Diagonal Ordering: A new approach to high-dimensional KNN processing

Jing Hu **Bin Cui** **Hengtao Shen**

Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543,
Email: {hujing, cuibin, shenht}@comp.nus.edu.sg

Abstract

In this paper, we propose Diagonal Ordering, a new technique for K-Nearest-Neighbor (KNN) search in a high-dimensional space. Our solution is based on data clustering and a particular sort order of the data points, which is obtained by "slicing" each cluster along the diagonal direction. In this way, we are able to transform the high-dimensional data points into one-dimensional space and index them using a B^+-tree structure. KNN search is then performed as a sequence of one-dimensional range searches. Advantages of our approach include: (1) irrelevant data points are eliminated quickly without extensive distance computations; (2) the index structure can effectively adapt to different data distributions; (3) online query answering is supported, which is a natural byproduct of the iterative searching algorithm. We conduct extensive experiments to evaluate the Diagonal Ordering technique and demonstrate its effectiveness.

Keywords: High-dimensional index structure, Diagonal Ordering, KNN query

1 Introduction

Over the last two decades, high-dimensional vector data has become widespread to support many emerging database applications such as multimedia, time series analysis and medical imaging. In these applications, the search of similar objects is often required as a basic functionality. To facilitate the similarity search, feature vectors are extracted from the original complex objects. Such feature vectors may bed tens (e.g. color histograms) or even hundreds of dimensions (e.g. astronomical indexes). The similarity of two objects is then measured as the distance between their corresponding feature vectors. Therefore, a similarity search can be translated into a nearest neighbor query in a high-dimensional vector space.

There is a long stream of research on solving the high-dimensional nearest neighbor problem, and many indexing techniques have been proposed (Berchtold et al. 1996, Bohm et al. 2001, Bozkaya et al. 1997, Ciaccia et al. 1997, Goldstein et al. 2000, Sakurai et al. 2000, Weber et al. 1998, Yu et al. 2001). The conventional approach addressing this problem is to adapt low-dimensional index structures to the requirements of high-dimensional indexing, e.g., the X-tree (Berchtold et al. 1996). Although this approach appears to be a natural extension to the low-dimensional indexing techniques, they suffer from the curse of dimensionality greatly. Another approach is to speed up the sequential scan by compressing the original feature vectors. A typical example is the VA-file (Weber et al. 1998). VA-file overcomes the dimensionality curse to some extent, but it cannot adapt to different data distributions effectively. These observations motivate us to come out with our own solution, the Diagonal Ordering technique.

Diagonal Ordering behaves similar to the Pyramid technique (Berchtold et al. 1998) and iDistance (Yu et al. 2001). It works by clustering the high-dimensional data space and organizing vectors inside each cluster based on a particular sorting order, the *diagonal order*. The sorting process also provides us a way to transform high-dimensional vectors into one-dimensional values. It is then possible to index these values using a B^+-tree structure and perform the KNN search as a sequence of range queries.

Using the B^+-tree structure is an advantage for our technique, as it brings all the strength of a B^+-tree, including fast search, dynamic update and height-balanced structure. It is also easy to graft our technique on top of any existing commercial relational databases.

Another feature of our solution is that the *diagonal order* enables us to derive a tight lower bound on the distance between two feature vectors. Using such a lower bound as the pruning criteria, KNN search is accelerated by eliminating irrelevant feature vectors without extensive distance computations.

Finally, our solution is able to support online query answering, i.e. obtain an approximate query answer by terminating the query search process prematurely. This is a natural byproduct of the iterative searching algorithm.

We implemented the Diagonal Ordering method and conducted an extensive performance study to evaluate its effectiveness. Our results show that the proposed technique is able to provide superior performance than sequential scan, the X-tree, iDistance and VA-file on various data sets.

The rest of this paper is organized as follows: In section 2, we review some related works. Then we introduce and discuss our new approach, the Diagonal Ordering, in section 3. The experimental evaluation of our approach is presented in section 4 and section 5 concludes the whole paper.

2 Related Works

In the recent literature, a variety of index structures have been proposed to facilitate high-dimensional nearest-neighbor search. Existing techniques mainly focus on three different approaches.

The first approach is based on data space partitioning, which include the R*-tree (Beckmann et al. 1990), the X-tree (Berchtold et al. 1996), the SR-tree (Katayama et al. 1997), the TV-tree (Lin et al. 1994)and many others. Although these methods generally perform well at low dimensionality, their performance degrades as dimensionality increases and the degradation can be so bad that sequential scanning becomes more efficient. This phenomenon has been termed "the curse of dimensionality". Generally speaking, nearest neighbor search in high-dimensional spaces becomes difficult due to the following two important factors:

- as the dimensionality increases, the distance to the nearest neighbor approaches the distance to the farthest neighbor.
- the computation of the distance between two feature vectors becomes significantly processor intensive as the number of dimensions increases.

The second approach is to represent original feature vectors using smaller, approximate representations. A typical example is the VA-file (Weber et al. 1998). The VA-file accelerates the sequential scan by the use of data compression. It divides the data space into a 2^b rectangular cells, where b denotes a user specified number of bits. By allocating a unique bit-string of length b to each cell, the VA-file approximates feature vectors using their containing cell's bit -string. KNN search is then equivalent to a sequential scan over the vector approximations with some look-ups to the real vectors. The performance of the VA-file has been reported to be linear to the dimensionality. However, the VA-file cannot adapt effectively to different data distributions, mainly due to its unified cell partitioning scheme.

