Indexing the Distance: An Efficient Method to KNN Processing

Cui Yu[1] Beng Chin Ooi[1] Kian-Lee Tan[1] H. V. Jagadish[2]

[1]Department of Computer Science
National University of Singapore

[2]Department of Electrical Engineering & Computer Science
University of Michigan

Abstract

In this paper, we present an efficient method, called iDistance, for K-nearest neighbor (KNN) search in a high-dimensional space. iDistance partitions the data and selects a reference point for each partition. The data in each cluster are transformed into a single dimensional space based on their similarity with respect to a reference point. This allows the points to be indexed using a B^+-tree structure 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. We describe how appropriate choices here can effectively adapt the index structure to the data distribution. We conducted extensive experiments to evaluate the iDistance technique, and report results demonstrating its effectiveness.

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 high-dimensional 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 a large number of multidimensional indexes have been developed for this purpose. However, most of these structures are not *adaptive* with respect to data distribution. In consequence, they tend to perform well for some data sets and poorly for others.

In this paper, we propose a new technique for KNN search that can be adapted based on the data distribution. For uniform distributions, it behaves similar to techniques such as pyramid tree[1] and iMinMax[13], that are known to be good for such situations. For highly clustered distributions, it behaves as if a hierarchical clustering tree had been created especially for this problem.

Our technique, called iDistance, relies on partitioning the data and defining a reference point for each partition. We 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 use a standard B^+tree to index this distance. As such, it is easy to graft our technique on top of an existing commercial relational database.

Finally, our technique can permit the immediate generation of a few results while additional results are searched for. In other words, we are able to support *online query answering*: an important facility for interactive querying and data analysis.

The effectiveness of iDistance depends on how the data are partitioned, and how reference points are selected. We also present several partitioning strategies, as well as reference point selection strategies.

We implemented the iDistance method together with the partitioning and reference point selection strategies, and conducted an extensive performance study to evaluate their effectiveness. Our results show that the proposed schemes can provide fast initial response time without sacrificing on the quality of the answers. Moreover, through appropriate choice of partitioning scheme, the iDistance method can compute the complete answer set much faster than linear scan, iMinMax and A-tree [15].

In Section 2, we review some related works. Section 3 provides the background for metric-based KNN search. In Section 4, we present the proposed iDistance algorithm and in Section 5, we discuss the clus-

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

tering mechanisms and policies for selecting reference points. Section 6 presents our performance study, and the results. Finally, we conclude in 7 with directions for future work.

2 Related Work

Many indexing techniques have been proposed for nearest neighbor and approximate search in high-dimensional databases. Existing multi-dimensional indexes [4] such as R-trees [11] have been shown to be inefficient even for supporting range/window queries in high-dimensional databases; they, however, form the basis for indexes designed for high-dimensional databases [12, 17]. To reduce the effect of high dimensionalities, use of bigger nodes [3], dimensionality reduction [7] and filter-and-refine methods [2, 16] have been proposed. Indexes were also specifically designed to facilitate metric based query processing [6, 8]. However, linear scan remains one of the best strategies for KNN search [5]. This is because there is a high tendency for data points to be equidistant to query points in a high-dimensional space. More recently, the p-sphere tree [9] was proposed to support *approximate* nearest neighbor (NN) search where the answers can be obtained quickly but they may not necessarily be the nearest neighbors.

To reduce the effect of high-dimensionality as experienced by the R-tree, the A-tree stores virtual bounding boxes that approximate actual minimum bounding boxes instead of actual bounding boxes. Bounding boxes are approximated by their relative positions with respect to the parents' bounding region. The rationale is similar to that of the X-tree[3] – tree nodes with more entries may lead to less overlap and hence a more efficient search. However, maintenance of effective virtual bounding boxes is expensive should the database be very dynamic. A change in the parent bounding box will affect all virtual bounding boxes referencing it. Further, additional CPU time in deriving actual bounding boxes is incurred. Notwithstanding, the performance study in [15] shows that it is more efficient than the SR-tree [12] and the VA-file [16].

