Contorting High Dimensional Data for Efficient Main Memory KNN Processing

Bin Cui[1] Beng Chin Ooi[1] Jianwen Su[2] Kian-Lee Tan[1]

[1] Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore
{cuibin, ooibc, tankl}@comp.nus.edu.sg

[2] Department of Computer Science
University of California
Santa Barbara, CA 93106-5110
su@cs.ucsb.edu

ABSTRACT

In this paper, we present a novel index structure, called Δ-tree, to speed up processing of high-dimensional K-nearest neighbor (KNN) queries in main memory environment. The Δ-tree is a multi-level structure where each level represents the data space at different dimensionalities: the number of dimensions increases towards the leaf level which contains the data at their full dimensions. The remaining dimensions are obtained using *Principal Component Analysis*, which has the desirable property that the first few dimensions capture most of the information in the dataset. Each level of the tree serves to prune the search space more efficiently as the reduced dimensions can better exploit the small cache line size. Moreover, the distance computation on lower dimensionality is less expensive. We also propose an extension, called Δ^+-tree, that globally clusters the data space and then further partitions clusters into small regions to reduce the search space. We conducted extensive experiments to evaluate the proposed structures against existing techniques on different kinds of datasets. Our results show that the Δ^+-tree is superior in most cases.

1. INTRODUCTION

Many emerging database applications such as image, time series and scientific databases, manipulate high-dimensional data. In these applications, one of the most frequently used and yet expensive operations is to find objects in the database that are similar to a given query object. Nearest neighbor search is a central requirement in such cases.

There is a long stream of research on solving the nearest neighbor search problem, and many multidimensional indexes have been proposed [3, 4, 7, 8, 14, 16, 17, 18]. 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 [2, 6, 13, 15].

In main memory systems, distance computations and L2 cache misses contribute significantly to the overall cost. Several main memory indexing schemes have been designed to be *cache conscious* [13, 15]. However, these schemes are targeted at single or low dimensional data. Moreover, for high-dimensional data, distance calculations are computationally expensive [4]. Therefore an efficient main memory index should exploit the L2 cache effectively and minimize the distance computation to improve the performance.

In this paper, we propose a novel multi-tier index structure, called Δ-tree[1], that can facilitate efficient KNN search in main memory environment. Each tier in the Δ-tree represents the data space as clusters in 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 [12]. 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. We present the KNN search algorithm as well as the update algorithm for the Δ-tree. 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 and then partitions clusters into small regions before building the tree. We compare the proposed schemes against other known schemes including the M-tree [8], TV-tree [14], CR-tree [13], VA-file [16], iDistance [18] and Sequential Scan. Our study shows that the Δ^+-tree is superior in most cases.

An indirect contribution of this paper is the comparative study of the various indexing structures. To our knowledge, this is the first comprehensive performance study that involves so many high-dimensional structures. The results provide insight into the strengths and limitations of these schemes that will help researchers/practitioners to pick an appropriate scheme to adopt.

The remainder of this paper is organized as follows. In the

[1] Δ reflects the structure of the tree where nodes closer to the root index keys with lower dimensions, while those towards the leaf index keys with higher dimensions.

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

next section, we review some related work. Section 3 provides some background on the Principal Component Analysis. In Section 4, we introduce our newly proposed Δ-tree and Δ^+-tree. We also present the search and update operations on the structures. Section 5 reports the findings of an extensive experimental study conducted to evaluate the proposed schemes, and finally, we conclude in Section 6.

2. RELATED WORK

Many multi-dimensional structures have been proposed in the literature [3]. Here, we shall just review five of them that are used for comparison in our experimental study.

CR-tree. In [13], the authors proposed a cache-conscious version of the R-tree called the CR-tree. To pack more entries in a node, the CR-tree compresses MBR keys. It first represents the coordinates of an MBR key relatively to the lower left corner of its parent MBR. Then, it quantizes the relative coordinates to further cut off the less significant trailing bits. Consequently, the CR-tree becomes significantly wider and smaller than the ordinary R-tree. The experimental and analytical studies on the CR-tree showed that it performs faster than the R-tree. Since the CR-tree is based on the R-tree, it inherits its problem of not being scalable (in terms of number of dimensions). Moreover, quantization of MBRs incurs additional computational cost compared to the R-tree.

TV-tree. In [14], the authors proposed a tree-structure that avoids the dimensionality problem. The idea is to use a variable number of dimensions for indexing, adapting to the number of objects to be indexed, and to the current level of the tree. The TV-tree defines two kinds of dimensions, inactive dimensions and active dimensions. Since the TV-tree indexes the active dimensions and the number of active dimensions is usually small, the method saves space and leads to a larger fan-out. 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 different number of dimensions at different levels of the tree. Second, 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. Third, the number of active dimensions of the TV-tree can be large, which means that it still suffers from the dimensional scalability problem, while our structures always employ few dimensions in the upper levels.