One dimensional transformations provide another direction for high-dimensional indexing. iDistance (Yu et al. 2001) is such an efficient method for KNN search in a high-dimensional data space. It relies on clustering the data and indexing the distance of each feature vector to the nearest reference point. 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. The choice of partition and reference point provides the iDistance technique with degrees of freedom most other techniques do not have. The experiment shows that iDistance can provide good performance through appropriate choice of partitioning scheme. However, when dimensionality exceeds 30, the equal distant phenomenon kicks in, and hence the effectiveness of pruning degenerates rapidly.

3 Diagonal Ordering

In this section, we propose our algorithm for KNN search in high-dimensional data spaces. The algorithm is based on a particular order of the data set, the *diagonal order*, which is defined in the first part of this section.

For the rest of this section, we assume that the underlying database is a set of feature vectors in a d-dimensional vector space. The Euclidean distance is used as our metric distance function.

3.1 The Diagonal Order

To alleviate the impact of the dimensionality curse, it helps to reduce the dimensionality of feature vectors. For real world applications, data sets are often skewed and uniform distributed data sets rarely occur in practice. Some features are therefore more important than the other features. It is then intuitive that a good ordering of the features will result in a more focused search. We employ Principle Component Analysis (Jolliffe 1986) to achieve such a good ordering and the first few features are favored over the rest.

The high-dimensional feature vectors are then grouped into a set of clusters by existing techniques, such as K-Means, CURE (Guha et al. 1998) or BIRCH (Zhang et al. 1996). In this paper, we just applied the clustering method proposed in iDistance (Yu et al. 2001). We approximate the centroid of each cluster by estimating the median of the cluster on each dimension through the construction of a histogram. The centroid of each cluster is used as the cluster reference point.

Without loss of generality, let us suppose that we have identified m clusters, $C_0, C_1, \cdots, C_m$, with corresponding reference points, $O_0, O_1, \cdots, O_m$ and the first d' dimensions are selected to split each cluster into $2^{d'}$ partitions. We are able to map a feature vector $P(p_1, \cdots, p_d)$ into an index key key as follows:

$$key = i * l_1 + j * l_2 + \sum_{t=1}^{d'} |p_t - o_t|$$

where P belongs to the j-th partition of cluster C_i with reference point $O_i(o_1, o_2, \cdots, o_d)$, l_1 and l_2 are constants to stretch the data range. The definition of the *diagonal order* follows from the above mapping directly:

Definition 1 The Diagonal Order ($\prec$)
For two vectors $P(p_1, \cdots, p_d)$ and $Q(q_1, \cdots, q_d)$ with corresponding index keys key_p and key_q, the predict $P \prec Q$ is true if and only if $key_p < key_q$.

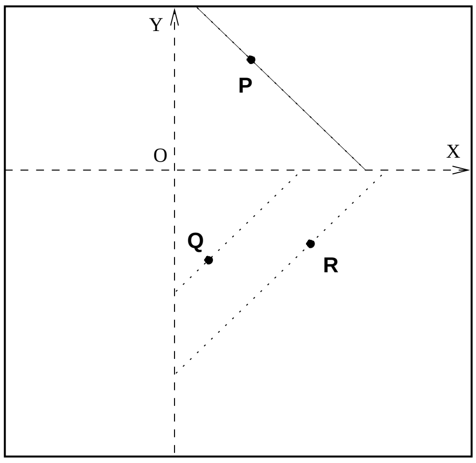


Figure 1: The Diagonal Ordering Example

Basically, feature vectors within a cluster are sorted first by partitions and then in the diagonal direction of each partition. As in the two-dimensional example depicted in Figure 1, $P \prec Q$, $P \prec R$ because P is in the second partition and Q, R are in the fourth partition. $Q \prec R$ because $|q_x - o_x| + |q_y - o_y| < |r_x - o_x| + |r_y - o_y|$. In other words, Q is nearer to O than R in the diagonal direction.

Note that for high-dimensional feature vectors, we usually choose d' to be a much smaller number than d; otherwise, the exponential number of partitions inside each cluster will become intolerable. Once the order of feature vectors has been determined, it is a simple task to build a B^+-tree upon the database. We also employ an array to store the m reference points. Minimum Bounding Rectangle (MBR) of each cluster is also stored.

3.2 Query Search Regions

The index structure of Diagonal Ordering requires us to transform a d-dimensional KNN query into one-dimensional range queries. However, a KNN query is equivalent to a range query with the radius set to the k-th nearest neighbor distance, therefore, knowing how to transform a d-dimensional range query into one-dimensional range searches suffices our needs.

Suppose that we are given a query point Q and a search radius r, we want to find out search regions that are affected by this range query. As the simple two-dimensional example depicted in Figure 2 shows, a query sphere may intersect several partitions and the computation of the area of intersection is not trivial. We first have to examine which partitions are affected, then determine the ranges inside each partition.

