Indexing High-Dimensional Data for Efficient In-Memory Similarity Search

Bin Cui, Beng Chin Ooi, *Member*, *IEEE*, Jianwen Su, *Senior Member*, *IEEE*, and Kian-Lee Tan, *Member*, *IEEE Computer Society*

Abstract—In main memory systems, the L2 cache typically employs cache line sizes of 32-128 bytes. These values are relatively small compared to high-dimensional data, e.g., $> 32D$. The consequence is that existing techniques (on low-dimensional data) that minimize cache misses are no longer effective. In this paper, we present a novel index structure, called $\Delta\text{-tree}$, to speed up the high-dimensional query in main memory environment. The $\Delta\text{-tree}$ is a multilevel structure where each level represents the data space at different dimensionalities: the number of dimensions increases toward the leaf level. The remaining dimensions are obtained using *Principal Component Analysis*. Each level of the tree serves to prune the search space more efficiently as the lower dimensions can reduce the distance computation and better exploit the small cache line size. Additionally, the top-down clustering scheme can capture the feature of the data set and, hence, reduces the search space. We also propose an extension, called $\Delta^+\text{-tree}$, that globally clusters the data space and then partitions clusters into small regions. The $\Delta^+\text{-tree}$ can further reduce the computational cost and cache misses. We conducted extensive experiments to evaluate the proposed structures against existing techniques on different kinds of data sets. Our results show that the $\Delta^+\text{-tree}$ is superior in most cases.

Index Terms—High-dimensional index, main memory, similarity query.

1 Introduction

With an increasing number of new database applications, such as multimedia content-based retrieval, time series, and scientific databases, the design of efficient indexing and query processing techniques over high-dimensional data sets becomes an important research area. These applications employ the so-called feature transformation, which transforms important features or properties of data objects into high-dimensional points, i.e., each feature vector consists of d values, which correspond to coordinates in a d-dimensional space. Searching for objects based on these features is, thus, a search of points in this feature space. In these high-dimensional databases, indexes are required to support either or all of the following frequently used queries:

- *K-nearest neighbor (KNN) queries*: find the K-most similar objects in the database with respect to a given object,
- *similarity range queries*: find all objects in the database which are within a given distance from a given point,
- *window queries*: find all objects whose attribute values fall within certain given ranges. We can regard the *window query* as a *similarity range query* around the center point of the given query range.

- *B. Cui, B.C. Ooi, and K.-L. Tan are with the Department of Computer Science, and Singapore-MIT Alliance, National University of Singapore, 3 Science Drive 2, Singapore 117543. E-mail: {cuibin, ooibc, tankl}@comp.nus.edu.sg.*
- *J. Su is with the Department of Computer Science, University of California, Santa Barbara, CA 93106-5110. E-mail: su@cs.ucsb.edu.*

Manuscript received 6 Oct. 2003; revised 14 May 2004; accepted 25 Aug. 2004.
For information on obtaining reprints of this article, please send e-mail to:

There is a long stream of research on solving the similarity search problem and many multidimensional indexes have been proposed [3], [6], [7], [14], [18], [23]. However, these index structures have largely been studied in the context of disk-based systems where it is assumed that the databases are too large to fit into the main memory. This assumption is increasingly being challenged as RAM gets cheaper and larger. This has prompted renewed interest in research in main memory databases [5], [9], [12], [19], [21].

As random access memory gets cheaper, it becomes increasingly affordable to build computers with large main memories, but main memory data processing is not as simple as increasing the buffer pool size. In main memory systems, minimizing L2 cache misses and computation cost has been an active area of research. Several main memory indexing schemes have been designed to be *cache conscious* [12], [19], [21], i.e., these schemes page the structure based on cache blocks whose sizes are usually 32-128 bytes. However, these schemes are targeted at single or low-dimensional data (that fit in a cache line) and cannot be effectively deployed for high-dimensional data. First, for high-dimensional data, the query processing is computationally expensive, which involves large amounts of distance calculation [7]. Second, the size of a high-dimensional point (e.g., 256 bytes for 64 dimensions) can be much larger than a typical L2 cache line size. Actually, [19] shows that, even for 2-dimensional data, the optimal node size for a similarity search can be up to 256-512 bytes and increases as the dimensionality increases. Therefore, an efficient main memory index should minimize the distance computation to improve the performance and also exploit the L2 cache effectively as well.

Our proposed novel multitier index structure, the Δ-tree, can facilitate efficient KNN searches in a main memory environment. Each tier in the Δ-tree represents the data space as clusters in a different number of dimensions and tiers closer to the root partition the data space using fewer number of dimensions. The numbers of tiers and dimensions are obtained using the Principal Component Analysis (PCA) technique [17]. After PCA transformation, the first few dimensions of the new data space generally capture most of the information and, in particular, two points that are distance d_i apart in i dimensions have the property that $d_i \leq d_j$ if $i \leq j$. More importantly, by hierarchical clustering and reducing the number of dimensions, we can decrease the distance computation and better utilize the L2 cache. An extension of the Δ-tree, called Δ^+-tree, is also proposed to further reduce the search space. The Δ^+-tree globally clusters the data space, partitions clusters into small regions, and, finally, builds a subtree for each region. The global clustering and cluster partitioning can significantly reduce the search space. Additionally, the PCA transformation is applied on each cluster separately and, hence, the reduced dimensions can represent the data point more precisely. We also present the tree construction, KNN searches, range query, and insertion and deletion algorithm for the Δ-tree. We compare the proposed schemes against other known schemes, including the TV-tree [20], iDistance [23], Slim-tree [18], Omni-technique [14], Pyramid-tree [3], and Sequential Scan. Our experimental study shows that the Δ^+-tree is superior to the other indexes.

A preliminary version of this paper appeared in [11], where we presented the basic idea of the Δ^+-tree. In this paper, we make the following additional contributions: First, we provide a more detailed description of the index method. Second, we extend the algorithms to support insertion, deletion, and different kinds of queries. Third, we run a comprehensive set of experiments to demonstrate the effectiveness of the Δ^+-tree, such as parameter tuning, KNN, and range query performance. Furthermore, we evaluate Δ^+-tree against a large number of competitive index structures.

The remainder of this paper is organized as follows: In the next section, we provide some background of our work. In Section 3, we introduce the structure and algorithms of our newly proposed Δ-tree and Δ^+-tree. Section 4 reports the findings of an extensive experimental study conducted to evaluate the proposed schemes and, finally, we conclude in Section 5.

2 Preliminary

2.1 Multidimensional Index

Multidimensional access methods can be classified into two broad categories based on the data type that they have been designed to support: Point Access Method (PAM) and Spatial Access Method (SAM). While PAMs have been designed to support queries on multidimensional points, SAMs are designed to support multidimensional objects with spatial extends and can function as PAMs. Many multi/high-dimensional index structures have been proposed to support applications in spatial and scientific databases, such as the M-tree [10], the VA-file [22], the CR-tree [19], etc. In this section, we will introduce and briefly discuss the index structures that are used for comparison in our experimental study.

2.1.1 The TV-Tree

In [20], the authors proposed a tree structure that avoids the dimensionality curse problem. The idea is to use a variable number of dimensions for indexing, adapting to the number of objects, and to the current level of the tree. The TV-tree structure bears some similarity to the R-tree, but defines inactive dimensions and active dimensions. The TV-tree only indexes the active dimensions and the number of active dimensions is usually small, thus, the method saves space and leads to a larger fanout. As a result, the tree is more compact and performs better than the R^*-tree. Although our proposed approaches also employ fewer dimensions in the internal node, they differ from the TV-tree in several ways. First, the TV-tree uses the same number of active dimensions at every level of the tree, while our schemes use a different number of dimensions at different levels and always employ few dimensions in the upper levels. Second, the TV-tree exploits the same value in a dimension in picking the active dimension and it is designed for string data sets (as demonstrated in the paper). Third, even though the TV-tree's internal nodes have fewer dimensions, the algorithm is the same as the R-tree-based algorithm. On the other hand, our proposed schemes exploit clustering to construct the tree and take advantage of PCA to prune the search space more effectively.

2.1.2 The Slim-Tree

In [18], the authors proposed a new dynamic tree for organizing metric data sets, named Slim-tree. The Slim-tree uses the triangle inequality to prune distance calculations needed to answer similarity queries over objects in metric spaces, which is similar to the M-tree. The proposed insertion algorithm uses new policies to select the nodes where incoming objects are stored. When a node overflows, the Slim-tree uses a Minimal Spanning Tree to help with the split. The new insertion algorithm leads to a tree with high storage utilization and improved query performance. The Slim-tree tackles the overlap problem between nodes and proposes "fat-factor" method to minimize it. The authors also presented a new visualization tool for interactive data mining and for the visual tracking of the behavior of a tree under updates. Although the Slim-tree performs similarly for distance calculations compared with the M-tree, it reduces the number of node accesses and, hence, has better performance.

2.1.3 The iDistance

The iDistance [23] was presented as an efficient method for KNN searches in a multidimensional space. iDistance partitions the data and selects a reference point for each partition. The data points in each cluster are transformed into a single-dimensional space based on their similarity with respect to a reference point. It then indexes the distance of each data point to the reference point of its partition. Since this distance is a simple scalar, with a small mapping effort to keep partitions distinct, it is possible to use a standard B^+-tree structure to index the data and the KNN search can be performed using a one-dimensional range search. The choice of partition and reference point provides the iDistance technique with degrees of freedom most other techniques do not have. The iDistance technique can permit the immediate generation of a few results while additional results are searched for. In other words, it is able to support online query answering, an important facility for interactive querying and data analysis. Since the cache conscious B^+-tree has been studied [21], and distance is a single-dimensional attribute (that fits into the cache line), iDistance is expected to be a promising candidate for main memory systems.