M-tree. In [8], the authors proposed the height-balanced M-tree to organize and search large datasets from a generic metric space, where object proximity is defined by a distance function. In an M-tree, leaf nodes store all indexed objects, whereas internal nodes store the routing objects. For each routing object O_r, there is an associated pointer, denoted ptr(T(O_r)), that references the root of a sub-tree, T(O_r), called the covering tree of O_r. All objects in T(O_r) are within the distance r(O_r) from O_r, r(O_r) > 0. Finally, a routing object O_r is associated with a distance to P(O_r), its parent object, that is the routing object which references the node where the O_r entry is stored. This distance is not defined for entries in the root of the M-tree. An entry for a database object O_j in a leaf node is quite similar to that of a routing object. The strength of the M-tree lies in maintaining the pre-computed distance in the index structure. Thus, the number of distance computation can be low, making it a good candidate for main memory environment.

iDistance. In [18], the authors presented an efficient method for KNN search in a multi-dimensional space, called iDistance. 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 index 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 used a standard B^+-tree structure to index the data and KNN search be performed using one-dimensional range search. Since cache conscious B^+-tree has been studied [15], 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.

VA-file. In [16], the authors described a simple vector approximation scheme, call VA-file. The VA-file divides the data space into a 2^b rectangular cells. The scheme allocates a unique bit-string of length b for each cell, and approximates data points that fall into a cell by that bit-string. The VA-file itself is simply an array of these approximations. KNN searches are performed by scanning the entire approximation file, and by excluding the vast majority of vectors from the search (filtering step) based on these approximations. After the filtering step, a small set of candidates are then visited and the actual distances to the query point Q are determined. The VA-file has been shown to perform well for disk-based systems as it reduces the number of random I/Os. However, it incurs higher computational cost making it less attractive for main memory databases: besides computing the actual distances of candidate points, it has to decode the bit-string and compute all the lower and some upper bounds on the distance to the query point.

3. PRINCIPAL COMPONENT ANALYSIS

The Principal Component Analysis (PCA) [12] is a widely used method for transforming points in the original (high-dimensional) space into another (usually lower dimensional) space [5, 11]. It examines the variance structure in the dataset 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.

We shall briefly review how the principal components are computed. Let the dataset contains N D-dimensional points. Let A be the $N \times D$ data matrix where each row corresponds to a point in the dataset. We first compute the mean and covariance matrix of the dataset 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 P(x_1, x_2, ..., x_D) is transformed into $V \times P = (y_1, y_2, \ldots, y_D)$. To reduce the dimensionality of a dataset 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 $O(N \cdot D^2)$ time [10].

Suppose we have two points, P and Q, in the dataset 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. $dist(P_{k1}, Q_{k1}) \leq dist(P_{k2}, Q_{k2}) \quad 0 < k1 < k2 \leq D$, where $dist(p, q)$ denotes the distance between two points p and q (See [5] 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 $k \ll D$ [5].
3. The above properties also hold for new points that are added into the dataset (despite the fact that they do not contribute to the derivation of the *eigenmatrix*) [5]. Thus, when a new point is added to the dataset, we can simply apply the *eigenmatrix* and map the new data from the original space into the new PCA space.

In [5], PCA is employed to organise 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.

4. 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. 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 KNN search in main memory databases. For the rest of this paper, we assume that the dataset consists of D-dimensional points and use the Euclidean distance as the metric distance function.

4.1 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 non-trivial 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 3, it has several very nice properties.

4.1.1 The structure of Δ-tree

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

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

Figure 1 shows a Δ-tree, which is essentially a multi-tier 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 is 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 shall 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 shall 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 $\prod((x_1, \ldots, x_D), m_i)$.

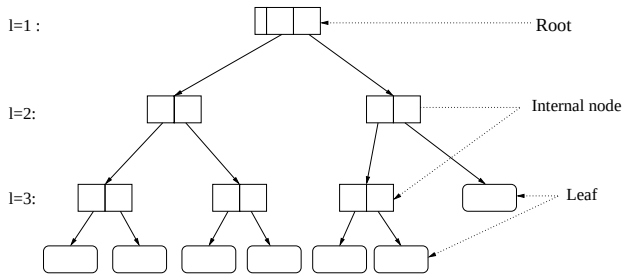


Figure 1: The Δ-tree

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 $C_1, C_2, \ldots C_n$. 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_1 contains points that are clustered together in m_1 dimensions in the PCA-Space. At level 2, C_i is partitioned into $C_{i1}, C_{i2}, \ldots, C_{in}$ sub-clusters using the m_2 dimensions of the points in C_i in the PCA-Space. This process is repeated for each sub-cluster at each subsequent level l where each subsequent clustering process operates on the m_l dimensions of the points in the sub-cluster 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.

Figure 2 shows an example of an internal node of Δ-tree with three sub-clusters. 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 sub-cluster. Each entry is a 4-tuple (c_l, r, num, ptr), where c_l is the center of the sub-cluster obtained at level l, r is the radius of the sub-cluster, num is the number of points in the sub-cluster, and ptr is a pointer to the next level node. The root node has the same structure as an internal node ex-

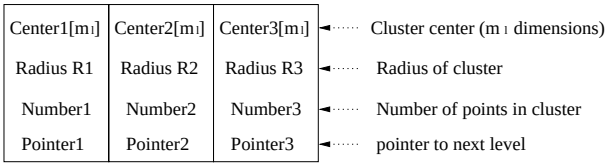


Figure 2: The structure of internal node

cept that it has to maintain additional information on the dataset. This is captured as a triple (L, m, $eigenmatrix$) header, where L represents the number of projection levels, $m = (m_1, m_2, \ldots, m_{L-1})$ 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 dataset after PCA processing.