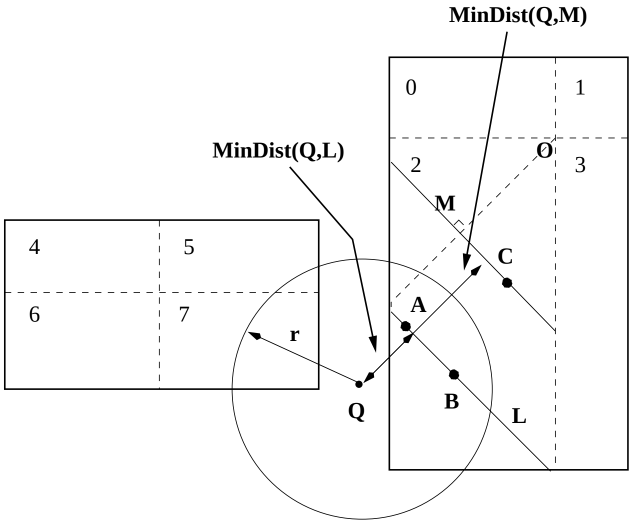


Figure 2: Search Regions

Knowing the reference point and the MBR of each cluster, the MBR of each partition can be easily obtained. Calculating minimum distance from a query point to an MBR is not difficult. If such a minimum distance is larger than the search radius r, the whole partition of data points are out of our search range, therefore, can be safely pruned. For example, in Figure 2, partitions 0, 1, 3, 4 and 6 need not to be searched. Otherwise, we have to do a further investigation for points inside the affected partitions. Since we have sorted all data points by the diagonal order, the test whether a point is inside the search regions has to be based on the transformed value.

In Figure 2, points $A(a_x, a_y)$ and $B(b_x, b_y)$ are on the same line segment L. Note that $|a_x - o_x| + |a_y - o_y| = |b_x - o_x| + |b_y - o_y|$. This equality is not a coincidence. In fact, any point $P(p_x, p_y)$ on the line segment L share the same value of $|p_x - o_x| + |p_y - o_y|$. In other words, line segment L can be represented by this value, which is exactly the $\sum_{t=1}^{d'} |p_t - o_t|$ component of the transformed key value.

If the minimum distance from a query point Q to such a line segment is larger than the search radius r, all points on this line segment are guaranteed not inside the current search regions. For example, in Figure 2, the minimum distance from line segment M to

Q is larger than r, from which we know that point C is outside the search regions. The exact representation of C need not to be accessed. On the other hand, the minimum distance from L to Q is less than r. A and B therefore become our candidates. It also can be seen in Figure 2 that some of the candidates are hits, others are false drops due to the lossy transformation of feature vectors. Then, an access to the real vectors is necessary to filter out all the false drops.

Before we extend the two-dimensional example to a general d-dimensional case, let us define the signature of a partition first:

Definition 2 Partition Signature
For a partition X with reference point $O(o_1, \cdots, o_d)$, its signature $S(s_1, \cdots, s_{d'})$ satisfies the following condition

$$\forall\ P(p_1, \cdots, p_d) \in \text{X},\ i \in [1, d'],\ s_i = \frac{|p_i - o_i|}{p_i - o_i}$$

This signature is shared by all vectors inside the same partition. In other words, if $P(p_1, \cdots, p_d)$ and $P'(p'_1, \cdots, p'_d)$ belongs to the same partition with signature $S(s_1, \cdots, s_{d'})$, then

$$\forall\ \text{i} \in [1, d'],\ s_i = \frac{|p_i - o_i|}{p_i - o_i} = \frac{|p'_i - o_i|}{p'_i - o_i}$$

Now we are ready to derive the formula for MinDist(Q, L) in a d-dimensional case:

Theorem 1 MinDist(Q, *key*)
For a query vector $\text{Q}(q_1, \ldots, q_d)$ and a set of feature vectors with the same *key* value, the minimum distance from Q to these vectors is given as follows:

$$\frac{|\sum_{t=1}^{d'}(s_t * (q_t - o_t)) - (key - i * l_1 - j * l_2)|}{\sqrt{d'}}.$$

Proof: All points $P(p_1, \cdots, p_d)$ with the same *key* value must reside in a same partition. Assume that they belong to the j-th partition of the i-th cluster and the partition has the signature $S(s_1, \cdots, s_{d'})$. In order to determine the minimum value of $f = (p_1 - q_1)^2 + \cdots + (p_{d'} - q_{d'})^2$, whose variables are subjected to the constraint relation $s_1 * (p_1 - o_1) + \cdots + s_{d'} * (p_{d'} - o_{d'}) + i * l_1 + j * l_2 = key$, Lagrange Multiplier is the standard technique to solve this problem and the result is, $\frac{[\sum_{t=1}^{d'}(s_t * (q_t - o_t)) - (key - i * l_1 - j * l_2)]^2}{d'}$. Note that $\sqrt{f}$ is always less than or equal to $dist(P, Q)$. Thus, $\frac{|\sum_{t=1}^{d'}(s_t * (q_t - o_t)) - (key - i * l_1 - j * l_2)|}{\sqrt{d'}}$ is a lower bound to $dist(P, Q)$.

