (b) Cache configuration.



(a) (b)

Fig. 10. Number of block accesses versus CPU time (Clustered Gaussian Data). (a) Block access. (b) CPU time (sec.).

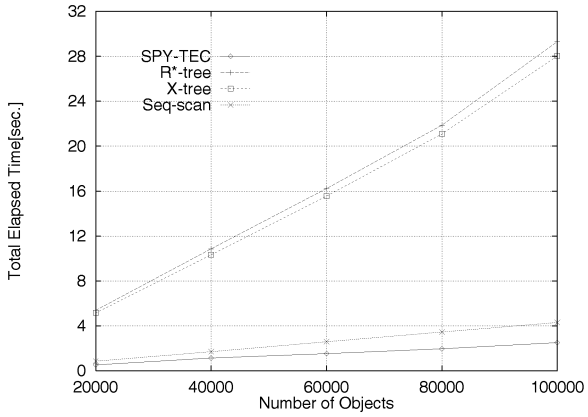
nodes. These drawbacks of the R*-tree or the X-tree degrade the performance of query processing in high-dimensional data spaces. However, since the SPY-TEC uses a data space partitioning strategy which produces no overlap in the index nodes, and can take advantage of all of the benefits of a B^+-tree, the nearest-neighbor algorithm of the SPY-TEC yields better performances than those of other related techniques such as the X-tree and the R*-tree in higher dimensions as well as lower dimensions.

5.2 Real Data

Since one may argue that synthetic databases such as uniformly distributed data are not realistic in high-dimensional data spaces, we also used real data in our experiment. To show the practical impact of our technique for real data sets, we performed experiments using 12-dimensional Fourier points [26] which correspond to contours of industrial parts. As in the previous experiments, we performed 10-nearest-neighbor queries with 100 query points that were selected from the real data itself, and varied the database size from 20,000 to 100,000.

Fig. 11 shows the total search time of the experiment using real data sets. In this experiment, we observed a similar result to that of the experiments using clustered Gaussian data sets. That is, the SPY-TEC significantly outperforms all the other techniques including the sequential scan regardless of the database size.

From this result, we found that the real data consists of well-formed clusters which are meaningful workloads for high-dimensional nearest-neighbor queries. The speed-up of the SPY-TEC in the total search time seems to be almost constant and ranges between 9.52 and 11.69 over the R*-tree, between 9.05 and 11.18 over the X-tree, and between 1.49 and 1.76 over the sequential scan.

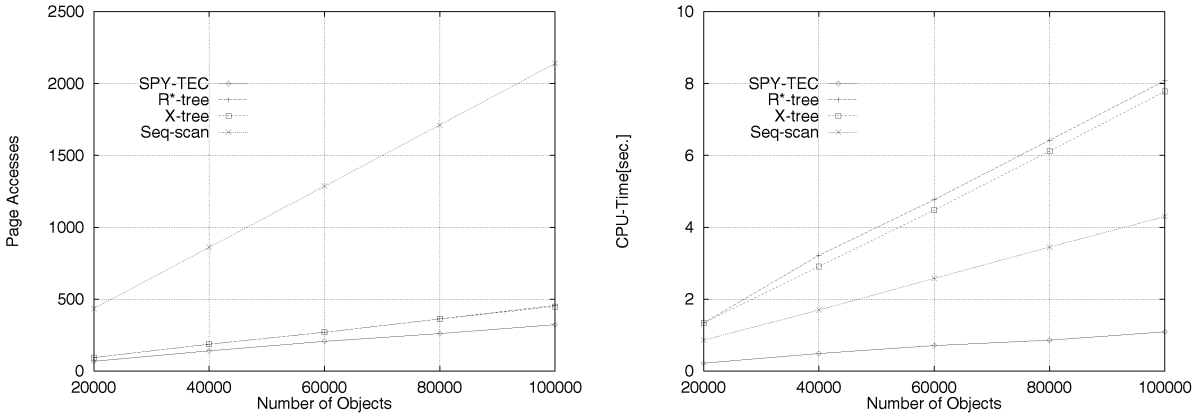


(a)

Dimension	20,000	40,000	60,000	80,000	100,000
Cache size (block)	14	29	43	58	72

(b)

Fig. 11. Total search time and cache configuration (Real Data). (a) Total search time (sec.). (b) Cache configuration.



(a) (b)

Fig. 12. Number of block accesses versus CPU time (Real Data). (a) Block access. (b) CPU time.