We note that the Δ-tree can be used to prune the search space effectively. Recall (in property 1) 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. 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 K-th NN, then it can be pruned away). More importantly, the lower dimensionality at upper levels of the 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 fan-out of the tree. For the number of levels, we adopt a simple strategy: we estimate the number of levels based on the fan-out of a node, e.g., given a set of N points, and a fan-out 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 [12]. Let the variance of the j-th 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.

Figure 3 shows the effect of cumulative variation of a dataset after applying PCA. Two 64-dimensional datasets are used: a real dataset containing color histograms extracted from the Corel Database [1] and a synthetic dataset that is uniformly distributed. It is clear that for the real dataset (where the data is skewed), the first few dimensions in the PCA-Space is sufficient to capture the variation , e.g.,

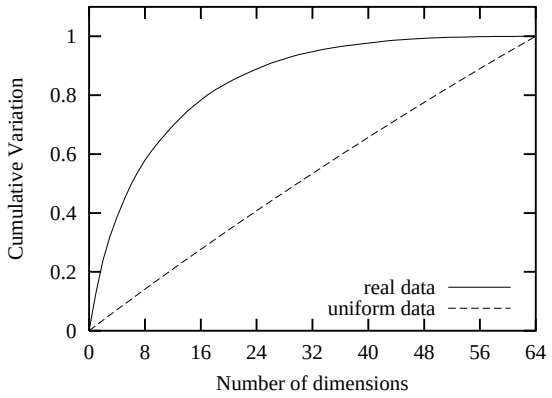


Figure 3: The proportion of cumulative variation

the first 8 dimensions already capture the 60% of variation in the data. On the other hand, for uniformly distributed data, all the dimensions have similar variance. Suppose we have 5 projection levels, the resultant m_l for each level is shown in Table 1.

Dataset	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$
Real data	2	4	8	17	64
Uniform data	12	24	37	50	64

Table 1: m_l for different level

We note that for efficiency reason, 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 sub-cluster 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 sub-clusters 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.

4.1.2 The Δ-tree construction

Figure 4 shows the algorithm for constructing a Δ-tree for a given dataset. We have adopted a top down approach. At first, routine **PCA()** transforms the dataset into the PCA space (line 1). We treat the whole dataset 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 **R_insert(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 **R_insert(node, pC, lev)**, we partition the data of the cluster pC into K sub-clusters (by K-means).

Algorithm Buildtree(dataset, tree)
Input: the high-dimensional dataset
Output: Δ-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);

Figure 4: The algorithm of building a Δ-tree

However, this partitioning is performed only on the m_{lev} dimensions of the cluster. For each sub-cluster, in lines 3-4, we fill the information on the center and radius into the corresponding entry in *node*. If the number of points in a sub-cluster fit 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.

4.1.3 KNN search algorithm

To facilitate KNN search, we employ two separate data structures. The first is a priority queue that maintains entries in non-descending 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 K-th NN and the query point is used to prune away points that are further away.

We summarize the algorithm in Figure 5. In the first stage, we initialize the priority queue, KNN list and the pruning distance (lines 1-3). After that 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 sub-cluster in PCA space (distance is computed with $m_{child.lev}$ dimensions using P_dist()). If the distance is shorter than the pruning distance, we do as follows. If the child node is an internal node, it means that there is a further partitioning of the space into sub-clusters, and we insert the node into the queue(lines 10-11). Otherwise, the child must be a leaf node, 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

Algorithm KNNSearch(QueryPoint, tree, K)
Input: Δ-tree, query point Q, K
Output: K nearest neighbors

1. Queue = NewPriorityQueue();
2. KNN = NewKNN();
3. prune_dist = ∞;
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);

Figure 5: KNN search algorithm for Δ-tree

equal to the distance between the query point and the K-th nearest neighbor candidate.

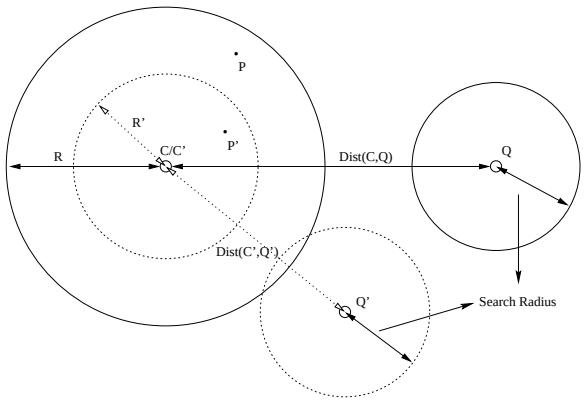


Figure 6: Prune with projection distance

We illustrate the effect of pruning a cluster in Figure 6. In the figure, Q represents the query point, and the solid circle represents the search space bounded by its distance to the current K-th NN. Q' represents the transformed point in a lower dimensional space. Note that the search radius remains the same, and the search region is denoted by the dotted circle. Consider a cluster centered on C (region bounded by solid circle). Since we only consider distance here, suppose the cluster on the transformed low-dimensional space is also centered at C (denoted C') but the region is smaller (bounded by dotted line). Suppose a point P in the former is transformed to a point P' in the latter. We have the following equation:

$$dist(P,Q) \geq dist(P',Q') \geq dist(C',Q') - R'.$$

If $dist(C',Q') - R'$ is larger than the search radius, then all the points in this cluster cannot be nearer than the current K-th NN (since $dist(P,Q)$ must be larger than search radius also). Thus, we can prune away the cluster. As the

values of C' and R' are already maintained in the Δ-tree, the computational cost is low, making the proposed scheme efficient.