Back to our original problem where we need to identify search ranges inside each affected partition, this is not difficult once we have the formula for MinDist. More formally:

Lemma 1 Search Range
For a search sphere with query point $\text{Q}(q_1, \ldots, q_d)$ and search radius r, the range to be searched within an affected partition j of cluster i in the transformed one-dimensional space is

$$[i * l_1 + j * l_2 + \textstyle\sum_{t=1}^{d'}(s_t * (q_t - o_t)) - r * \sqrt{d'},$$
$$i * l_1 + j * l_2 + \textstyle\sum_{t=1}^{d'}(s_t * (q_t - o_t)) + r * \sqrt{d'}]$$

where partition j has the signature $S(s_1, \cdots, s_{d'})$.

3.3 KNN Search Algorithm

Let us denote the k-th nearest neighbor distance of a query vector Q as KNNDist(Q). Searching for k nearest neighbors of Q is then the same as a range query with the radius set to KNNDist(Q). However, KNNDist(Q) cannot be predetermined with 100% accuracy. In Diagonal Ordering, we adopt an iterative approach to solve the problem. Starting with a relatively small radius, we search the data space for nearest neighbors of Q. The range query is iteratively enlarged until we have found all the k nearest neighbors. The search stops when the distance between the query vector Q and the farthest object in Knn (answer set) is less than or equal to the current search radius r.

Algorithm KNN
Input: Q, CurrentKNNDist(initial value:∞), r
Output: Knn (K nearest neighbors to Q)
step: Increment value for search radius
$sv : i * l_1 + j * l_2 + \sum_{t=1}^{d'}(s_t * (q_t - o_t))$

```
KNN(Q, step, CurrentKNNDist)
  load index
  initialize r
  while (r < CurrentKNNDist)
    r = r + step
    for each cluster i
      for each partition j
        if searched[i][j] is false
          if partition j intersects
          sphere(Q,r)
            searched[i][j] = true
            lnode = LocateLeaf(sv)
            lb = LowerBound(sv,r)
            ub = UpperBound(sv,r)
            lp[i][j] = Downwards(lnode,lb)
            rp[i][j] = Upwards(lnode,ub)
        else
          if lp[i][j] not null
            lb = LowerBound(sv,r)
            lp[i][j] = Downwards(
                       lp[i][j]->left,lb)
          if rp[i][j] not null
            ub = UpperBound(sv,r)
            rp[i][j] = Upperwards(
                       rp[i][j]->right,ub)
```

Figure 3: Main KNN Search Algorithm

Figures 3 and 4 summarize the algorithm for KNN query search. The KNN search algorithm uses some

Algorithm Upperwards
Input: LeafNode, UpperBound
Output: LeafNode

```
Upwards(node, ub)
  if the first entry in node has
  a key value larger than ub
    return node->left
  for each entry E inside node
    calculate dist(E, Q)
    update CurrentKNNDist
    update Knn
  if end of partition is reached
    return null
  else if the last entry in node has
  a key value less than ub
    return Upperwards(node->right, ub)
  else
    return node
```

Figure 4: Routine Upwards

important notations and routines. We shall discuss them briefly before examining the main algorithm. *CurrentKNNDist* is used to denote the distance between Q and its current k-th nearest neighbor during the search process. This value will eventually converge to $KNNDist(Q)$. $searched[i][j]$ indicates whether the j-th partition in cluster i has been searched before. $sphere(Q, r)$ denotes the sphere with radius r and centroid Q. *lnode*, *lp*, and *rp* store pointers to the leaf nodes of the B^+-tree structure. Routine *LowerBound* and *UpperBound* return values $i * l_1 + j * l_2 + \sum_{t=1}^{d'} (s_t * (q_t - o_t)) - r * \sqrt{d'}$ and $i * l_1 + j * l_2 + \sum_{t=1}^{d'} (s_t * (q_t - o_t)) + r * \sqrt{d'}$ correspondingly. As a result, lower bound *lb* and upper bound *ub* together represent the current search region. Routine *LocateLeaf* is a typical B^+-tree traversal procedure which locates a leaf node given the search value. Routine *Upwards* and *Downwards* are similar, we will only focus on *Upwards*. Given a leaf node and an upper bound value, routine *Upwards* first decides whether entries inside the current node are within the search range. If so, it continues to examine each entry to determine whether they are among the k nearest neighbors, and update the answer set Knn accordingly. By following the right sibling link, *Upwards* calls itself recursively to scan upwards, until the index key value becomes larger than the current upper bound or the end of the partition is reached.

Figure 3 describes the main routine for our KNN search algorithm. Given query point Q and the step value for incrementally adjusting the search radius r, KNN search commences by assigning an initial value to r. It has been shown that starting the range query with a small initial radius keeps the search space as tight as possible, and hence minimizes unnecessary search. r is then increased gradually and the query results are refined, until we have found all the k nearest neighbors of Q.

For each enlargement of the query sphere, we look for partitions that are intersected with the current sphere. If the partition has never been searched but intersects the search sphere now, we begins by locating the leaf node where Q may be stored. With the current one-dimensional search range calculated, we then scan *upperwards* and *downwards* to find the k nearest neighbors. If the partition was searched before, we can simply retrieve the leaf node where the scan stopped last time and resume the scanning process from that node onwards.