In Fig. 12, we show the number of block accesses and CPU time for a more detailed comparison. The performance behavior of the number of block accesses and of CPU time are analogous to that of the total search time except that the R*-tree and the X-tree access fewer blocks than does the sequential scan.

In Fig. 12a, the speed-up with respect to the number of block accesses seems to be almost constant and ranges between 1.30 and 1.41 over the R*-tree, between 1.30 and 1.39 over the X-tree, and between 6.11 and 6.63 over the sequential scan. The speed-up in CPU time is analogous to that of the number of block accesses, but is higher than it except in the case of the sequential scan. The speed-up in CPU time ranges between 6.09 and 7.47 over the R*-tree, between 5.94 and 7.14 over the X-tree, and between 3.47 and 4.01 over the sequential scan.

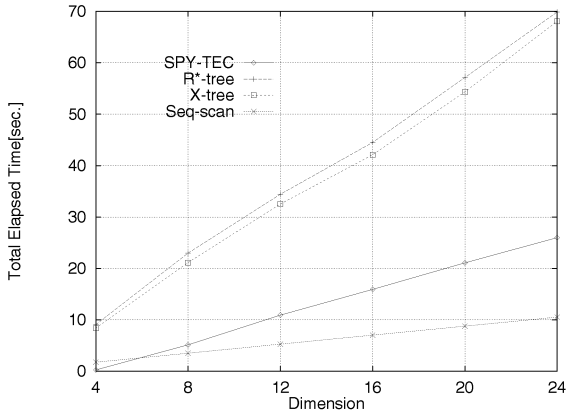
Through the experiments using real data, we found that the SPY-TEC significantly outperforms all the other techniques presented in all cases.

5.3 Uniform Data

Uniform data sets were created for 4-24 dimensions. Each data sets consisted of 20,000 - 100,000 points that were uniformly generated in the range [0, 1] for each dimension. For each uniform data set, we performed 10-nearest-neighbor queries with 100 query points that were taken from within the data set itself.

In our first experiment, we measured the performance behavior while we varied data space dimension. For this experiment, we created six files with the dimensionalities 4, 8, 12, 16, 20, and 24. The database size was set to 100,000 points.

Fig. 13 shows the total search time and cache configuration of the index structures used in our experiments. As expected, the search time of all of the index structures increases with growing dimension. For all cases, the sequential scan outperforms the R*-tree and the X-tree. This is not a surprising result. In uniformly distributed data



(a)

Dimension	4	8	12	16	20	24
Cache size (block)	8	32	71	126	203	301

(b)

Fig. 13. Total search time depending on the dimensionality and cache configuration (Uniform Data). (a) Total search time (sec.). (b) Cache configuration.

and query points, as dimensionality increases, the difference in distance between the nearest neighbor and any other point in the data set becomes very small [21]. Thus, most index structures have to access most of the blocks and compute the distance between the query point and almost all points in the data set for higher dimensional data spaces. Although the SPY-TEC outperforms the sequential scan in lower dimensions (under eight dimensions), it also does not yield a better performance than the sequential scan in high-dimensional data spaces.

In Fig. 14, we show a more detailed comparison, namely the number of block accesses and CPU time needed for processing the queries. As depicted in Fig. 14a, the index structures including the SPY-TEC, the R*-tree, and the X-tree access fewer blocks than does the sequential scan. However, they execute the query with random I/O's. Thus, they have the heavy penalty associated with doing random I/O's as opposed to sequential I/O's. Therefore, although the index structures including the SPY-TEC access fewer blocks, they do not yield a better performance than the sequential scan in the total search time.