2.1.4 The Pyramid-Tree

The basic idea of the Pyramid Technique [3] is to transform the D-dimensional data points into 1-dimensional values and then store and access the values using a conventional index, such as the B^+-tree. It splits the data space into 2D pyramids, which share the center point of the data space as their top and have a (D-1)-dimensional surface of the data space as their base. The location of a point within its pyramid is indicated by a single value, which is the distance from the point to the center point according to dominant dimension. To perform a range query, the pyramids that intersect the search region are first obtained and, for each pyramid, a subquery range is worked out. Each subquery is then used to search the B^+-tree. For each range query, 2D subqueries may be required, one against each pyramid.

2.1.5 The Omni-Technique

The Omni-technique [14] was proposed to be used with existing index structures and reduce the distance computational cost. The Omni-technique chooses a number of objects from the data set as "foci," from which distance calculations to answer queries can be pruned. The foci can be used as global reference points to any object in the database and improve any underlying index structure. The authors also proposed a method to define and select an adequate number of objects to be used as Omni-foci, with the best tradeoff between increasing memory requirements and decreasing distance computations. By applying the Omni-concept with sequential scan and R-tree, they can achieve up to 10 times fewer calculations, disk accesses, and overall time.

2.2 Principal Component Analysis

The Principal Component Analysis (PCA) [17] is a widely used method for transforming points in the original (high-dimensional) space into another (usually lower-dimensional) space [8], [16]. It examines the variance structure in the data set and determines the directions along which the data exhibits high variance. The first principal component (or dimension) accounts for as much of the variability in the data as possible and each succeeding component accounts for as much of the remaining variability as possible. Using PCA, most of the information in the original space is condensed into a few dimensions along which the variances in the data distribution are the largest. A single example is illustrated in Fig. 1 where a single dimension (the first PC) can capture the variation of data in the 2D space after PCA processing. In such cases, it is possible to eliminate some dimensions (the later PCs) with little loss of distance information.

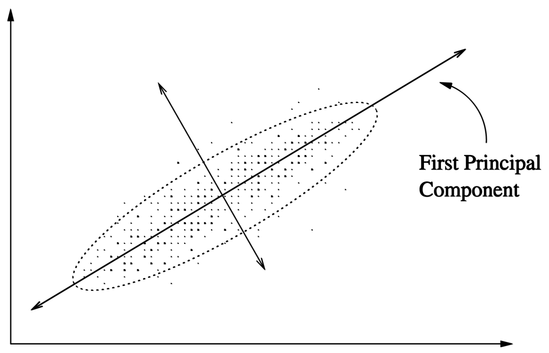

Fig. 1. Illustration of PCA.

We briefly review how the principal components are computed. Let the data set contain N D-dimensional points. Let A be the $N \times D$ data matrix where each row corresponds to a point in the data set. We first compute the mean and covariance matrix of the data set to get the *eigenmatrix*, V, which is a $D \times D$ matrix. The first principal component is the eigenvector corresponding to the largest eigenvalue of the variance-covariance matrix of A, the second component corresponds to the eigenmatrix with the second largest eigenvalue, and so on.

The second step is to transform the data points into the new space. This is achieved by multiplying the vectors of each data point with the *eigenmatrix*. More formally, a point $\mathrm{P}(x_1, x_2, \ldots, x_D)$ is transformed into $V \times P = (y_1, y_2, \ldots, y_D)$. To reduce the dimensionality of a data set to k, $0 < k < D$, we only need to project out the first k dimensions of the transformed points. The mapping (to reduced dimensionality) corresponds to the well-known Singular Value Decomposition (SVD) of data matrix A and can be done in $\mathrm{O}(N \cdot D^2)$ time [15].

Suppose we have two points, P and Q, in the data set in the original D-dimensional space. Let P_{k1} and P_{k2} denote the transformed points of P projected on $k1$ and $k2$ dimensions, respectively (after applying PCA), $0 < k1 < k2 \leq D$. Q_{k1} and Q_{k2} are similarly defined. The PCA method has several nice properties:

1. $$dist(P_{k1}, Q_{k1}) \leq dist(P_{k2}, Q_{k2}) 0 < k1 < k2 \leq D,$$
   where $dist(p, q)$ denotes the distance between two points, p and q (see [8] for a proof).
2. Because the first few dimensions of the projection are the most important, $dist(P_k, Q_k)$ can be very near to the actual distance between P and Q for $k \ll D$ [8].
3. The above properties also hold for new points that are added into the data set (despite the fact that they do not contribute to the derivation of the *eigenmatrix*) [8]. Thus, when a new point is added to the data set, we can simply apply the *eigenmatrix* and map the new data from the original space into the new PCA-

space. Note that the effectiveness of the *eigenmatrix* may degrade after a large number of insertions.

In [8], PCA is employed to organize the data into clusters and find the optimal number of dimensions for each cluster. Our work applies PCA differently. We use it to facilitate the design of an index structure that allows pruning at different levels with different number of dimensions. This can reduce the computational overhead and L2 cache misses.

3 The Δ-tree

Handling high-dimensional data has always been a challenge to the database research community because of the dimensionality curse. In main memory databases, the curse has taken a new twist: a high-dimensional point may not fit into the L2 cache line, whose size is typically 32-128 bytes. Additionally, the distance computation of high-dimensional query occupies a large portion of the overall cost in the absence of disk I/O. As such, existing indexing schemes are not adequate in handling high-dimensional data. In this section, we present a new index structure, called Δ-tree, to facilitate fast query processing in main memory databases. For the rest of this paper, we assume that the data set consists of D-dimensional points and use the Euclidean distance as the metric distance function.

3.1 The Index Structure

The proposed structure is based on three key observations. First, dimensionality reduction is an important technique to deal with the dimensionality curse. In particular, by reducing the dimensionality of a high-dimensional point, it is possible to "squeeze" it into the cache line. Second, ascertaining the number of dimensions to reduce to is a nontrivial task. In addition, even if we can decide on the number of dimensions, it is almost impossible to identify the dimensions to be retained for optimal performance. Third, PCA offers a very good solution: the first component captures the most dominant information of points, the second the next most dominant, and so on. Moreover, as discussed in Section 2.2, it has several very nice properties.

3.1.1 The Structure of Δ-tree

Consider a data set of D-dimensional points. Suppose we apply PCA on the data set to transform the points into a new space that is also D-dimensional. We refer to the transformed space as PCA-Space. Consider a data point P in the PCA-Space, say $(x_1, \dots, x_D)$. We define $\prod(P, m)$ to be an operator that *projects* point P on its first m dimensions ($2 \leq m \leq D$):

$$\prod((x_1, \dots, x_D), m) = (x_1, \dots, x_m).$$

Fig. 2 shows a Δ-tree, which is essentially a multitier tree. The data space is split into clusters and the tree directs the search to the relevant clusters. However, the indexing keys at each level of the tree are different—nodes closer to the root have keys with fewer dimensions and the keys at the leaves are in the full dimensions of the data. We discuss how the number of levels of the tree and the number of dimensions to be used at each level can be determined shortly. For the moment, we assume that the tree has L levels and the number of dimensions at level k is m_k, $1 \leq k \leq L$, $m_i < m_j$ for $i < j$. Moreover, we note that the m_i dimensions selected for level i are given by $\prod((x_1, \dots, x_D), m_i)$, i.e., first m_i dimensions of points are used for indexing at level i.

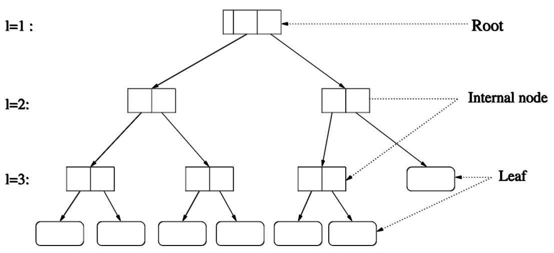

Fig. 2. The Δ-tree.

In the Δ-tree, the data is recursively split into smaller clusters at each level. This is done as follows: At level 1 (root), the data is partitioned into n clusters $C_1, C_2, \dots C_n$. We employ a clustering algorithm for this purpose and, in our implementation, we use the K-means scheme. The clustering is, however, performed using the m_1 dimensions in the PCA-Space of the transformed data. In other words, C_i contains points that are clustered together in m_1 dimensions in the PCA-Space. At level 2, C_i is partitioned into $C_{i1}, C_{i2}, \dots, C_{in}$ subclusters using the m_2 dimensions of the points in C_i in the PCA-Space. This process is repeated for each subcluster at each subsequent level l where each subsequent clustering process operates on the m_l dimensions of the points in the subcluster in the PCA-Space. At the leaf level (level L), the full dimensions in the original space are used as the indexing key, i.e., the leaf nodes correspond to clusters of the actual data points.

An internal node at level l contains information of the cluster it covers at m_l dimensions and each entry corresponds to information of a subcluster. Each entry is a 4-tuple (c_l, r, num, ptr), where c_l is the center of the subcluster obtained at level l, r is the radius of the subcluster, num is the number of points in the subcluster, and ptr is a pointer to the next level node. The root node has the same structure as an internal node except that it has to maintain additional information on the data set. This is captured as a triple (L, m, $eigenmatrix$) header, where L represents the number of projection levels, $m = (m_1, m_2, \dots, m_{L-1})$ is a vector of size $L-1$ representing the number of dimensions in each projection level (excluding the last level which stores the full dimensions of points), and $eigenmatrix$ is the eigenmatrix of data set after PCA processing.