3 Background

To search for the K nearest neighbors of a query point q, the distance of the Kth nearest neighbor to q defines the minimum radius required for retrieving the complete answer set. Unfortunately, such a distance cannot be predetermined with 100% accuracy. Hence, an iterative approach that examines increasingly larger sphere in each iteration can be employed. The algorithm works as follows. Given a query point q, finding K nearest neighbors (NN) begins with a query sphere defined by a *relatively small* radius about q, $querydist(q)$. All data spaces that intersect the query sphere have to be searched. Gradually, the search region is expanded till all the K nearest points are found and all the data subspaces that intersect with the current query space are checked. The K data points are the nearest neighbors when further enlargement of query sphere does not introduce new answer points. We note that starting the search query with a small initial radius keeps the search space as tight as possible, and hence minimizes unnecessary search (had a larger radius that contains all the K nearest points been used).

4 Our Proposal

In this section, we propose a new KNN processing scheme, called iDistance, to facilitate efficient distance-based KNN search. The design of iDistance is motivated by the following observations. First, the (dis)similarity between data points can be derived with reference to a chosen reference point. Second, data points can be ordered based on their distances to a reference point. Third, distance is essentially a single dimensional value. This allows us to represent high-dimensional data in single dimensional space, thereby enabling reuse of existing single dimensional indexes such as the B^+-tree.

iDistance is designed to support similarity search – both similarity range search and KNN search. However, we note that similarity range search is a window search with a fixed radius and is simpler in computation than KNN search. Thus, we shall concentrate on the KNN operations from here onwards.

For the rest of this section, we assume a set of data points $Points$ in a unit d-dimensional space. Let $dist$ be a metric distance function for pairs of points. In our research, we use the Euclidean distance as the distance function, although other distance functions may be used for certain applications.

4.1 The Data Structure

In iDistance, high-dimensional points are transformed into a single dimensional space. This is done using a three step algorithm. In the first step, the high-dimensional data space is split into a set of partitions. In the second step, a reference point is identified for each partition. Without loss of generality, let us suppose that we have m partitions, $P_0, P_1, \ldots, P_{m-1}$ and their corresponding reference points, $O_0, O_1, \ldots, O_{m-1}$. We shall defer the discussion on how the partitions and reference points are obtained to the next section.

Finally, in the third step, all data points are represented in a single dimension space as follows. A data point $p(x_1, x_2, \ldots, x_d), 0 \leq x_j \leq 1, 1 \leq j \leq d$, has an index key, y, based on the distance from the nearest reference point O_i as follows:

$$y = i * c + dist(p, O_i)$$

where $dist(O_i, p)$ is a distance function that returns the distance between O_i and p, and c is some constant to stretch the data ranges. Essentially, c serves to partition the single dimension space into regions so that all points in partition P_i will be mapped to the range $[i * c, (i + 1) * c)$.

In iDistance, we also employ two data structures:

- A B^+-tree is use to index the transformed points to facilitate speedy retrieval. We use the B^+-tree since it is available in all commercial DBMS. In our implementation of the B^+-tree, leaf nodes are linked to both the left and right siblings [14]. This is to facilitate searching the neighboring nodes when the search region is gradually enlarged.
- An array is also required to store the m reference points and their respective nearest and farthest radii that define the data space.

Clearly, iDistance is lossy in the sense that multiple data points in the high-dimensional space may be mapped to the same value in the single dimensional space. For example, different points within a partition that are equidistant from the reference point have the same transformed value.

4.2 KNN Search in iDistance

Before we examine the KNN algorithm for iDistance, let us look at the search regions. Let O_i be the reference point of partition i, and $dist\_max_i$ and $dist\_min_i$ be the maximum and minimum distance between O_i and the points in partition P_i respectively. We note that the region bounded by the spheres obtained from these two radii defines the effective data space that need to be searched. Let q be a query point and $querydist(q)$ be the radius of the sphere obtained about q. For iDistance, in conducting NN search, if $dist(O_i, q) - querydist(q) \leq dist\_max_i$, then P_i has to be searched for NN points. The range to be searched within an affected partition in the single dimensional space is $[max(0, dist\_min_i), min(dist\_max_i, dist(O_i, q) + querydist(q))]$. Figure 1 shows an example. Here, for query point q_1, only partition P_1 needs to be searched; for query point q_2, both P_2 and P_3 have to be searched. From the figure, it is clear that all points along a fixed radius have the same value after transformation due to the lossy transformation of data points into distance with respect to the reference points. As such, the shaded regions are the areas that need to be checked.