4.1.4 Updating 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 sub-cluster. 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.

Algorithm Update(newpoint, tree)
Input: the Δ-tree, newpoint P

```
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)
```

Figure 7: Update algorithm for Δ-tree

The algorithmic description of the update operation is shown in Figure 7. 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)) for the leaf node to insert the point by always selecting the nearest sub-cluster 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. 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 pre-determined threshold, say 50% of the original data size, we rebuild the tree regardless of the precision of 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 pre-defined threshold, we rebuild the tree. The decision factor of this method is related to the distribution of newly inserted points.

4.2 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 dataset is globally correlated, i.e., most of the variations can be captured by a few principle components in the transformed space. For real datasets that are typically not globally correlated, more clusters may have to be searched. Second, the complete dataspace has to be examined for each query. 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 distance 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 $Dist_{min}$ and $Dist_{max}$ 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:

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

where $f = (Dist_{max} - Dist_{min})/k$. Figure 8 shows an example of a cluster with six regions.

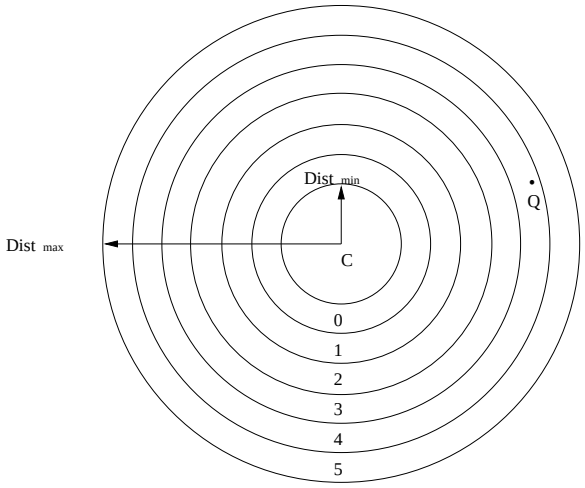


Figure 8: Cluster partitioning and searching

3. Given a query point, we can order the regions in non-descending 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., $Dist_{min}$, $Dist_{min}+f$, ..., $Dist_{min} + (k-1)f$) of the region. For example, consider the query point Q in Figure 8. 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 the query point and region 3, then 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 shall refer to this new structure as the Δ^+-tree. Figure 9(b) shows a Δ^+-tree structure. The whole dataset has two global clusters and we partition the cluster into 3 regions. For comparison purpose, we also show the Δ-tree (Figure 9(a)).

As described above, the operations on the Δ^+-tree are quite similar to those on the Δ-tree. For all the operations, we start from the nearest global cluster and region, and then traverse the subtree. This process continues with other clusters, while cluster or regions containing points further than the K-th NN are not traversed.

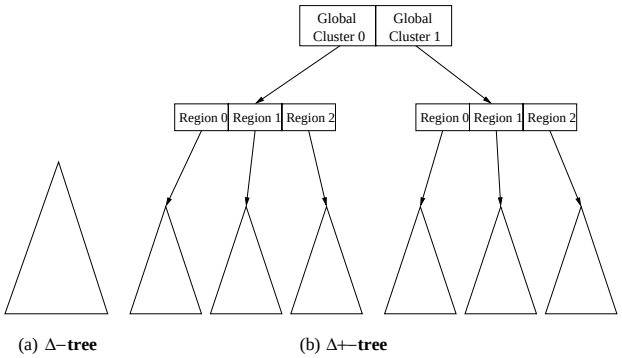


Figure 9: The structure of Δ-tree variants

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 would eventually be needed if the global clusters are no longer optimal. As we shall see in our experimental study, the index remains effective for a high percentage of newly added points.

5. 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 [2] and distance computation for KNN search over 100 different queries. All the experiments are conducted on SUN E450 machine with 450 MHz CPU, 4 GB RAM and 2 MB L2 cache. The machine is running SUN OS 5.7. The whole trees are loaded into the main memory before each experiment begins.