We note that the Δ-tree can be used to prune the search space effectively. Recall (in Property 1 of PCA) that the distance between two points in a low dimensionality in the PCA-Space is always smaller than the distance between the two points in a higher dimensionality. In the Δ-tree, the distances between the clusters in the upper levels are usually larger. Thus, we can use the distance at low dimensionality to prune away points that are far away (i.e., if the distance between a database point and the query point at low dimensionality is larger than the real distance of the current Kth NN, then it can be pruned away). More importantly, the lower dimensionality at upper levels of the

```
Algorithm Buildtree(dataset, tree)

1. $pC = PCA(dataset)$;
2. Init(root);
3. R_insert(root, $pC$, 1);

R_insert(node, pC, lev)
1. pClusters = LevCluster($pC$, K, $m_{lev}$);
2. for each $pC_j \in pClusters$
3.     node.center[j] = $pCenter_j$;
4.     node.radius[j] = $pRadius_j$;
5.     if $(sizeof(pC_j) <$ leafsize)
6.         New (leaf);
7.         Insert_leaf($pC_j$);
8.         node.children[j]=leaf;
9.     else
10.        New (inter_node);
11.        node.children[j]=inter_node;
12.        R_insert(inter_node, $pC_j$, lev+1);
```

Fig. 3. The algorithm of building a Δ-tree.

tree decreases the distance computational cost and also allows us to exploit the L2 cache more effectively to minimize cache misses.

For the Δ-tree to be effective, we need to be able to determine the optimal number of levels and the number of dimensions to be used at each level. In our presentation, we fix the fanout of the tree. For the number of levels, we adopt a simple strategy: We estimate the number of levels based on the fanout of a node, e.g., given a set of N points and a fanout of f, the number of levels is $L = \lceil log_f N \rceil$.

To determine the number of dimensions m_l to be used at level l, our criterion is to select a cumulative percentage of the total variation that these dimensions should contribute [17]. Let the variance of the jth dimension be v_j. Then, the percentage of variation accounted for by the first k dimensions is given by

$$V_k = \frac{\sum_{j=1}^{k} v_j}{\sum_{j=1}^{D} v_j}.$$

With this definition, we can choose a cut-off V_l^* for each level. Suppose there are L projection levels, we have

$$V_l^* = \frac{l}{L}, 1 \leq l \leq L.$$

Hence, we can retain m_l dimensions in level l, where m_l is the smallest k for which $V_k \geq V_l^*$. In practice, we always retain the first m_l dimensions which preserve as much information as possible, e.g., the first eight dimensions already capture 60 percent of the variation in the 64-dimensional color histogram data.

We note that, for efficiency reasons, we do not require the Δ-tree to be height-balanced. Since the data may be skewed, it is possible that some clusters may be large, while others contain fewer points. If the points in a subcluster at a level l ($< L$) fit into a leaf node, we will not partition it further. In this case, the height of this branch may be shorter than L. On the other hand, for a large cluster, if the number of points at level L is too large to fit into a leaf node, we further split it into subclusters using the full dimensions of the data. We have $m_l = D$ for $l > L$. However, in practice, we find that the difference in height between different subtrees is very small. Moreover, if we should bound the size of a cluster, we can control the height differences.

```
Algorithm KNNSearch(QueryPoint, tree, K)

1. Queue = NewPriorityQueue();
2. KNN = NewKNN();
3. prune_dist = $\infty$;
4. Q' = GetPCA(eigenmatrix, Q);
5. Enqueue(Queue, root);
6. while (Queue is not empty)
7.     node = RemoveFirst(Queue);
8.     for( each child of node)
9.         if ( P_dist(child, Q', $m_{child.lev}$)<prune_dist )
10.            if ( child is an internal node )
11.                Enqueue(Queue, child);
12.            else
13.                for ( each point in leaf )
14.                    if(dist(point, Q)<prune_dist)
15.                        Insert (point, KNN);
16.                        Adjust(prune_dist);
```

Fig. 4. KNN search algorithm for Δ-tree.

3.1.2 The Δ-tree Construction

Fig. 3 shows the algorithm for constructing a Δ-tree for a given data set. We have adopted a top-down approach. At first, routine **PCA()** transforms the data set into the PCA space (line 1). We treat the whole data set as a cluster and refer to these new points as pC. In line 2, the function Init() initiates parameters of root node according to the information of PCA, such as the $eigenmatrix$, the value of L, and the vector m. The default value of $m_l = D$ for $l > L$ and we do not save this value in the root node explicitly. In line 3, we call the recursive routine $\mathbf{R\_insert}(\mathbf{node}, pC, lev)$ that essentially determines the content of the entries of the node at level lev—one entry per subcluster. Note that we are dealing with points in the transformed space (i.e., pC) and that lev determines the number of dimensions that this node is handling.

In line 1 of $\mathbf{R\_insert}(\mathbf{node}, \mathbf{pC}, \mathbf{lev})$, we partition the data of the cluster pC into K subclusters (by K-means). However, this partitioning is performed only on the m_{lev} dimensions of the cluster. For each subcluster, in lines 3-4, we fill the information on the center and radius into the corresponding entry $node$. If the number of points in a subcluster fits into the leaf node (lines 5-8), we insert the points into the leaf node directly. Otherwise (lines 9-12), we recursively invoke routine **R_insert()** to build the next level of the tree.

3.1.3 KNN Search Algorithm

To facilitate the KNN search, we employ two separate data structures. The first is a priority queue that maintains entries in nondescending order of distance. Each item in the queue is an internal node of the Δ-tree. The second is the list of KNN candidates. The distance between the Kth NN and the query point is used to prune away points that are further away.

```
Algorithm RangeQuery(QueryPoint, tree, dis)

1. Queue = NewPriorityQueue();
2. Q' = GetPCA(eigenmatrix, Q);
3. Enqueue(Queue, root);
4. while (Queue is not empty)
5.       node = RemoveFirst(Queue);
6.       for( each child of node)
7.             if ( P_dist(child, Q', $m_{child.lev}$)< dis )
8.                if ( child is an internal node )
9.                   Enqueue(Queue, child);
10.               else
11.                  for ( each point in leaf )
12.                     if(dist(point, Q)<prune_dist)
13.                        Insert point into result set;
```

Fig. 5. Range query algorithm for Δ-tree.

We summarize the algorithm in Fig. 4. In the first stage, we initialize the priority queue, KNN list, and the pruning distance (lines 1-3). Next, we transform the query point from the original space to the PCA-based space using the *eigenmatrix* in the root (line 4). In line 5, we insert the root node into the priority queue as a start. After that, we repeat the operations in lines 7-16 until the queue is empty. We get the first item of the queue which must be an internal node (line 7). For each child of the node, we calculate the distance from Q' to the subcluster in PCA-space (distance is computed with $m_{child.lev}$ dimensions using P_dist()). If the distance is shorter than the pruning distance, we proceed as follows: If the child node is an internal node, it means that there is a further partitioning of the space into subcluster, and we insert the node into the queue (lines 10-11). Otherwise, the child must be a leaf node and we access the real data points in the node and compute their distances to the query point; points that are nearer to the query point are then used to update the current KNN list (lines 12-15). The function Adjust() in line 16 updates the value of pruning distance when necessary, which is always equal to the distance between the query point and the Kth nearest-neighbor candidate.

3.1.4 Range Query Algorithm

As we mentioned previously, *window/range query* can be treated as a *similarity range query* around the center point of the given query range. Therefore, we only present the algorithm which supports *similarity range query*, and call it *range query* shortly. Like KNN search algorithm, we employ two separate data structures. The first is a priority queue that maintains entries within the search distance. Each item is an internal node of the Δ-tree. The second is the list of current query results. The search distance, which defines the query range, is used to prune away points directly.

We summarize the range query algorithm in Fig. 5. The algorithm is very straightforward. At first, we initialize the priority queue. After that, we transform the query point from the original space to the PCA-based space using the *eigenmatrix* in the root (line 2). In line 3, we insert the root node into the priority queue as a start. After that, we repeat the operations in lines 5-13 until the queue is empty. We get the first item of the queue which must be an internal node (line 5). For each child of the node, we calculate the distance from Q' to the subcluster in PCA-space (distance is computed with $m_{child.lev}$ dimensions using P_dist()). If the distance is shorter than the search distance *dis*, we proceed as follows: If the child node is an internal node, it means that there is a further partitioning of the space into subclusters and we insert the node into the queue (lines 8-9). Otherwise, the child must be a leaf node and we access the real data points in the node and compute their distances to the query point; if the distance between point and the query point is less than the search distance, the point is inserted into the result set.

```
Algorithm Insert(newpoint, tree)

1. P' = GetPCA(eigenmatrix, P);
2. node = root;
3. while ( node is not leaf )
4.       NearSubCluster = FindBestCluster(node, P);
5.       Adjust( node_radius[NearSubCluster]);
6.       if (node_child[NearSubCluster] is internal node)
7.             node = node_child[NearSubCluster];
8.       else
9.             leaf = node_child[NearSubCluster];
10. if (leaf is not full)
11.      Insert(leaf, P);
12. else
13.      Split(leaf, newleaf)
14.      if (leaf.parent is not full )
15.            InsertLeaf(leaf.parent, leaf, newleaf)
16.      else
17.            New(inter_node)
18.            InsertNode(leaf.parent, inter_node)
19.            InsertLeaf(inter_node, leaf, newleaf)
```

Fig. 6. Insert algorithm for the Δ-tree.