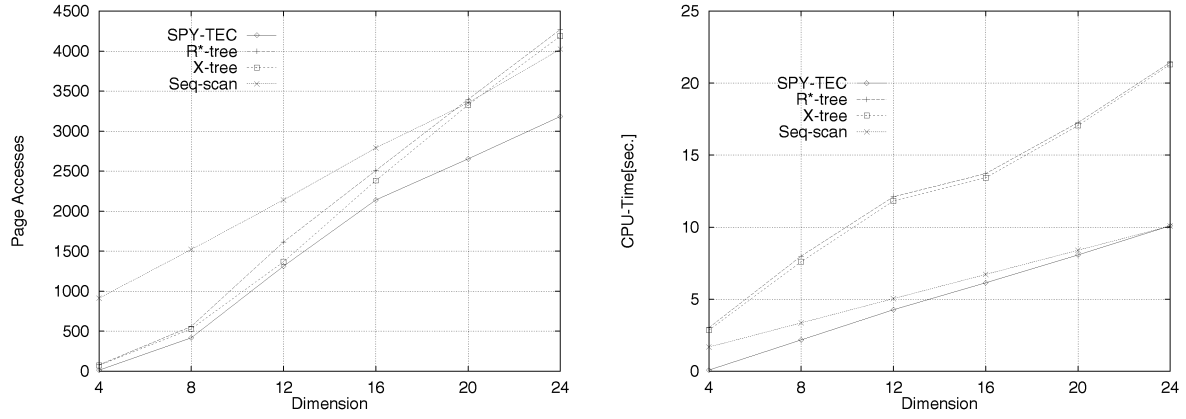
However, as we will discuss in Section 5.4, these results should not be interpreted to mean that high-dimensional nearest-neighbor queries on the SPY-TEC is never meaningful. The important fact is that the SPY-TEC significantly outperforms competitive index structures such as the R*-tree and the X-tree. It turned out that both the number of block accesses and CPU time of the SPY-TEC are better than those of the R*-tree, the X-tree, and the sequential scan. When the dimension is below 20, the R*-tree and the X-tree access fewer blocks than does the sequential scan. However, they have to access more blocks than the sequential scan when the dimension is above 20. This effect may be explained by the fact that the R*-tree and the X-tree have to access most leaf nodes as well as most internal nodes in higher dimensions.

Through the experiments using uniform data, we found that, although the SPY-TEC does not yield a better performance than the sequential scan in the total search time, it yields a better performance than all the other techniques including the sequential scan in the number of block accesses and CPU time. It also significantly outperforms competitive index structures such as the R*-tree and the X-tree in all cases.

5.4 Summary of Experiments

A number of conclusions can be drawn from these experiments. First, in the experiments on uniform data, although the SPY-TEC significantly outperforms other related techniques such as the R*-tree and the X-tree in higher dimensions as well as lower dimensions, it still does not yield a better performance than the sequential scan except in lower dimensions (under eight dimensions). In this experiment, the SPY-TEC accesses fewer blocks and uses less CPU time than does the sequential scan. However, it executes the query with random I/O's that may be a heavy penalty in total running time as the R-tree or the X-tree does.

However, as mentioned above, the results of the experiments on uniform data should not be interpreted to



(a) (b)

Fig. 14. Number of block accesses versus CPU time (Uniform Data). (a) Block access. (b) CPU time (sec.).

mean that high-dimensional nearest-neighbor queries on the SPY-TEC is never meaningful. According to [21], most high-dimensional index techniques including the SS-tree, the SR-tree, etc., are usually beaten by the sequential scan in high-dimensional data space. Thus, the authors of [21] argued that there exist situations in which high-dimensional nearest-neighbor queries are meaningful and that the evaluations of nearest-neighbor techniques should focus on those situations in which index-based query processing may be more efficient than the sequential scan.

In fact, a uniform data set is not realistic in high-dimensional data spaces and unsuitable for evaluating the performance of index structures in high dimensionality [21], [28]. We observed this point by identifying some high-dimensional workloads for which the SPY-TEC significantly outperforms other related techniques including the sequential scan. In the experiments using clustered Gaussian data and real data, we found that the SPY-TEC significantly outperforms the R*-tree, the X-tree, and the sequential scan in all cases.

Finally, we also observed that, although the R*-tree and the X-tree access fewer blocks than does the sequential scan in clustered Gaussian data and real data having high dimensionality, they still need a lot of CPU time. Therefore, they do not yield a good performance in high-dimensional data spaces regardless of any workloads used in our experiments.

6 Conclusions

The SPY-TEC is based on a special partitioning strategy which divides the d-dimensional data space first into $2d$ spherical pyramids and then cuts each spherical pyramid into several bounding slices. In this paper, we proposed the incremental nearest-neighbor algorithm on the SPY-TEC. We also introduced a metric that can be used to guide an ordered best-first traversal when finding nearest neighbors on the SPY-TEC. The metric (MINDIST), the minimum possible distance of the query point from a spherical pyramid or a bounding slice, produces the most optimistic ordering possible when finding nearest neighbors on the SPY-TEC. We implemented the incremental nearest-neighbor algorithm on the SPY-TEC and performed extensive experiments using clustered Gaussian data, real data, and uniform data sets, to show the practical impacts of these algorithms. Through the experiments, we showed that the incremental algorithm on the SPY-TEC clearly outperforms that of the X-tree, the R*-tree, and the sequential scan in clustered data and real data sets.