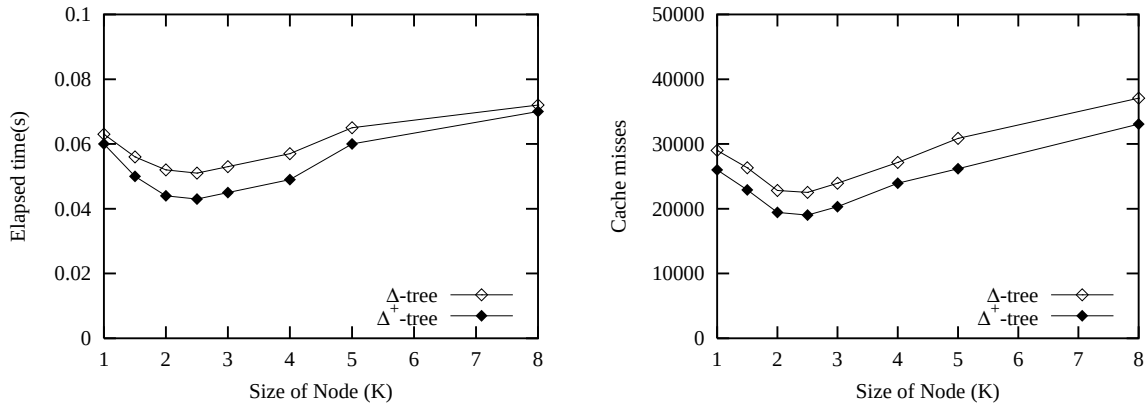
For the Δ^+-tree, it has two extra parameters: the number of global clusters and regions. When both these parameters are set to 1, the Δ^+-tree becomes the Δ-tree. The optimal values of the parameters may vary with different datasets. We observed that the performance of Δ^+-tree improves with a larger number of clusters. This is expected as the search space can be efficiently reduced as we have explained in the previous section. For simplicity, we set the default number of clusters and regions to be 10 and 5 respectively throughout the performance study.

5.1 Comparing Δ-tree and Δ^+-tree

We conducted an extensive performance study to tune the two proposed schemes for optimality. Due to space constraint, we shall only present one representative set that studies the effect of node size. We used a real-life dataset consisting of 64 dimensional color histograms extracted from 70,000 color images obtained from the Corel Database [1]. We note that we do not know the number of clusters in the dataset, i.e., the number of clusters may not exactly match the apriori number of clusters used in the Δ^+-tree (which is 10).

In this experiment, we vary the node size from 1 KB to 8 KB. Although the optimal node size for single-dimensional

[2]We use the *Perfmon* tool [9] to count L2 cache misses.



(a) Elapsed time

(b) cache misses

Figure 10: NN search for different node size

datasets has been shown to be the cache line size [15], this choice of node size is not optimal in high-dimensional cases – the L2 cache line size on a typical modern machine 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. [13] 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.

Figure 10 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 fan-out 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 fan-out of the tree is also small. As a result, more nodes will have to be accessed. At the same time, more TLB misses[3] will be incurred when we traverse the tree. There is the case that even if the node size is smaller than that of a memory page, each node access incurs one TLB miss if the address mapping is not in TLB. We observe that as the node size increases, the number of accessed nodes decreases. The performance becomes optimal when 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 Figure 10, clearly demonstrate the superiority of the Δ^+-tree over the Δ-tree: it is about 15% better than Δ-tree. This is because the Δ^+-tree only need to search less data space compared to the Δ-tree. First, the Δ^+-tree globally partitions the dataset 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.

We found that the actual cache miss cost is only around 10% of the overall time cost. This is the case in our studies as we did 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. Our investigation shows that the number of cache misses for an independent query can be twice as much, i.e., 20%.

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 node size determined above.

5.2 Comparison with other structures

In this section, we compare the Δ^+-tree with some existing methods on different datasets, such as the CR-tree, TV-tree, M-tree, VA-file, iDistance and Sequential Scan. To ensure a fair comparison, we optimize these methods for main memory indexing purposes such as tuning the node size[4]. We shall only present the optimal result of each structure.

5.2.1 *On uniformly distributed dataset*

In this experiment, we first generate a uniformly distributed dataset with up to 64 dimensions. The data size is 1,000,000 points. We shall present the results for NN queries only, as KNN queries show similar performance for uniformly distributed datasets. The results are summarized in Figure 11.

From the figure, we can see that Sequential Scan is the best scheme for high-dimensional data space ($D > 30$). This is expected for uniformly distributed dataset as the distance to the NN is too large, and we must scan all the data even if we have an index. Since we need to access all the data when

[3]The translation lookahead buffer (TLB) is an essential part of modern processors, which keeps the mapping from logical memory address to a physical memory address. The cost of TLB miss is about 100 cycles.

[4]One parameter of TV-tree is the number of active dimensions α, e.g. optimal α is around 20 for real dataset.

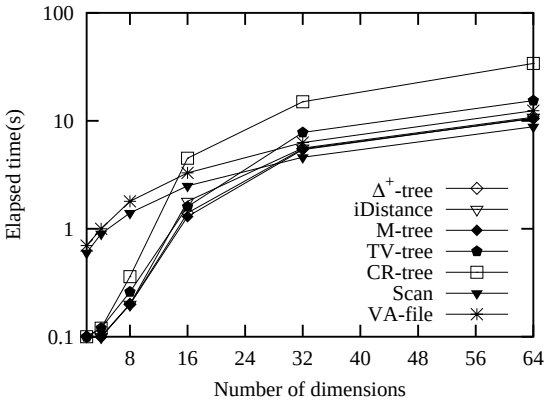


Figure 11: The comparisons of NN search time

the number of dimensions is high, the tree structures cause more TLB misses and cache misses – accessing the internal nodes is the overhead. Not surprisingly, the VA-file is worse than Sequential Scan. As mentioned, the VA-file needs three cost overhead: decoding cost, computational cost and actual data access. In disk-based environments, disk I/O cost is dominant, so the cost overhead does not affect the performance of the VA-file much. But, in main memory systems, the search cost is bounded by CPU cost. Although the VA-file can reduce the cache misses compared to Sequential Scan, the cost overhead overwhelms the gain. The performances of the Δ^+-tree, M-tree and iDistance are quite similar, because they are all distance-based trees. When the data are uniformly distributed, PCA has no value, because all the dimensions have the same weight. When the dimension is low (2 or 4), the CR-tree is efficient and performs as well as the Δ^+-tree. However, since the R-tree is not scalable to high-dimensionality, the CR-tree's performance starts to degrade as the number of dimensions increases; additionally it incurs higher computational cost (to uncompress the MBRs). The performance of the TV-tree is only better than the CR-tree when the dimensionality is high, because although the TV-tree uses reduced dimensions, its structure is similar to the R-tree and dimensionality reduction is not efficient for uniformly distributed dataset.