The whole search process stops when the *CurrentKNNDist* is less than r, which means further enlargement will not change the answer set. In other words, all the k nearest neighbors have been identified. The reason is that all data spaces within *CurrentKNNDist* range from Q have been searched and any point outside this range will have a distance larger than *CurrenKNNDist* definitely. Therefore, the KNN algorithm returns k nearest neighbors of query point correctly.

A natural byproduct of this iterative algorithm is that it can provide fast approximate k nearest neighbor answers. In fact, at each iteration of the algorithm KNN, there are a set of k candidate NN vectors available. These tentative results will be refined in subsequent iterations. If a user can tolerate some amount of inaccuracy, the processing should be terminated prematurely to obtain quick approximate answers.

3.4 Analysis and Comparison

In this section, we are going to do a simple analysis and comparison between Diagonal Ordering and iDistance. iDistance shares some similarities with our technique in the following ways:

- Both techniques map high-dimensional feature vectors into one-dimensional values. KNN query is evaluated as a sequence of range queries over the one-dimensional space.
- Both techniques rely on data space clustering and defining a reference point for each cluster.
- Both techniques adopt an iterative querying approach to find the k nearest neighbors to the query point. The algorithms support online query answering and provide approximate KNN answers quickly.

iDistance is an adaptive technique with respect to data distribution. However, due to the lossy transformation of data points into one-dimensional values,

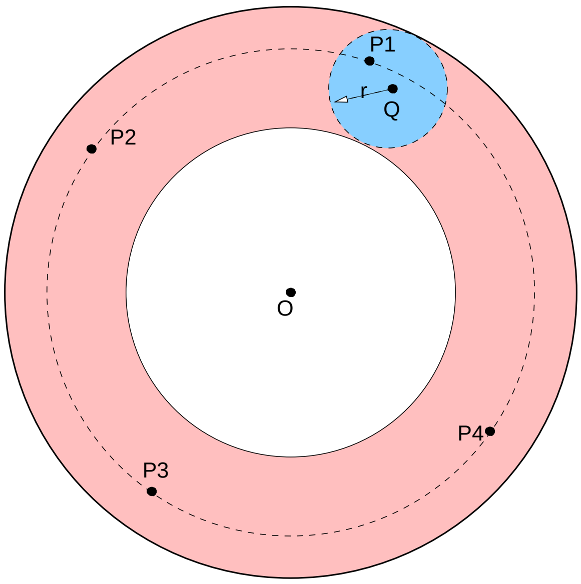


Figure 5: iDistance Search Regions

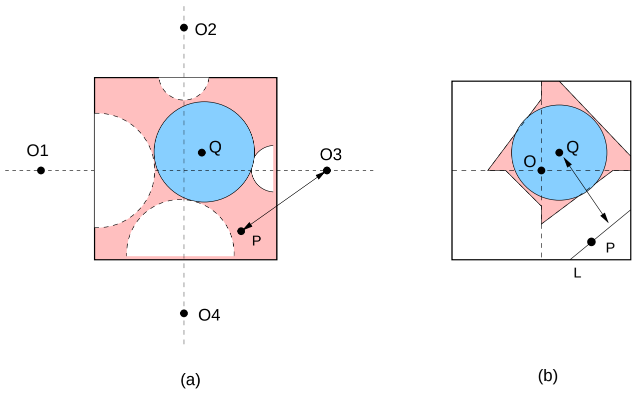


Figure 6: iDistance and Diagonal Ordering (1)

false drops occur very significantly during the iDistance search. As illustrated in the two-dimensional example depicted in Figure 5, in order to search the query sphere with radius r and query point Q, iDistance has to check all the shaded areas. Apparently, $P2$, $P3$, $P4$ are all false drops. iDistance can't eliminate these false drops because because they have the same transformed value (distance to the reference point O) as $P1$. Our technique overcomes this difficulty by diagonally ordering data points within each partition. Let us consider two simple two-dimensional cases to demonstrate the strengths of Diagonal Ordering.

Case one The query point Q is near to the reference point O. Figure 6(a) shows the affected data space by this query sphere in iDistance. Comparing to iDistance, the affected area by the same query sphere for our technique is much smaller. As shown in Figure 6(b), P is considered to be a candidate in iDistance since $dist(P, O3) \in [dist(Q, O3) - r, dist(Q, O3) + r]$; whereas P is pruned by Diagonal Ordering, for the minimum distance from Q to line L is already larger than r.

Case two The query point Q is far from the reference point O. As shown in Figure 7(a), the affected area in iDistance is still quite large, which almost consists of half of the data space. Again, we observe that the affected space under our technique is a lot smaller comparing to Figure 7(a). This is because partition 0, 2 and 3 are already out of the search region. We only need to consider partition 1 and diagonal ordering helps us reduce the affected space further.