Recently, in many applications that require the nearest-neighbor algorithm in high-dimensional data spaces, fast searching is a much more important issue than exact searching. Therefore, algorithms that return an $approximate$ result rather than an $exact$ result have been developed, thereby saving search time in computing it [12]. Arya et al. [23], [24] proposed the concept of the $(1+\varepsilon)$-approximate nearest-neighbor search for this purpose. This concept is that, given any positive real ε, a data point p is a $(1+\varepsilon)$-approximate nearest neighbor of the query point q if its distance from q is within a factor of $(1+\varepsilon)$ of the distance to the true nearest neighbor. They also generalized the approximate nearest-neighbor procedure to the problem of computing approximations to the k nearest neighbors of a query point. In [24], it was found that in moderate dimensions, significant savings in running time can be achieved by computing the approximate k nearest neighbors as opposed to the exact k nearest neighbors. Moreover, with relatively high probability, the result is the same in the exact and approximate cases. We can easily extend Algorithm 1 to provide the approximate nearest-neighbor search on the SPY-TEC. The only change required to Algorithm 1 to make it the approximate nearest-neighbor search is in the key used for the spherical pyramids and bounding slices in the priority queue. That is, before enqueueing spherical pyramids and bounding slices, we multiply their distances by $(1+\varepsilon)$. Actually, we implemented the approximate nearest-neighbor algorithm on the SPY-TEC. And, through the experiments, we found that there are significant reductions in block accesses and CPU time when finding the approximate k nearest neighbors as opposed to the exact k nearest neighbors.

As depicted in Fig. 3 or Fig. 4, the outer slice, that is the farthest slice from the center, is not topologically equivalent to a usual slice. The outer slice of a spherical pyramid has 2^{d-1} corners in a d-dimensional data space. And, the corners that belong to the outer slice are not closely located with one another and they don't even touch. However, as mentioned in Section 3.1, the points in these corners are stored in the same data page because a slice as well as the outer slice corresponds to one data page of the B^+-tree. Also, in a d-dimensional data space, one corner consists of d outer slices that are adjacent. Thus, although the points in one corner are closely located with one another, they may fall into d different outer slices.

These topological features of the outer slice degrade the performance of processing k-nearest-neighbor queries on the SPY-TEC. That is, if the query sphere is large and includes a corner in a d-dimensional space, all of the outer slices of d spherical pyramids must be accessed for processing the query. We think that our technique may perform worse than other index structures for highly skewed data distributions or queries toward the corners or axes of the data spaces.

However, none of the index structures proposed so far can handle highly skewed data or queries efficiently [25]. We plan to address the problem of handling highly skewed data or queries in our future work. We also plan to study the parallel version of the nearest-neighbor algorithm on the SPY-TEC using an efficient declustering technique that distributes the data onto the disks so that the data which has to be read when executing a query are distributed as equally as possible among the disks.

Acknowledgments

This research was partially funded by the 1999 BK21 IT area grant of the Ministry of Education in Korea.

References

[1] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 47-57, June 1984.

[2] A. Henrich, "The LSDh-Tree: An Access Structure for Feature Vectors," *Proc. 14th Int'l Conf. Data Eng.*, pp. 362-369, 1998.