Figures 2 to 4 summarize the algorithm for KNN with iDistance method. The algorithm is similar to its high-dimensional counterpart. It begins by searching a small 'sphere', and incrementally enlarges the search space till all K nearest neighbors are found. The search stops when the distance of the farthest object in S (answer set) from query point q is less than or equal to the current search radius r.

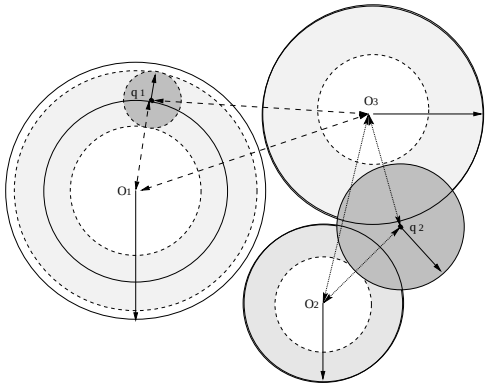


Figure 1: Search regions for NN queries q_1 and q_2.

The algorithm *iDistanceKNN* is highly abstracted. Before examining it, let us briefly discuss some of the important routines and notations. Routine **farthest(S,q)** returns the object in S farthest in distance from q. **sphere(q,r)** denote the sphere with radius r and centroid q. Since both routines Inward and Outward are similar, we shall only explain routine Inward. Given a leaf node, routine Inward examines the entries of the node to determine if they are among the K nearest neighbors, and updates the answers accordingly. We note that because iDistance is lossy, it is possible that points with the same values are actually not close to one another - some may be closer to q, while others are far from it. If the first element (or last element for Outward) of the node is contained in the query sphere, then it is likely that its predecessor with respect to distance from the reference point (or successor for Outward) may also be close to q. As such, the left (or right for Outward) sibling is examined. In other words, Inward (Outward) searches the space towards (away from) the reference point of the partition. The routine LocateLeaf is a typical B^+-tree traversal algorithm which locates a leaf node given a search value, and hence the detailed description of the algorithm is omitted.

We are now ready to explain the search algorithm. Searching in iDistance begins by scanning the auxiliary structure to identify the reference points whose data space (sphere area of partition) overlaps with the query region. The search starts with a small radius (*querydist*), and step by step, the radius is increased to form a bigger query sphere. For each enlargement, there are three main cases to consider.

1. The data space contains the query point, q. In this case, we want to traverse the data space sufficiently to determine the K nearest neighbors. This can be done by first locating the leaf node where q may be stored. Since this node does not necessarily contain points whose distance are closest to q compared to its sibling nodes, we need to search inward or outward from the reference point accordingly.

Algorithm iDistanceKNN (q, Δr, max_r)

```
r = 0;
Stopflag = FALSE;
initialize lp[], rp[], oflag[];
while r < max_r and Stopflag == FALSE
   r = r + Δr;
   SearchO(q, r);
```

Figure 2: iDistance KNN main search algorithm

Algorithm SearchO(q, r)

```
p_farthest = farthest(S,q)
if dist(p_farthest, q) < r and |S| == K
   Stopflag = TRUE; break;
for i = 0 to m − 1
   dis = dist(O_i, q);
   /* if O_i has not been searched before */
   if not oflag[i]
      if sphere(O_i, dist_max_i) contains q
         oflag[i] = TRUE;
         lnode = LocateLeaf(btree, i * c + dis);
         lp[i] = Inward(lnode, i * c + dis − r);
         rp[i] = Outward(lnode, i * c + dis + r);
      else if sphere(O_i, dist_max_i) intersects
         sphere(q, r)
         oflag[i] = TRUE;
         lnode = LocateLeaf(btree, dist_max_i);
         lp[i] = Inward(lnode, i * c + dis − r);
   else
      if lp[i] not nil
         l[i] = lp[i] → leftnode;
         lp[i] = Inward(l[i], i * c + dis − r);
      if rp[i] not nil
         r[i] = rp[i] → rightnode;
         rp[i] = Outward(r[i], i * c + dis + r);
```

Figure 3: iDistance KNN search algorithm: SearchO

Algorithm Inward($node$, $ivalue$)

```
for each entry e in node
   (e = e_j, j = 1, 2, ..., Number_of_entries)
   if |S| == K
      p_farthest = farthest(S,q)
      if dist(e, q) < dist(p_farthest, q)
         S = S − p_farthest;
         S = S ∪ e;
   else
      S = S ∪ e;
if e_1.key > ivalue
   lnode = node → leftnode;
   node = Inward(lnode, i * c + dis − r);
if end of partition is reached
   node = nil;
return(node);
```

Figure 4: iDistance KNN search algorithm: Inward

2. The data space intersects the query sphere. In this case, we only need to search inward since the query point is outside the data space.