For the rest of this paper, we shall only focus on the TV-tree, M-tree, iDistance, Sequential Scan and the Δ^+-tree. The VA-file is worse than Sequential Scan, and is expected to perform worse when the dataset is skew because of bit conflict. The CR-tree performs poorly for high-dimensional datasets, so we also excluded it.

5.2.2 On clustered dataset

In many applications, data points are often correlated in some ways. In this set of experiments, we evaluate the M-tree, TV-tree, iDistance, Sequential Scan and Δ^+-tree on clustered datasets. 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 [5] to generate the clusters in subspaces of different orientations and dimensionalities. All datasets have 1,000,000 points.

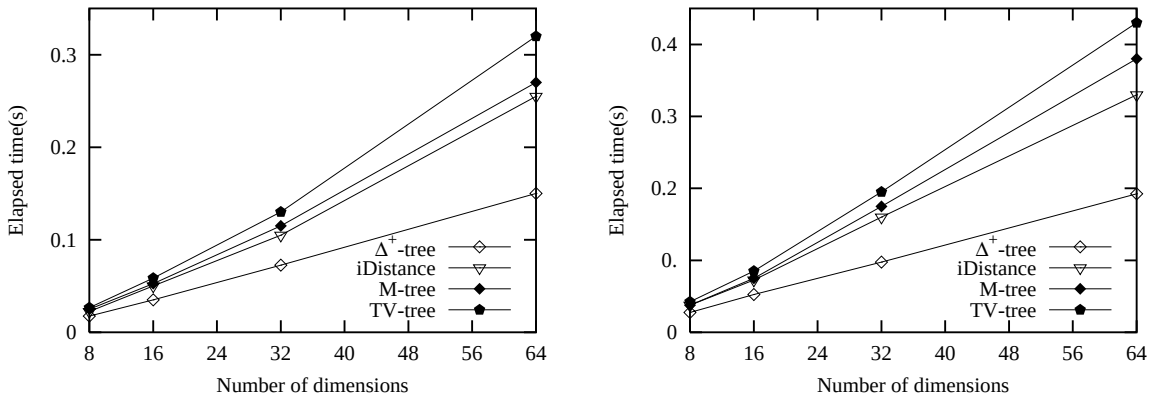
Figure 12 shows the results of KNN search as we vary the number of dimensions from 8 to 64; and the details of performance of Nearest Neighbor search, such as distance computation and cache misses are shown in Figure 13. Because Sequential Scan performs poorly and all these tree structures achieve a speedup by a factor of around 15 over the Sequential Scan, we remove 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 dataset to get the nearest neighbors, and the cost is proportional to the size and dimension of the dataset. When the dataset 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% faster than the other three methods, especially when the dimensionality is high. The Δ^+-tree has two advantages over the M-tree. First, the Δ^+-tree reduces the cache misses compared to the M-tree. Because we index different levels of projections of the dataset, 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 M-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 M-tree. Second, the computational cost of Δ^+-tree is also smaller than the M-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% 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% of the actual data distance computation. Additionally, we build the tree from top down exploiting the global clustering compared with local partition of M-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. Mapping high-dimensional data point to 1-dimensional point is less efficient in filtering data compared to the Δ^+-tree.

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

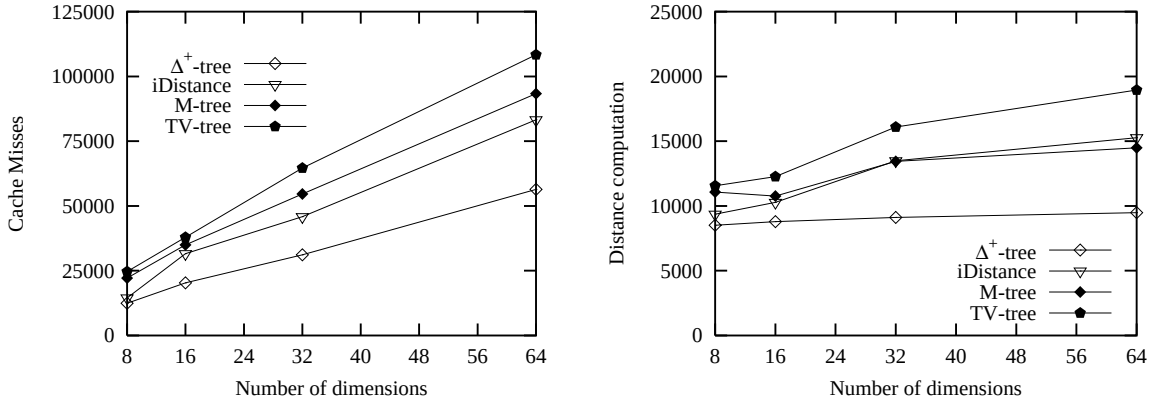
In the next experiment, we test the scalability of performance with respect to the data size. We fix the number of dimensions at 64, and vary the dataset size from 1,000,000