3.1.5 Insertion of the Δ-tree

So far, we have seen the Δ-tree as a structure for static databases. However, the Δ-tree can also be used for dynamic databases. This is based on the properties of PCA. When a new point is inserted, we simply apply *eigenmatrix* on the new point to transform it into PCA space and insert it into the appropriate subcluster. To reduce the complexity of the algorithm, we only update the radius and keep the original center of the affected cluster. This may result in a larger cluster space and degrade the precision of *eigenmatrix* gradually, but, as our study shows, the Δ-tree is still effective.

The algorithmic description of the insert operation is shown in Fig. 6. We first transform the newly inserted point into the PCA-space using the *eigenmatrix* in the root node (line 1). We then traverse down the tree (beginning from the root node (line 2)) to the leaf node by always selecting the nearest subcluster along the path (lines 3-10). If the leaf node has free space, we insert the new point into the leaf (line 12). Otherwise, we must split the leaf node before insertion. To split the leaf, we generate two clusters. If the parent node is not full, the routine **InsertLeaf( )** (line 15) inserts two new clusters into the parent. Otherwise, we generate a new internal node and insert the leaf nodes (lines 17-19). In this case, a new internal node level is introduced.

We note that our algorithm does not change the cluster $eigenmatrix$ as new points are added. As such, it may not reflect the real feature of a cluster as more points are added since the optimal $eigenmatrix$ is derived from all known points. On the other hand, once we change the $eigenmatrix$, we have to update the keys of the original data points as well. As a result, insertion may make it necessary to rebuild the tree periodically. Two mechanisms to decide when to rebuild the tree are as follows:

1. *Insertion threshold*: In this naive method, once the newly inserted points exceed a predetermined threshold, say 50 percent of the original data size, we rebuild the tree regardless of the precision of the original $eigenmatrix$.
2. *Distance variance threshold*: Given a set of N points, we form a cluster with center c. After PCA transformation, we have another center c' in the projected lower-dimensional space. Originally, we have the average distance variance V_a:

$$V_a = \frac{\sum_{i=1}^{N} |dist(P_i, C) - dist(P_i', c')|}{N}.$$

Once we insert a point P, we get a new average distance variance, called V_a':

$$V_a' = \frac{V_a * N + |dist(P, C) - dist(P', c')|}{N + 1}.$$

V_a' may change after every insertion. Once $\frac{V_a' - V_a}{V_a}$ is larger than a predefined threshold, we rebuild the tree. Note that we do not modify the value of cluster centers C and c' for each insertion unless the cluster has to be split. In this case, the new cluster centers will be generated. This method is expected to be more efficient because the decision factor is related to the distribution of newly inserted points.

3.1.6 *Deletion of the* Δ-tree

In the above section, we have discussed the algorithm to deal with insertion of the Δ-tree. Although deletion is different from insertion, we can apply the similar scheme on the delete operation. The details are as follows:

1. We first transform the point into the PCA-space using the $eigenmatrix$. We then traverse down the tree to the leaf node holding the point. Either the KNN search or range query algorithm can be used to locate the point.
2. We delete the point and, in the case of underflow, the node is merged with the sibling node.
3. Note that we do not change the cluster $eigenmatrix$ in the first two steps. However, we record the effect of deletion on $eigenmatrix$. Like insert operations, large volumes of deletion may trigger rebuilding of the tree.

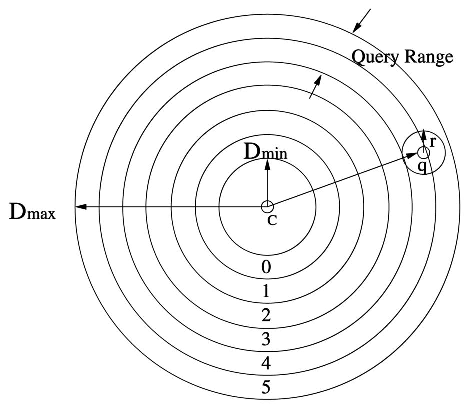

Fig. 7. An example of the pruning searching space.

3.2 The Δ^+-tree: A Partition-Based Enhancement of the Δ-tree

While the proposed Δ-tree can efficiently prune the search space, it has several limitations. First, its effectiveness depends on how well a data set is globally correlated, i.e., most of the variations can be captured by a few principle components in the transformed space. For real data sets that are typically not globally correlated, more clusters may have to be searched. Second, the complete dataspace has to be evaluated to generate the PCA $eigenmatrix$, which may cause significant loss of distance information resulting in more false drops and, hence, a high query cost. Third, there is a need to periodically rebuild the whole tree for optimal performance.

In this section, we propose an extension, called Δ^+-tree, that addresses the above three limitations. To deal with the first limitation, we globally partition the dataspace into multiple clusters and manage the points in each cluster with a Δ-tree. For simplicity, we also employ the K-means clustering scheme to generate the global clusters and apply PCA for clusters individually. We use a directory to save the information of global clusters. Each entry of the directory represents a global cluster and has its own $eigenmatrix$, L, m, cluster center, radius, and a pointer to the corresponding Δ-tree that manages its points.

Even with the proposed enhancement, the second limitation remains: Each global cluster space has to be examined completely. Our solution is to partition the cluster into smaller regions so that only certain regions need to be examined. We made the following observations of a cluster:

1. Points close to each other have similar distances to a given reference point. The distance value is single dimensional and it can be easily divided into different intervals.
2. A cluster can be split into regions ("concentric circle") as follows: First, each point is mapped into a single-dimensional space based on the distance to the cluster center. Second, the cluster is partitioned. Let D_{min} and D_{max} be the minimum and maximum distance of points within the cluster to the center. Let there be k regions (k is a predetermined parameter). The points in region i must satisfy the following equations:

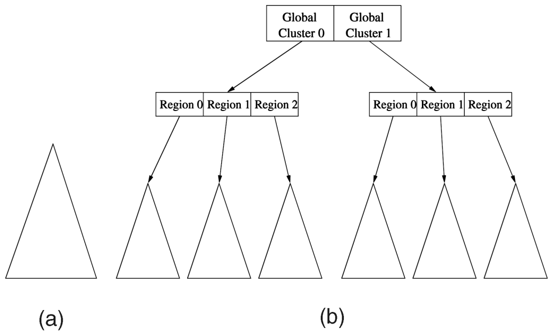

Fig. 8. The structure of Δ-tree variants. (a) Δ-tree and (b) $\Delta+$-tree.

$$\begin{cases} D_{min}+i*f \le Dist_i \le D_{min}+(i+1)*f & i=0 \\ D_{min}+i*f < Dist_i \le D_{min}+(i+1)*f & 1 \le i < k, \end{cases}$$

where $f = (D_{max} - D_{min})/k$. Fig. 7 shows an example of a cluster with six regions.

3. Given a query point, we can order the regions in nondescending order of their minimum distance to the query point. The regions are then searched in this order. This step can be efficiently performed by checking against the partitioning vectors (i.e., $D_{min}, D_{min}+f, \ldots, D_{min}+(k-1)f$) of the region. For example, consider the query point q in Fig. 7. q falls in region 4. As such, region 4 will be examined first, followed by regions 5, 3, 2, 1, and 0.
4. We note that this partitioning scheme can potentially minimize the search space by pruning away some regions. Using the same example as before, if the current KNN points after searching, say, region 5, are already nearer than the minimum distance between qt and region 3, we only need to search the subtrees of certain partitions and regions 3, 2, 1, and 0 need not be examined.

Based on these observations, we can introduce a new level immediately after the directory. In other words, instead of building a Δ-tree for each global cluster, we partition it as described above. For each region, we build a Δ-tree. We refer to this new structure as the Δ^+-tree. Fig. 8b shows a Δ^+-tree structure. The whole data set has two global clusters and we partition the cluster into three regions. For comparison purposes, we also show the Δ-tree (Fig. 8a).

As described above, the operations on the Δ^+-tree are quite similar to those on the Δ-tree. To build a Δ^+-tree, we first globally partition the dataspace into multiple clusters and apply PCA for clusters individually. The cluster information is stored in the flat directory. Second, we partition the cluster into smaller regions according to the distance to the cluster center. Finally, we build a Δ-tree for each region deploying the algorithm **Buildtree()**, as shown in Section 3.1.2. For all the other operations, we first locate the nearest global cluster and region and then traverse the subtree. The operations within the subtree are same as that of the Δ-tree. This process continues with other clusters, while cluster or regions containing points further than the Kth NN are not traversed.

The proposed Δ^+-tree is also more update efficient—-while it cannot avoid a complete rebuild, it can defer a complete rebuild to a longer period (compared to Δ-tree). Recall that the structure partitions the data space into global clusters before PCA transformation. As such, it localizes the rebuilding to only clusters whose *eigenmatrix* is no longer optimal as a result of insertions, while other clusters are not affected at all. A complete rebuild will eventually be needed if the global clusters are no longer optimal. As we will see in our experimental study, the index remains effective for a high percentage of updates.

4 A Performance Study

In this section, we present an experimental study to evaluate the Δ-tree and Δ^+-tree. The performance is measured by the average execution time, cache misses, and distance computation for KNN searches over 100 different queries. We use the *Perfmon* tool [13] to count L2 cache misses. All the experiments are conducted on a SUN E450 machine, with 4 GB RAM and 2 MB L2 cache with cache line size 64 bytes. The machine is running SUN OS 5.7. All the data and index structures are loaded into the main memory before each experiment begins. We demonstrate the results on the random data set, synthetical clustered data set, and real-life data sets.