Back to a general example where the cluster does not contain the query point but intersects with the query sphere. Figure 8(a) and Figure 8(b) demon-

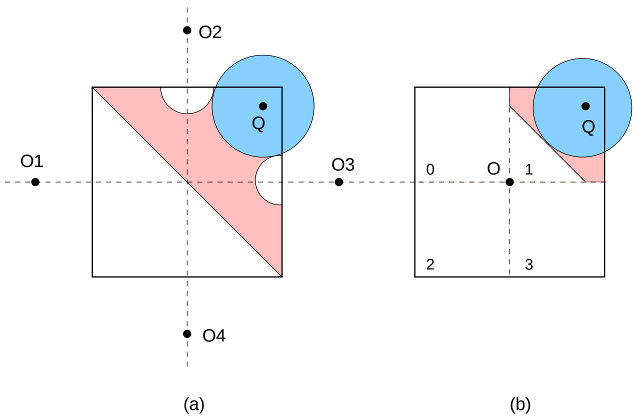


Figure 7: iDistance and Diagonal Ordering (2)

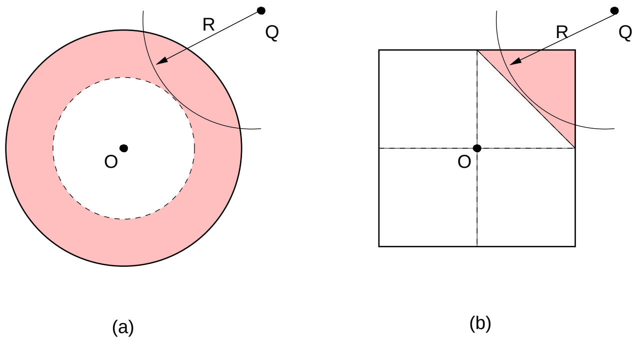


Figure 8: iDistance and Diagonal Ordering (3)

strate the affected space for iDistance and Diagonal Ordering correspondingly. It is easy to see that our technique outperforms iDistance as well.

4 Performance Evaluation

To demonstrate the practical impact of Diagonal Ordering and to verify our theoretical results, we performed an extensive experimental evaluation of our technique and compared it to the following competitive techniques:

- iDistance
- X-tree
- VA-file
- Sequential Scan

4.1 Experimental Setup

Our evaluation comprises both real and synthetic high-dimensional data sets. The synthetic data sets are either uniformly distributed or clustered. We use a method similar to that of (Chakrabarti et al. 1996) to generate the clusters in subspaces of different orientations and dimensionalities. The real data set contains 32 dimensional color histograms extracted from 68,040 images. All the following experiments were performed on a Sun E450 machine with 450Mhz CPU, running SUN OS 5.7. Page size is set to 4KB. The performance is measured in terms of the average disk page access, and the CPU time over 100 different queries. For each query, the number of nearest neighbor to search is 10 unless otherwise stated.

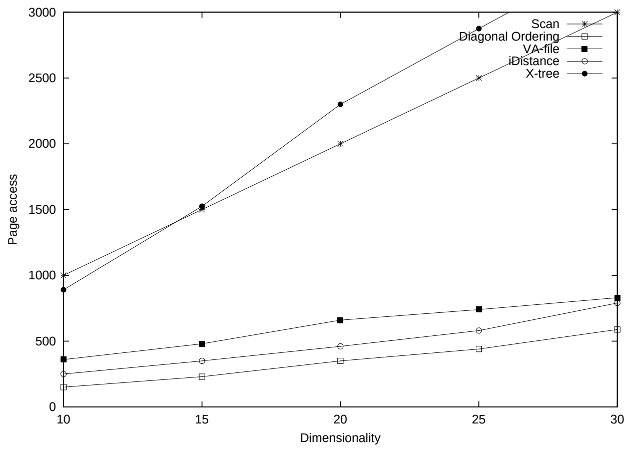
4.2 Performance behavior over dimensionality

In our first experiment, we determined the influence of the data space dimension on the performance of KNN queries. For this purpose, we have created five 100K clustered data sets with the dimensionality 10, 15, 20, 25 and 30 to run our experiments.

Figure 9 demonstrates the efficiency of the Diagonal Ordering technique as we increase dimensionality. It is shown that Diagonal Ordering outperforms other methods in terms of disk page access and CPU cost.

During the index construction, Diagonal Ordering performs clustering and partitioning, which helps to prune faster and access less IO pages. On the other hand, VA-file cannot make full use of the clustering characteristics and perform worse than Diagonal Ordering. However, as the dimensionality increases, the gap between the performance of VA-file and Diagonal Ordering becomes smaller. This is mainly because it is more and more difficult to find a good clustering scheme as the dimensionality keeps growing.

The iDistance technique also relies on the efficiency of clustering and partitioning of the data space.



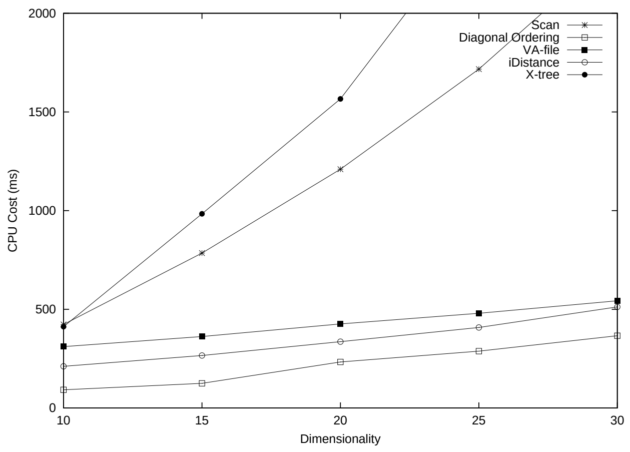


Figure 9: Performance Behavior over Data Size

Diagonal Ordering performs better because it can eliminate more false drops during the querying process, as we have presented in section 3.4. In figure 9, Diagonal Ordering achieves averagely 30% improvement over iDistance.