(a) Elapsed time (NN)

(b) Elapsed time (10-NN)

Figure 12: KNN search for clustered datasets



(a) Cache misses

(b) Distance computation

Figure 13: Details of NN search performance

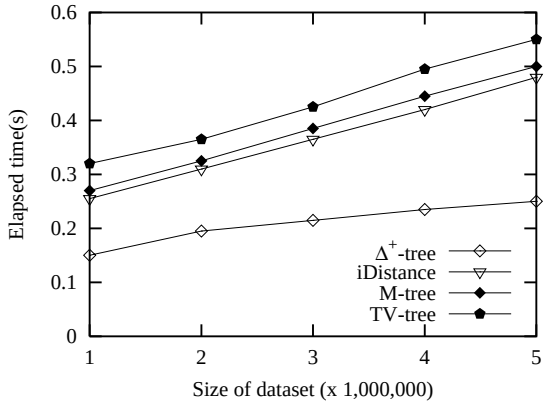
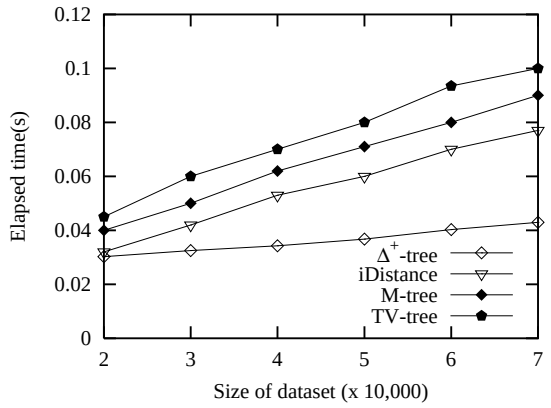


Figure 14: Scalability of index structures

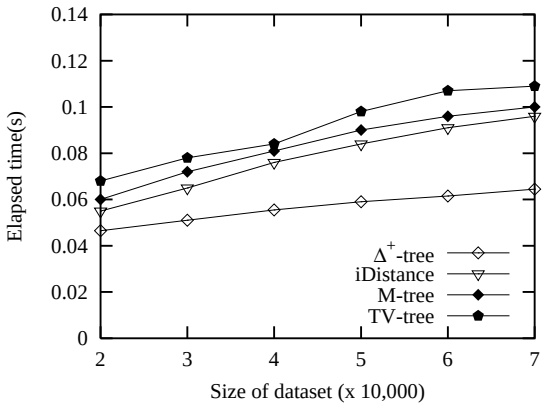
to 5,000,000 points. The result of NN search is shown in Figure 14. Because the cost of Sequential Scan increases almost linearly and is more expensive than other tree structures, we exclude it from the figure. The M-tree, TV-tree, Δ^+-tree and iDistance remain very effective for large datasets, demonstrating their scalability. In fact, the gap between these schemes and Sequential Scan widens as the dataset size increases. The relative performance between the three schemes remain largely the same as earlier experiments: the Δ^+-tree is the best scheme, followed by the iDistance, M-tree and finally TV-tree.

5.2.3 *On real dataset*

In this experiment, we evaluate the various schemes on the same real-life dataset used in section 5.1. First, we test the performance of KNN search. The performance is quite similar to the clustered datasets. The tree-based methods are at least 10 times faster than Sequential Scan because the real dataset is generally skew. As such, we will not present the results for Sequential Scan.

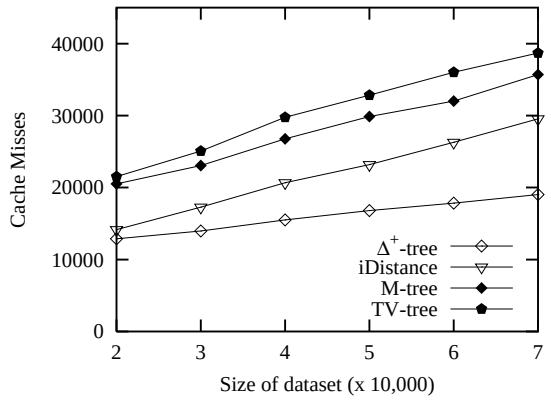


(a) Elapsed time (NN)

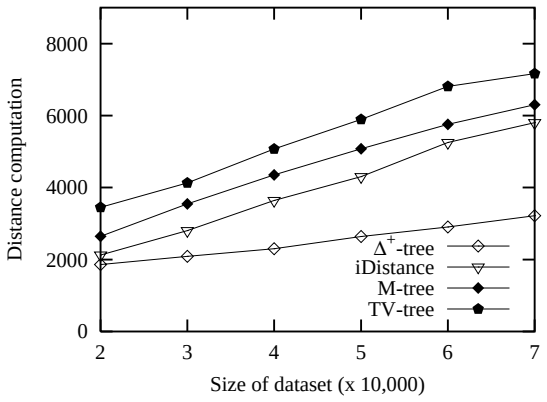


(b) Elapsed time (10-NN)