4.1 Tuning the Δ^+-tree

The Δ^+-tree has two extra parameters: the number of global clusters and regions. When both parameters are set to 1, the Δ^+-tree becomes the Δ-tree. In the first experiment, we tune

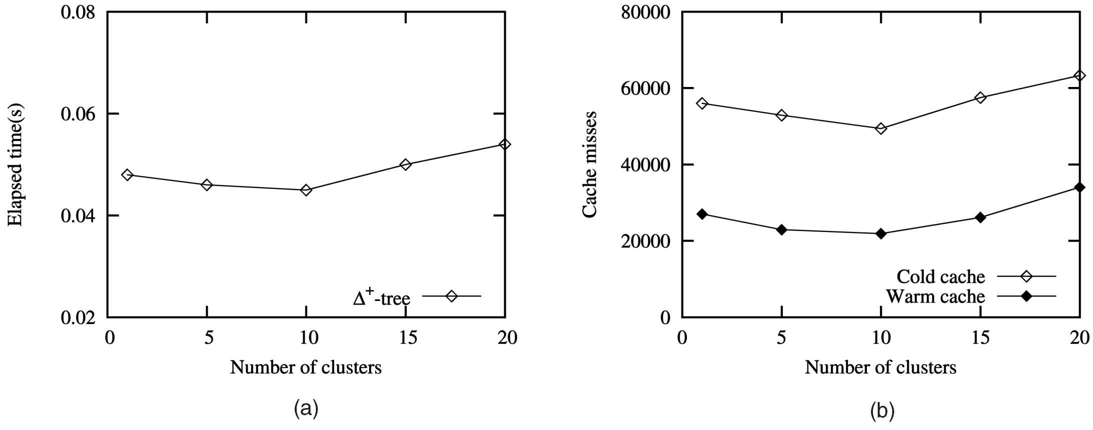

Fig. 9. NN search for different cluster numbers. (a) Elapsed time and (b) cache misses.

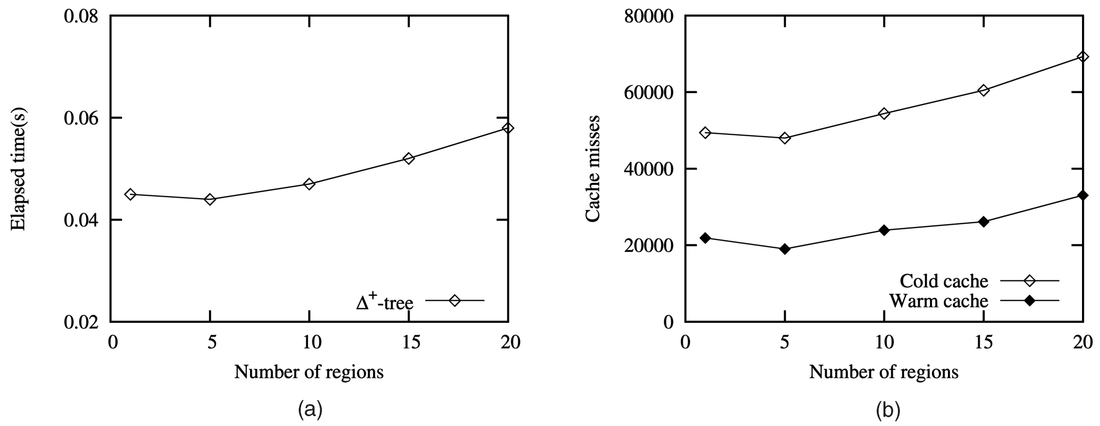

Fig. 10. NN search for different region numbers. (a) Elapsed time and (b) cache misses.

these two parameters of the Δ^+-tree. We use a real-life data set consisting of 64-dimensional color histograms extracted from the Corel Database [2].

First, we set the number of region 1 and vary the number of clusters. The default fanout of the subtree we used is 10, which is near optimal, as shown in the following Section 4.2. Fig. 9 shows the NN search performance of our new structures for different cluster numbers.

We observe that the performance of Δ^+-tree improves with relatively large numbers of clusters. As shown in Fig. 9a, when the number of clusters is fewer than 10, the larger the cluster number the better the performance. Because the image data set is typically skewed, it is possible for queries to be constrained within a certain cluster. In this case, the search space can be reduced efficiently, compared with the Δ-tree, as we only need to search a subtree. However, the performance starts to degenerate when the number exceeds 10. If we partition the data set into too many clusters, the query may incur multiple subtree traversals, which introduce more cost overhead.

We conduct two sets of experiments to test the number of cache misses:

- *cold cache*: the cache is flushed after every query; that is, the cache does not contain any index nodes or pages,
- *warm cache*: the queries run consecutively without cache flushing.

In both cases, we record the average number of cache misses for 100 queries. We found that the actual cache miss cost is only around 10 percent of the overall time cost if we run the queries consecutively, i.e., *warm cache*. This is the case in our studies as we do not perform any cache flushing between queries. Since we have run many queries on a single index structure, the cache hits are high, because some highly accessed cache lines can always reside in the L2 cache. However, the number of cache misses can be much larger in the case of *cold cache*. Our investigation shows that the number of cache misses for an independent query can be twice as much.

Another parameter of the Δ^+-tree is the number of regions. We set the cluster number to 10 and show the experiment result for varying number of regions in Fig. 10. We found that the performance is near optimal when the number of regions is between 3 and 5. The query can be limited to a region and, hence, the search space is reduced. Partitioning too many regions also introduces more subtree traversal which degrades the performance.

In this experiment, we test the performance of two extra parameters of the Δ^+-tree. However, the optimal values of the parameters may vary with different data sets. If we know the distribution of the data set, we can apply the optimal value to the index structure. For simplicity, we set the default number of clusters and regions to be 10 and 5, respectively, throughout the performance study. Additionally, we only show the average number of cache misses for 100 consecutive queries without cache flushing, i.e., *warm cache*.

4.2 Comparing Δ-tree and Δ^+-tree

We conduct an extensive performance study to tune the two proposed schemes for optimality. Here, we present one representative set that studies the effect of node size, i.e., we vary the node size from 1 KB to 8 KB. Although the optimal node size for single-dimensional data sets has been shown to be the cache line size [21], this choice of node size is not optimal in high-dimensional cases—the L2 cache line size is usually 64 bytes, which is not sufficient to store a high-dimensional data point (256 bytes for 64 dimensions) in a single cache block. [19] shows that, even for 2-dimensional data, the optimal node size can be up to 256-512 bytes and increases as the dimensionality increases. The minimum cache misses is a compromise of node size and tree height. High-dimensional indexes require more space per entry and, therefore, the optimal node size is larger than cache line size.

Fig. 11 shows the NN search performance of our new structures for different node sizes. The node size here represents the size of leaf node. Since we fix the fanout of tree in our implementation, we have varied internal node size depending on the remaining dimensions in each level. As shown in the figure, there is an optimal node size that should be used. When the node size is small ($< 2K$), the fanout of the tree is also small. In multidimensional indexes, more than one node of the same level needs to be accessed and the small fanout introduces high overlap between

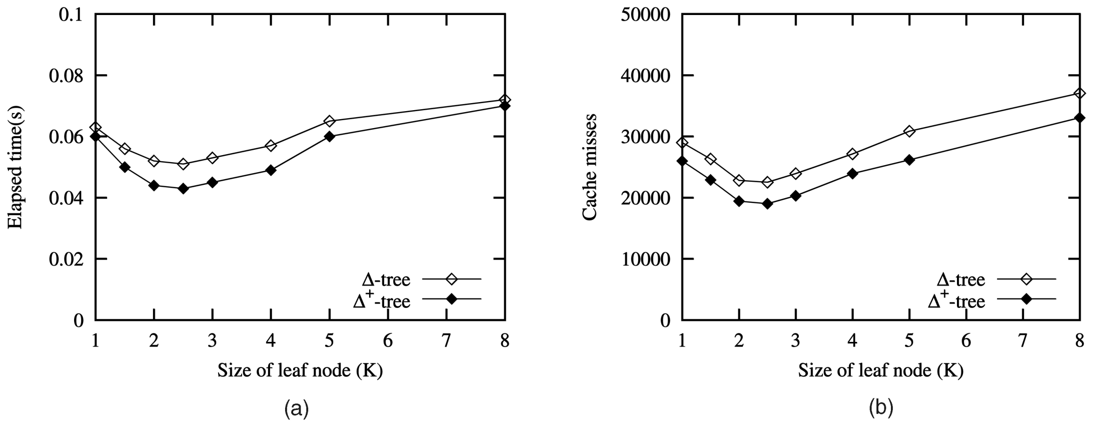

Fig. 11. NN search for different node sizes. (a) Elapsed time and (b) cache misses.

nodes. As a result, more nodes will have to be accessed. We observe that as the node size increases, the number of accessed nodes decreases. The performance is optimal when the node size is around 2-3K. However, as the node size reaches beyond a certain point ($> 3K$), the performance starts to degenerate again. This is because too large a node size results in more cache misses per node. Therefore, the total cache misses increase. The optimal node size (2-3K) is a compromise of these factors.