It is also observable that the efficiency of query processing using the X-tree rapidly decreases with increasing dimensions. When the dimensionality is higher than 15, almost all vectors inside the data set are completely scanned. From this point on, the querying cost of the X-tree grows linearly and even become worse than a sequential scan, which is mainly due to the X-tree index traversal overhead. The reason is that the X-tree employs rectangular MBRs to partition the data space; whereas high-dimensional MBR tends to overlap with each other significantly. As a result, the X-tree cannot prune effectively in high-dimensional data space and incurs a high querying cost.

MBR is also used by Diagonal Ordering, but in a different way. First, MBR is used to represent the data space of each partition. It is not involved in the partition process at all. The generated MBRs are guaranteed not to overlap with each other. Second, the partition process is working on a d'-dimensional space instead of the original d-dimensional space. By using Principle Component Analysis, these MBRs of size $2 * d'$ can still capture the most characteristics of each partition. Therefore, in our case, using MBR in the pruning process is still valid and effective.

4.3 Performance behavior over data size

In this experiment(c.f. Figure 10), we measured the performance behavior with varying number of data points. We performed 10NN queries over the 16-dimensional clustered data space and varied the data size from 50,000 to 300,000.

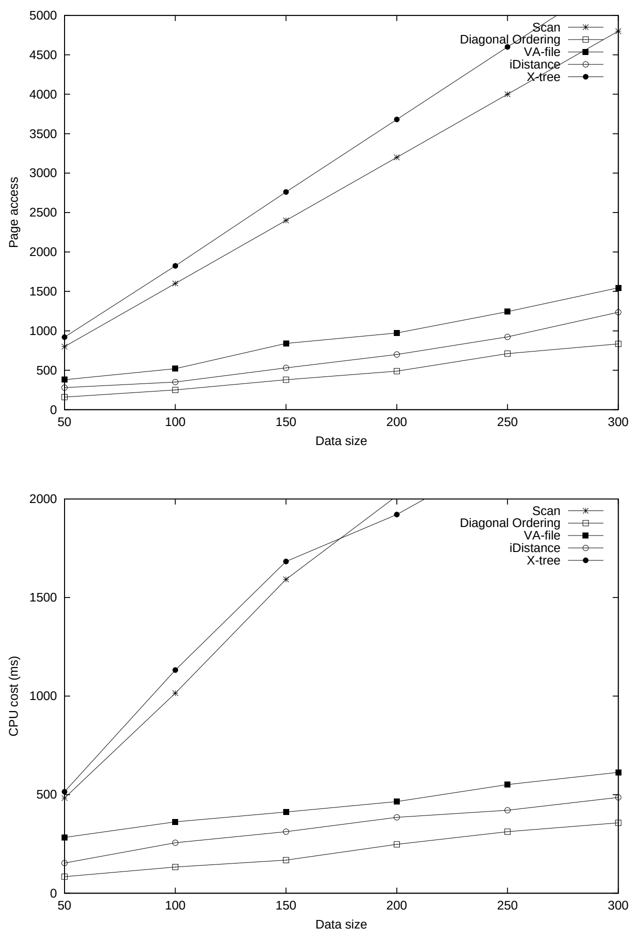


Figure 10: Performance Behavior over Data Size

Figure 10 shows the performance of query processing in terms of page access and CPU cost. It is evident that Diagonal Ordering outperforms the other four methods significantly. We also noticed that the X-tree has exhibited an interesting phenomenon in Figure 10(b): the performance of the X-tree is worse than a sequential scan when the size of database is small ($size < 150K$) and slightly better than a sequential scan when the size of the database becomes large. This is because the expected nearest neighbor distance decreases as the size of the data set increases. A smaller KNNDist(Q) will help the X-tree to achieve a better pruning effect such that less parts of the X-tree will be traversed and the CPU cost could be improved.

4.4 Performance behavior over K

In this series of experiments, we used the real data set extracted from 68,040 pixel images. The effects of an increasing value of K in a K nearest neighbor search are tested. Figure 11 demonstrate the experimental results when K ranges from 10 to 100. Among these indexes, the cost of the X-tree is still most expensive. The performance of iDistance and VA-file are pretty close to each other. In fact, the pruning effect of iDistance keeps degenerating as the dimensionality increases and it is finally caught up by the VA-file when the dimensionality exceeds 30. As shown in figure 11, Diagonal Ordering still retains a good performance. The smarter partitioning scheme and the pruning effectiveness of Diagonal Ordering helps to benefit more from the skewness of the color histograms.