Figure 15: KNN search for real dataset



(a) Cache misses



(b) Distance computation

Figure 16: Details of NN search performance

The comparisons among the Δ^+-tree, M-tree, TV-tree and iDistance are shown in Figure 15. The Δ^+-tree is about 50% faster than other methods for NN search, and the performance of 10-NN search is quite similar to the NN search. In Figure 16, the comparisons of distance computation and cache misses also show that the Δ^+-tree performs best. These results clearly show the effectiveness of the Δ^+-tree even if the number of clusters employed may not match that of the dataset. The M-tree is poor because it incurs more computation and it uses all the dimensions in the internal nodes resulting in more cache misses. The iDistance is worse than the Δ^+-tree due to its 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 memory indexing, this translates into higher computational cost and number of cache misses.

5.3 On effect of updates

In the last experiment, we study the effect of updates on the Δ^+-tree. For the Δ^+ tree, we evaluated four versions: Δ^+-tree represents the version that is based on the proposed update algorithm (without rebuilding); Δ^+-rebuild represents the version that always rebuild the tree upon insertion[5]; 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 20% for the latter two schemes.

We used the real dataset for this experiment. We randomly select the data from the image dataset regardless of the data distribution in advance. We first build the tree structure with 35,000 data. Subsequently, we insert up to 100% more new data points. We recorded the execution times of NN search on the new datasets after 20% of newly

[5]This is ideal and represents the optimal Δ^+-tree. In our experiments, we only rebuild the tree at the point of where we run the experiments, e.g., after every additional 20% of newly inserted points.

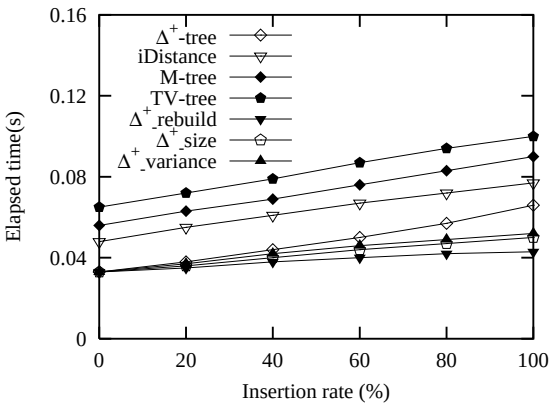


Figure 17: Effect of update operation

inserted data. The results are shown in Figure 17.

First, we observe that all the Δ^+-tree versions outperform the other tree structures. This clearly demonstrate 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 only around 15% even for 100% 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 is more than the threshold, it rebuilds the sub-tree 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. 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 datasets.

6. CONCLUSION

In this paper, we have addressed the problem of accessing high-dimensional data in main memory databases. We presented an efficient novel index method, called Δ-tree, for KNN search. The Δ-tree employs hierarchical clustering and multiple level of projections of points to allow nodes to better fit into the L2 cache. Thus, the search process can be accelerated by reducing computational cost and cache misses. We also proposed an extension, called Δ^+-tree, that further partitions a cluster into regions. We conducted extensive experiments to evaluate the Δ-tree and Δ^+-tree against several known techniques, and the results showed that our technique is superior in most cases. To our knowledge, this is the first work on conducting extensive experimental study on high-dimensional indexes using both real and synthetic data in the context of main memory databases.

7. REFERENCES

[1] *Corel Image Features*. available from http://kdd.ics.uci.edu.

[2] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *Proc. of the ACM SIGMOD Conference*, pages 163–174, 2001.

[3] C. Bohm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. In *ACM Computing Surveys*, pages 322–373, 2001.

[4] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. of the ACM SIGMOD Conference*, pages 357–368, 1997.

[5] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *Proc. 26th VLDB Conference*, pages 89–100, 2000.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proc. of the ACM SIGMOD Conference*, pages 139–150, 2001.

[7] Y. S. Chen, Y. P. Hung, and C. S. Fuh. Fast algorithm for nearest neighbor search based on a lower bound tree. In *Proc. 8th ICCV Conference*, pages 446–453, 2001.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. 24th VLDB Conference*, pages 194–205, 1997.

[9] R. Enbody. *Perfmon: Performance Monitoring Tool*. available from http://www.cps.msu.edu/ enbody/perfmon.html, 1999.

[10] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[11] J. Hui, B. C. Ooi, H. Shen, C. Yu, and A. Zhou. An adaptive and efficient dimensionality reduction algorithm for high-dimensional indexing. In *Proc. 19th ICDE Conference*, 2003.

[12] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, 1986.

[13] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proc. of the ACM SIGMOD Conference*, pages 139–150, 2001.

[14] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[15] J. Rao and K. Ross. Making B+-trees cache conscious in main memory. In *Proc. of the ACM SIGMOD Conference*, pages 475–486, 2000.

[16] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th VLDB Conference*, pages 194–205, 1998.

[17] C. Yu. *High-dimensional indexing. Lecture Notes in Computer Science 2341*. Springer-Verlag, 2002.

[18] C. Yu, B. C. Ooi, K. L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *Proc. 27th VLDB Conference*, pages 421–430, 2001.