The results, shown in Fig. 11, clearly demonstrate the superiority of the Δ^{+}-tree over the Δ-tree: It is about 15 percent better than the Δ-tree. This is because the Δ^{+}-tree searches a smaller data space compared to the Δ-tree. First, the Δ^{+}-tree globally partitions the data set into clusters before PCA transformation, thus the *eigenmatrix* is more efficient than that of the Δ-tree. Second, the Δ^{+}-tree may only need to search a few clusters and exclude the other clusters that are far to the query point. Third, partitioning the cluster into regions can further reduce the search space compared to the Δ-tree that examines the whole data space. Hence, the total cost of cache misses and computation is reduced by the Δ^{+}-tree.

Since Δ^{+}-tree performs better than Δ-tree, in the following experiments, we shall restrict our discussion to the Δ^{+}-tree and use the optimal parameters determined above.

4.3 Comparison with Other Structures

In this section, we only demonstrate the results on the synthetical clustered data set and real-life data set. For uniformly distributed random data, all the index structures yield similar performance when the dimensionality is beyond 30. Because of the large NN distance, all the methods have to access most of the data for a single query when dimensionality is high. As shown in [4], we can determine the expected distance of the query point to the nearest neighbor in the database. Assuming a uniformly distributed data set in a normalized data space $[0,1]^d$ with N points, the nearest-neighbor distance can be approximated by the volume of the sphere, which, on the average, contains one data point. The data space with radii r can be calculated by

$$sp^d(r) = \frac{\sqrt{\pi^d}}{\Gamma(d/2+1)} \cdot r^d,$$

where $\Gamma(n)$ is the gamma function ($\Gamma(x+1) = x \cdot \Gamma(x)$, $\Gamma(1) = 1$, and $\Gamma(1/2) = \sqrt{\pi}$). Since $sp^d(dist^{nn}) = \frac{1}{N}$, we can get the the expected nearest-neighbor distance

$$dist^{nn}(N,d) = \frac{1}{\sqrt{\pi}} \cdot \sqrt[\lceil d \rceil]{\frac{\Gamma(d/2+1)}{N}}.$$

Based on this formula, the $dist^{nn}$ can become larger than the length of the data space, i.e., 1, when the dimensionality is higher than 30. Because of a large NN distance, it is almost impossible to partition the data space well, thus the tree index cannot be efficient for uniformly distributed data due to its extra tree operations. Therefore, the Sequential Scan can be the best scheme for high-dimensional data space ($D > 30$). Interested readers can refer to [11] for the experimental results of uniformly distributed data set.

For the rest of this paper, we only focus on the TV-tree, Slim-tree, iDistance, Omni-sequential, Sequential Scan, and the Δ^{+}-tree. We exclude the CR-tree due to its poor performance for high-dimensional data sets. Because the Slim-tree is optimized from the M-tree and performs better, we also omit the M-tree in the comparison for the clarity of figures. To ensure a fair comparison, we optimize these methods for main memory indexing purposes, such as tuning the node size. We only present the optimal result of each structure.

4.3.1 On Clustered Data Sets

In many applications, data points are often correlated in some ways. In this set of experiments, we evaluate the Slim-tree, Omni-sequential, TV-tree, iDistance, Sequential Scan, and Δ^{+}-tree on clustered data sets. We generate the data for different dimensional spaces ranging from 8 to 64 dimensions, each having 10 clusters. We use a method similar to that of [8] to generate the clusters in subspaces of different orientations and dimensionalities. All data sets have 1,000,000 points. We vary the cluster number of the Δ^{+}-tree; as we expected, the tree yields optimal performance when the number of clusters exactly matches the a priori number of clusters used in the Δ^{+}-tree (which is 10).

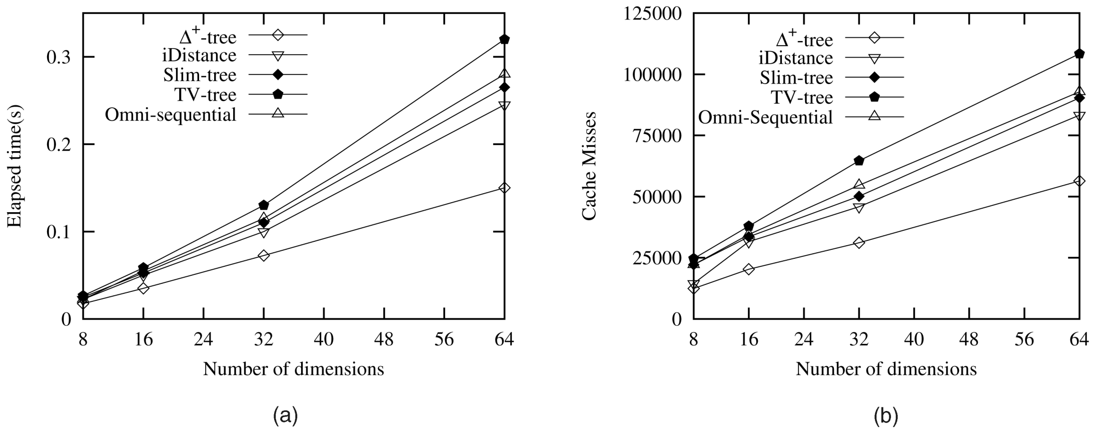

Fig. 12. NN search for clustered data sets. (a) Elapsed time and (b) cache misses.

Fig. 12 shows the time and cache misses of a NN search as we vary the number of dimensions from 8 to 64. Since the cost of memory access is mainly determined by cache access and the node sizes varies for different indexes, we omit the details of node accesses. Because Sequential Scan performs poorly and all these tree structures achieve a speedup by a factor of around 15 over the Sequential Scan, we omit Sequential Scan from the figures to clearly show the differences between the other schemes. The reason is clear: For Sequential Scan, we must scan the whole data set to get the nearest neighbors and the cost is proportional to the size and dimension of the data set. When the data set is clustered, the nearest neighbors are (almost) always located in the same cluster with the query point. Thus, the tree structures can prune most of the data when traversing the trees.

Our Δ^+-tree can be 60 percent faster than the other three methods, especially when the dimensionality is high. The Δ^+-tree has three advantages over the Slim-tree. First, the Δ^+-tree reduces the cache misses compared to the Slim-tree. Because we index different levels of projections of the data set, the number of projected dimensions in the upper levels is much smaller than that of real data. When the original space is 64 dimensions, the dimensions in the first three upper levels are fewer than 15. The node size of the Δ^+-tree in the upper levels can be much smaller than that of the Slim-tree; consequently, the index size of the Δ^+-tree is also smaller. In the tree operations, upper levels of the tree will probably remain in the L2 cache as they are accessed with high frequency, so the Δ^+-tree can benefit more from this property. Furthermore, the internal nodes of the Δ^+-tree are full and fewer node accesses are needed because of hierarchical clustering. The effect is that the Δ^+-tree can reduce more cache misses compared to the Slim-tree. Second, the computational cost of Δ^+-tree is also smaller than the Slim-tree. The reason is, in the projection level, the distance computation is much faster because of reduced dimensionality. For example, when the dimension of the projection is 8, it already captures more than 60 percent of the information of the point. This means we can prune the data efficiently in this projection level as the computational cost is only 12.5 percent of the actual data distance computation. Third, we build the tree from top down exploiting the global clustering compared with local partition of Slim-tree, so the benefit of the Δ^+-tree is also from the improved data clustering which reduce the search space and, hence, the fewer operations for a query.

Comparing with the iDistance, although the B^+-tree structure of iDistance is more cache conscious, the iDistance incurs more distance computation and cache misses than the Δ^+-tree, because the search space of iDistance is larger. Omni-sequential also transforms the high-dimensional data into single-dimensional space based on a set of global foci and uses precomputed distance to prune the distance calculations. Although these two methods exploit different pruning mechanisms, mapping high-dimensional data to 1-dimensional space is less efficient in filtering data, compared to the Δ^+-tree, because of the heavy information loss.

Not surprisingly, the TV-tree performs worst among these index methods, especially when dimensionality is high. There are several reasons for this behavior. First, the TV-tree is essentially similar to the R-tree and its effectiveness depends on α, the number of active dimensions. It turns out that for the data sets used, the data are not globally correlated. As a result, the optimal α value for the TV-tree remains relatively large. For example, for 64 dimensions, we found that $\alpha \approx 20$. The reduced number of dimensions is still too large for an R-tree-based scheme

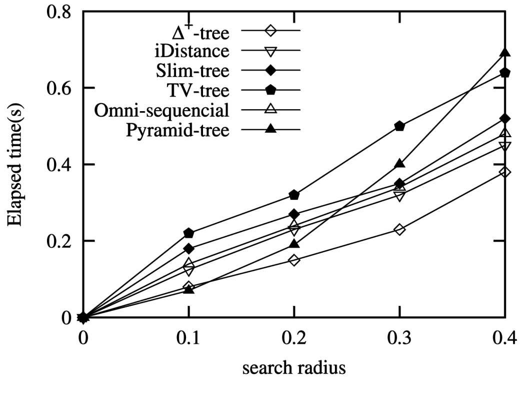

Fig. 13. Range query for clustered data set.