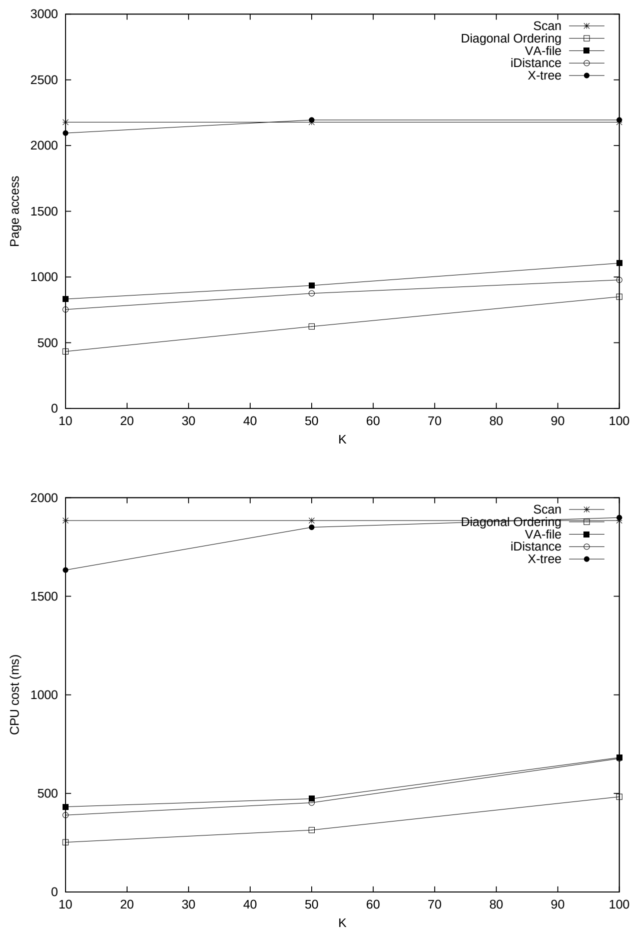


Figure 11: Performance Behavior over K

5 Conclusion

In this paper, we have addressed the problem of KNN query processing for high-dimensional data. We presented an efficient index method, called Diagonal Ordering, for KNN search, and can be easily extended to support other queries, such as range query. Diagonal Ordering applied a particular sort order of the data points along the diagonal direction, and also utilized

Principle Component Analysis and data clustering to adapt to different data distributions. We derived a lower distance bound as the pruning criteria, which accelerates the KNN query processing further. Extensive experiments were conducted and the results show that Diagonal Ordering is an efficient method for high-dimensional KNN searching. We also show that Diagonal Ordering can achieve a better performance than many other indexing techniques.

6 Acknowledgments

We would like to thank BengChin Ooi for his helpful comments and discussion on this paper.

References

N. Beckmann, H.-P. Kriegel R. Schneider, B. Seeger The R*-tree: An efficient and robust access method for points and rectangles, Proc. 1990 ACM SIGMOD International Conference on Management of Data, pp. 322–331.

S. Berchtold, C. Bohm, H. V. Jagadish, H. P. Kriegel, J. Sander (2000), Independent quantization: An index compression technique for high-dimensional data spaces, Proc. 16th ICDE Conference, pp. 577-588.

S. Berchtold, C. Bőhm, H-P. Kriegel The pyramid-technique: Towards breaking the curse of dimensionality, Proc. 1998 ACM SIGMOD International Conference on Management of Data, pp. 142–153.

S. Berchtold, D. A. Keim, H. P. Kriegel (1996), The x-tree: An index structure for high-dimensional data, Proc. 22th VLDB Conference, pp. 28-39.

C. Bohm, S. Berchtold, D. Keim (2001), Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases, ACM Computing Surveys 33(3), pp. 322-373.

T. Bozkaya, M. Ozsoyoglu (1997), Distance-based indexing for high-dimensional metric spaces, Proc. of the ACM SIGMOD Conference, pp. 357-368.

K. Chakrabarti, S. Mehrotra (2000), Local dimensionality reduction: A new approach to indexing high dimensional spaces, Proc. 26th VLDB Conference, pp. 89-100.

P. Ciaccia, M. Patella, P. Zezula (1997), M-tree: An efficient access method for similarity search in metric spaces, Proc. 24th VLDB Conference, pp. 194–205.

J. Goldstein, R. Ramakrishnan (2000), Contrast plots and p-sphere tree: Space vs. time in nearest neighbor searches, Proc. 26th VLDB Conference, pp. 429–440.

S. Guha, R. Rastogi, K. Shim, (1998), Cure: an efficient clustering algorithm for large databases, Proc. ACM SIGMOD International Conference on Management of Data.

I. T. Jolliffe (1986), Principle Component Analysis, Springer-Verlag

N. Katayama, S. Satoh (1997),The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries, Proc. of the ACM SIGMOD Conference, pp. 369-380.

K. Lin, H. V. Jagadish, C. Faloutsos (1994), The TV-tree: An index structure for high-dimensional data, The VLDB Journal, Vol. 3(4), pp. 517-542.

Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima (2000), The a-tree: An index structure for high-dimensional spaces using relative approximation, Proc. 26th VLDB Conference, pp. 516-526.

R. Weber, H. J. Schek, S. Blott (1998), A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, Proc. 24th VLDB Conference, pp. 194–205.

C. Yu, B. C. Ooi, K. L. Tan, and H. V. Jagadish (2001), Indexing the distance: An efficient method to knn processing, Proc. 27th VLDB Conference, pp. 421–430.

T. Zhang, R. Ramakrishnan, M. Livny, (1996), BIRCH: an efficient data clustering method for very large databases, Proc. ACM SIGMOD International Conference on Management of Data.