[3] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equiz, "Efficient and Effective Querying by Image Content," *J. Intelligent Information System (JIIS),* vol. 3, no. 3, pp. 231-262, July 1994.
[4] B.C. Ooi, K.L. Tan, T.S. Chua, and W. Hsu, "Fast Image Retrieval Using Color-Spatial Information," *The VLDB J.,* vol. 7, no. 2, pp. 115-128, 1998.
[5] C. Böhm, "A Cost Model for Query Processing in High-Dimensional Data Spaces," *ACM Trans. Database Systems,* vol. 25, no. 2, pp. 129-178, 2000.
[6] C.E. Jacobs, A. Finkelstein, and D.H. Salesin, "Fast Multiresolution Image Query," *Proc. 1995 ACM SIGGRAPH Int'l Conf. Computer Graphics and Interactive Techniques,* 1995.
[7] D.A. White and R. Jain, "Similarity Indexing: Algorithms and Performance," *Proc. SPIE Storage and Retrieval for Image and Video Databases IV,* vol. 2670, pp. 62-75, 1996.
[8] D.A. White and R. Jain, "Similarity Indexing with the SS-Tree," *Proc. 12th Int'l Conf Data Eng.,* pp. 516-523, 1996.
[9] D.B. Lomet and B. Salzberg, "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance," *ACM Trans. Database Systems,* vol. 15, no. 4, pp. 625-658, 1990.
[10] D.H. Lee and H.J. Kim, "SPY-TEC: An Efficient Indexing Method for Similarity Search in High-Dimensional Data Spaces," *Data & Knowledge Eng.,* vol. 34, no. 1, pp. 77-97, 2000.
[11] C. Faloutsos, "Fast Searching by Content in Multimedia Databases," *Data Eng. Bull.,* vol. 18, no. 4, 1995.
[12] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems,* vol. 24, no. 2, pp. 265-318, 1999.
[13] J. Bentley, "Mutidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM,* vol. 18, no. 9, pp. 509- 517, 1975.
[14] J. Nievergelt, H. Hinterberger, and K.C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. Database Systems,* vol. 9, no. 1, pp. 38-71, Mar. 1984.
[15] J.R. Smith and S.-F. Chang, "VisualSEEk: A Fully Automated Content-Based Image Query System," *Proc. ACM Multimedia 96,* 1996.
[16] J.T. Robinson, "The k-d-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 10-18, Apr. 1981.
[17] K.-I. Lin, H.V. Jagadish, and C. Faloutsos, "The TV-Tree: An Index Structure for High-Dimensional Data," *The VLDB J.,* vol. 3, no. 4, pp. 517-542, 1994.
[18] L. Leithold, *Trigonometry.* Addison-Wesley, 1989.
[19] N. Katayama and S. Satoh, "The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 517-542, May 1997.
[20] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 71-79, 1995.
[21] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is 'Nearest Neighbor' Meaningful?" *Proc. Seventh Int'l Conf. Database Theory,* pp. 217-235, Jan. 1999.
[22] R. Weber, H.-J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. 24nd Int'l Conf. Very Large Database,* pp. 194-205, Aug. 1998.
[23] S. Arya, D.M. Mount, N. Netanyahu, R. Silverman, and A.Y. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions," *Proc. Fifth ACM-SIAM Symp. Discrete Algorithms,* pp. 573-582, 1994.
[24] S. Arya, D.M. Mount, N. Netanyahu, R. Silverman, and A.Y. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions," *J. ACM,* vol. 45, no. 6, pp. 891-923, 1998.
[25] S. Berchtold, C. Böhm, and H.-P. Kriegel, "The Pyramid-Technique: Towards Breaking the Curse of Dimensionality," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 1998.
[26] S. Berchtold, C. Böhm, B. Braunmuller, D.A. Keim, and H.-P. Kriegel, "Fast Parallel Similarity Search in Multimedia Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 1-12, 1997.
[27] S. Berchtold, C. Böhm, D.A. Keim, and H.-P. Kriegel, "A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space," *Proc. ACM PODS Symp. Principles of Database Systems,* 1997.
[28] S. Berchtold, D.A. Keim, and H.-P. Kriegel, "The X-Tree: An Indexing Structure for High-Dimensional Data," *Proc. 22nd Int'l Conf. Very Large Database,* pp. 28-39, Sept. 1996.
[29] S. Berchtold, D.A. Keim, H.-P. Kriegel, and T. Seidl, "Fast Nearest Neighbor Search in High-Dimensional Spaces," *Proc. 14th Int'l Conf. Data Eng.,* 1998.
[30] P.M. Kelly, T.M. Cannon, and D.R. Hush, "Query by Image Example: The CANDID Approach," *Proc. SPIE Storage and Retrieval for Image and Video Databases III,* vol. 2420, pp. 238-248, 1995.

Dong-Ho Lee received the BS degree from Hong-Ik University and the MS degree in computer engineering from Seoul National University, Seoul, Korea, in 1995 and 1997, respectively. He is currently enrolled in the PhD degree program in computer engineering at Seoul National University. His research interests include high-dimensional index techniques, content-based retrieval system, multimedia databases, and object-oriented databases.

Hyoung-Joo Kim received the BS degree in computer engineering from Seoul National University, Seoul, Korea, in 1982 and the MS and PhD degrees in computer engineering from University of Texas at Austin in 1985 and 1988, respectively. He was an assistant professor at the Georgia Institute of Technology and is currently a professor in the Department of Computer Engineering at Seoul National University. His research interests include object-oriented databases, multimedia databases, HCI, and computer-aided software engineering. He is a member of the IEEE.