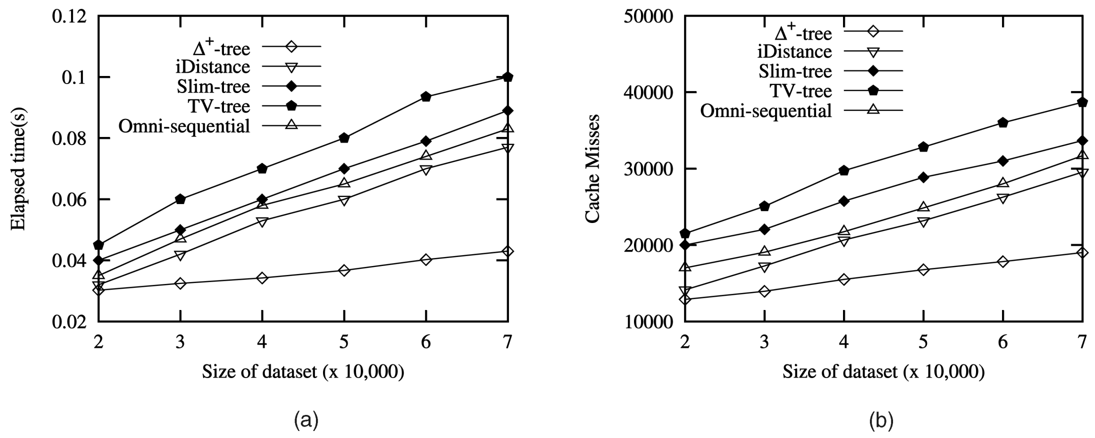

Fig. 14. NN search for 64D real data set. (a) Elapsed time and (b) cache misses.

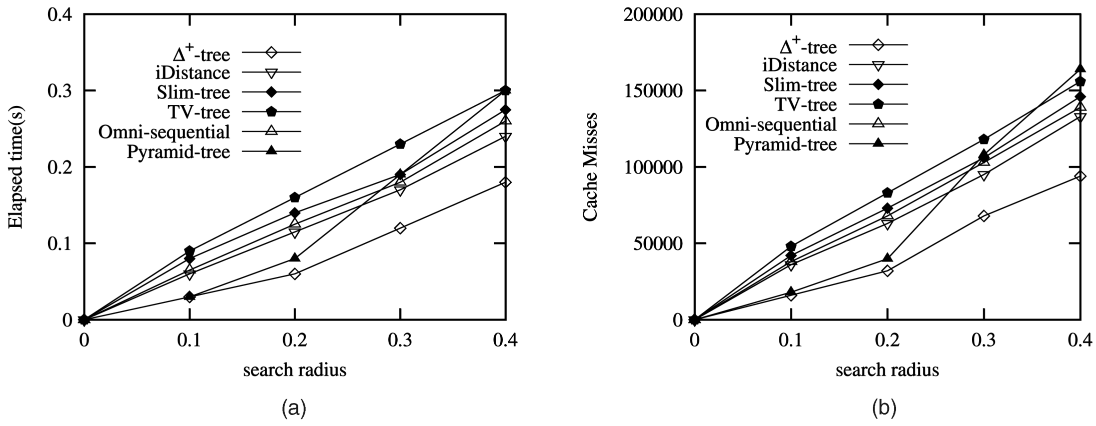

Fig. 15. Range query for 64D real data set. (a) Elapsed time and (b) cache misses.

(like the TV-tree) to perform well. Moreover, searching the reduced dimensions leads to false admissions, which results in more nodes being accessed.

In the next experiment, we compare the range query performance for these indexes. Since the Pyramid-tree was specifically proposed for range search in high-dimensional space, we also include it in the comparison. Fig. 13 shows the results of range search for 64D clustered data set with varying search radius from 0 to 0.4. It is clear that all the indexes perform worse as we increase the search radius or the dimensionality, because the query have to search more space, which means more cache misses and distance computation. Although the Pyramid-tree performs well when the search radius is small, it degrades faster than other methods. The Pyramid-tree is primarily designed and optimized for queries of small side length on uniform data, but it does not work well with big sized queries. The Δ^+-tree is the best among these structures for most of the case, however, the gap between other indexes is narrowed. The reason is that a large search radius reduces the efficiency of the index technology.

4.3.2 *On Real Data Sets*

In this experiment, we evaluate the various schemes on the different real-life data sets, 70K 64-dimensional color histograms [2] and 11K 79-dimensional motion capture data set [1]. First, we test the performance of KNN search. The performance is quite similar to the clustered data sets. The tree-based methods are at least 10 times faster than Sequential Scan because the real data set is generally skewed. As such, we will not present the results for Sequential Scan.

The NN performance comparisons for 64D data set among the Δ^+-tree, Slim-tree, TV-tree, Omni-sequential, and iDistance are shown in Fig. 14. The Δ^+-tree is about 50 percent faster than other methods for NN search in terms of cache misses and time cost. These results clearly show the effectiveness of the Δ^+-tree, even if the number of clusters employed may not match that of the data set. The Slim-tree is poor because it incurs more computation and it uses all the dimensions in the internal nodes, resulting in more cache misses. Although it adjusts the nodes to reduce the overlap, the partition is not as good as that of the Δ^+-tree, because the Δ^+-tree clusters the data set from top-down by capturing the overall data distribution and makes optimal partition. The iDistance and Omni-sequential are worse than the Δ^+-tree because the distance information loss incurs larger search space and, hence, more computation and cache misses. The number of active dimensions for the TV-tree remains large, so its performance is affected by the scalability problem of the R-tree. In the context of main

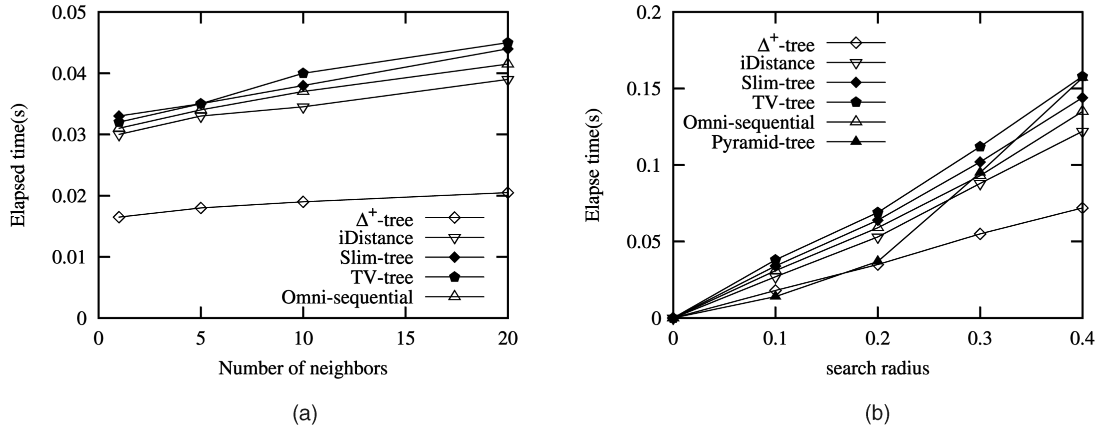

Fig. 16. Performance for 79D real data set. (a) KNN and (b) range query.

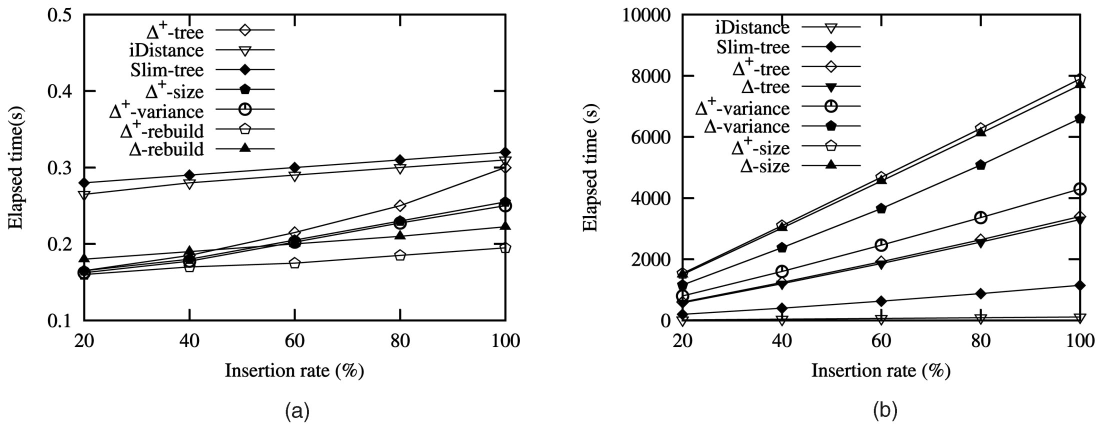

Fig. 17. Performance for the effect of insertion. (a) NN search and (b) insertion.

memory indexing, this translates into higher computational cost and number of cache misses.

As in Section 4.3.1, we also conduct the range queries for these indexes. Fig. 15 shows the range query results with varying the search radius from 0 to 0.4. All the indexes degrade as we increase search radius because a large search range incurs more cache misses and distance computation. However, our method is still 30 percent better for relatively large search radius. We can expect that the performance of all the index structures will degrade to that of the Sequential Scan for very large search radius where all the data needs to be accessed.

The results for KNN and range queries in the motion capture data set are presented in Fig. 16. The performance differences are quite similar to that of color histogram data set. Moveover, the gap between the Δ^+-tree and other indexes is even widened. The reason is that the motion data set is more skewed and, hence, the Δ^+-tree can benefit more from the PCA transformation. Checking the effect of cumulative variation of the data set after applying PCA, we found the first six dimensions in the PCA-space can capture 80 percent of the data information; furthermore, the first 17 dimensions in the PCA-space capture 95 percent of the information. Therefore, the Δ^+-tree can apply the reduced dimensionality to decrease the distance computational cost and cache misses without much information loss. We also note that the TV-tree performs better as we can use fewer active dimensions for this data set.

4.3.3 *On the Effect of Insertion*

In the last experiment, we study the effect of insertion on the Δ^+-tree and Δ-tree. The delete operations yield similar performance and we omit it. For the Δ^+ tree and Δ-tree, we evaluated four versions, e.g., Δ^+-tree represents the version that is based on the proposed insert algorithm (without rebuilding); Δ^+-rebuild represents the version that always rebuild the tree upon insertion, i.e., this is ideal and represents the optimal Δ^+-tree; Δ^+-size represents the version that rebuilds the tree after a certain size threshold is reached; and Δ^+-variance represents the version that rebuilds the tree after a certain variance threshold is reached. In our experiment, we set the threshold as 10 percent for the latter two schemes. We used the 64D clustered synthetic data set for this experiment. We first build the tree structure with 1,000,000 data. Subsequently, we insert up to 100 percent more new data points. The results of insertion and NN search are shown in Fig. 17. The performances of Omni-sequential and TV-tree are similar to iDistance and Slim-tree, respectively, and we omit them for the clarity of figure.

Fig. 17a shows the average execution time of the NN search on the new data sets after 20 percent of newly inserted data. First, we observe that the Δ^+-trees and

Δ-trees outperform the other tree structures. We only show the performance of the Δ-rebuild for comparison because the Δ-trees show the similar performance degeneration as the Δ^+-trees. This clearly demonstrates the effectiveness of the proposed schemes. Second, as expected, as more points are inserted, the performance of Δ^+-tree degenerates as the newly inserted data affect the precision of cluster *eigenmatrix*. However, the degradation of performance is marginal. More importantly, the accuracy is not affected—the Δ^+-tree may examine more nodes because of the relatively larger radius. Third, it is clear that the rebuilding algorithms can reduce the performance degradation. For Δ^+-size and Δ^+-variance, the degradation is around 30 percent even for 100 percent new insertions. So, the Δ^+-tree remains very effective after new data points are inserted. We also observe that Δ^+-size is slightly better than Δ^+-variance. This is because it incurs more rebuilding. Once the new points of a global cluster are more than the threshold, it rebuilds the subtree regardless of the data distribution. The results show Δ^+-tree's robustness with respect to updates, in the sense that it can take sufficiently large number of updates before we need to rebuild the tree structure.

Fig. 17b shows the execution time for insertions. The Δ^+-tree and Δ-tree incur more expensive insertion cost because K-means clustering and PCA transformation are necessary for the reorganization of the tree. We did not show the rebuild version as the rebuild cost for each insertion is extremely high. However, our two proposed update mechanisms can significantly reduce the insertion cost, especially the *Distance variance threshold* method. The Δ^+-variance yields better performance than Δ-variance as the points are inserted into respective global cluster and the efficiency of *eigenmatrix* degenerates less than that of Δ-tree, hence, reduces the rebuild frequency. The iDistance is the most efficient in terms of insertion, as we only need to insert a new record into the B^+-tree. However, the insertion cost is small compared with NN query cost because the query in high-dimensional space typically needs to access a great amount of nodes. Additionally, we can do the rebuilding offline while not interfering with the other queries. This makes the Δ^+-tree a promising candidate even for dynamic data sets.

5 Conclusion

In this paper, we have addressed the problem of in-memory similarity search for high-dimensional data. We presented an efficient novel index method, called Δ-tree. The Δ-tree employs hierarchical clustering top down and applies multiple level of projections of points to allow nodes to better fit into the L2 cache. The search process can prune the search space efficiently, thus be accelerated by reducing computational cost and cache misses. We also proposed an extension, called Δ^+-tree, that partitions the data set into clusters and then further divides a cluster into regions. The global clustering of Δ^+-tree can capture the overall feature of the data set and better utilize the PCA. We conducted extensive experiments to evaluate the Δ-tree and Δ^+-tree against a large number of known techniques and the results showed that our technique is superior.

References

[1] CMU Graphics Lab Motion Capture Database, available from http://mocap.cs.cmu.edu/, 2004.
[2] Corel Image Features, available from http://kdd.ics.uci.edu, 2000.
[3] S. Berchtold, C. Böhm, and H.P. Kriegel, "The Pyramid-Tree: Breaking the Curse of Dimensionality," *Proc. ACM SIGMOD Conf.*, pp. 142-153, 1998.
[4] S. Berchtold, C. Bohm, D. Keim, F. Krebs, and H.P. Kriegel, "On Optimizing Nearest Neighbor Queries in High-Dimensional Data Spaces," *Proc. Eighth Int'l Conf. Database Theory*, pp. 435-449, 2001.
[5] P. Bohannon, P. Mcllroy, and R. Rastogi, "Main-Memory Index Structures with Fixed-Size Partial Keys," *Proc. ACM SIGMOD Conf.*, pp. 163-174, 2001.
[6] C. Bohm, S. Berchtold, and D. Keim, "Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases," *ACM Computing Surveys*, pp. 322-373, 2001.
[7] T. Bozkaya and M. Ozsoyoglu, "Distance-Based Indexing for High-Dimensional Metric Spaces," *Proc. ACM SIGMOD Conf.*, pp. 357-368, 1997.
[8] K. Chakrabarti and S. Mehrotra, "Local Dimensionality Reduction: A New Approach to Indexing High Dimensional Spaces," *Proc. 26th Very Large Data Bases Conf.*, pp. 89-100, 2000.
[9] S. Chen, P.B. Gibbons, and T.C. Mowry, "Improving Index Performance through Prefetching," *Proc. ACM SIGMOD Conf.*, pp. 139-150, 2001.
[10] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. 24th Very Large Data Bases Conf.*, pp. 194-205, 1997.
[11] B. Cui, B.C. Ooi, J.W. Su, and K.L. Tan, "Contorting High Dimensional Data for Efficient Main Memory Processing," *Proc. ACM SIGMOD Conf.*, pp. 479-490, 2003.
[12] B. Cui, B.C. Ooi, J.W. Su, and K.L. Tan, "Main Memory Indexing: The Case for BD-Tree," *IEEE Trans. Knowledge and Data Eng.*, 2003.
[13] R. Enbody Perfmon: Performance Monitoring Tool, available from http://www.cps.msu.edu/enbody/perfmon.html, 1999.
[14] R.F.S. Filho, A. Traina, C. Traina Jr., and C. Faloutsos, "Similarity Search without Tears: The Omni-Family of All-Purpose Access Methods," *Proc. 17th Int'l Conf. Data Eng.*, 2001.
[15] G.H. Golub and C.F. Van Loan, *Matrix Computations.* The Johns Hopkins University Press, 1989.
[16] J. Hui, B.C. Ooi, H. Shen, C. Yu, and A. Zhou, "An Adaptive and Efficient Dimensionality Reduction Algorithm for High-Dimensional Indexing," *Proc. 19th Int'l Conf. Data Eng.*, 2003.
[17] I.T. Jolliffe, *Principal Component Analysis.* Springer-Verlag, 1986.
[18] C. Traina Jr., A. Traina, C. Faloutsos, and B. Seeger, "Fast Indexing and Visualization of Metric Data Sets Using Slim-Trees," *IEEE Trans. Knowledge and Data Eng.*, 2002.
[19] K. Kim, S.K. Cha, and K. Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," *Proc. ACM SIGMOD Conf.*, pp. 139-150, 2001.
[20] K. 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.
[21] J. Rao and K. Ross, "Making B+-Trees Cache Conscious in Main Memory," *Proc. ACM SIGMOD Conf.*, pp. 475-486, 2000.
[22] R. Weber, H.J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. 24th Very Large Data Bases Conf.*, pp. 194-205, 1998.
[23] C. Yu, B.C. Ooi, K.L. Tan, and H.V. Jagadish, "Indexing the Distance: An Efficient Method to KNN Processing," *Proc. 27th Very Large Data Bases Conf.*, pp. 421-430, 2001.

Bin Cui received the BS degree in computer science from Xi'an Jiaotong University, China, in 1996, and the PhD degree in computer science from the National University of Singapore in 2004. Currently, he is a research fellow at Singapore-MIT Alliance. His major research interests include index techniques, multi/high databases, multimedia retrieval, and database performance. He has served on program committees of several database conferences.

Beng Chin Ooi received the BSc (First Class Honors) and PhD degrees from Monash University, Australia, in 1985 and 1989, respectively. He is currently a professor of computer science at the School of Computing, National University of Singapore. His current research interests include database performance issues, index techniques, XML, spatial databases, and P2P/grid computing. He has published more than 100 conference/journal papers and served as a program committee member for a number of international conferences (including SIGMOD, VLDB, ICDE, EDBT, and DASFAA). He is an editor of *GeoInformatica*, the *Journal of GIS*, *ACM SIGMOD Disc*, *VLDB Journal* and the *IEEE Transactions on Knowledge and Data Engineering*. He is a member of the ACM and the IEEE.

Jianwen Su received the BS and MS degrees in computer science from Fudan University and the PhD degree in computer science from the University of Southern California. He joined the Department of Computer Science at the University of California at Santa Barbara in 1990. Dr. Su is a member of the ACM and SIGMOD, and a senior member of the IEEE and the IEEE Computer Society. He has served on program/organizational committees of several database conferences.

Kian-Lee Tan received the BSc (Hons) and PhD degrees in computer science from the National University of Singapore in 1989 and 1994, respectively. He is currently an associate professor in the Department of Computer Science, National University of Singapore. His major research interests include query processing and optimization, database security, and database performance. He has published more than 140 conference/journal papers in international conferences and journals. He has also coauthored three books. Dr. Tan is a member of the ACM and the IEEE Computer Society.