3. The data space does not intersect the query sphere. Here, we do not need to examine the data space.

The search stops when the K nearest neighbors have been identified from the data subspaces that intersect with the current query sphere and when further enlargement of query sphere does not change the K nearest list. In other words, all points outside the subspaces intersecting with the query sphere will definitely be at a distance D from the query point such that D is greater than *querydist*. This occurs when the distance of the farther object in the answer set, S, from query point q is less than or equal to the current search radius r. Therefore, the answers returned by iDistance are of 100% accuracy.

An interesting by-product of iDistance is that it can provide approximate KNN answers quickly. In fact, at each iteration of algorithm iDistanceKNN, we have a set of K candidate NN points. These results can be returned to the users immediately and refined as more accurate answers are obtained in subsequent iterations. It is important to note that these K candidate NN points can be partitioned into two categories: those that we are certain to be in the answer set, and those that we have no such guarantee. The first category can be easily determined, since all those points with distance smaller than the current spherical radius of the query must be in the answer set. Users who can tolerate some amount of inaccuracy can obtain quick approximate answers and terminate the processing prematurely (as long as they are satisfied with the guarantee). Alternatively, max_r can be specified with an appropriate value to terminate $iDistanceKNN$ prematurely.

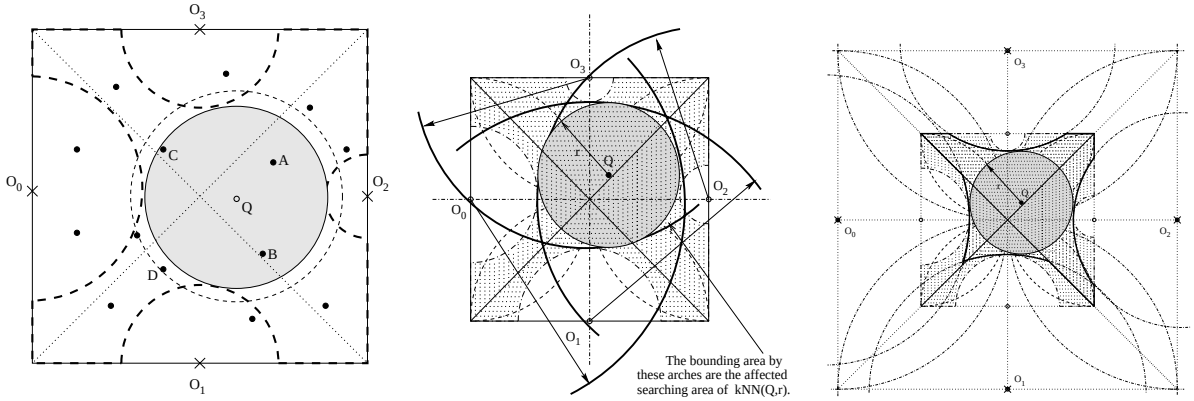
5 Selection of Reference Points and Data Space Partitioning

To support distance-based similarity search, we need to split the data space into partitions and for each partition, we need a reference point. In this section we look at some choices.

5.1 Equal Partitioning of Data Space

A straightforward approach to data space partitioning is to sub-divide it into equal partitions. In a d-dimensional space, we have $2d$ hyperplanes. The method we adopt is to partition the space into $2d$ pyramids with the centroid of the unit cube space as their top, and each hyperplane forming the base of each pyramid.[1] Now, we expect equi-partitioning to

[1]We note that the space is similar to that of the Pyramid method [1]. However, the rationale behind the design and the



(a) (Centroid, Closest) (b) (Centroid, Farthest) (c) (External point, Closest)

Figure 5: Space partitioning methods.

be effective if the data are uniformly distributed. We study the following possible reference points, where *the actual data space of hyperspheres do not overlap.*

Centroid of hyperplane, Closest Distance. The centroid of each hyperplane can be used as a reference point, and the partition associated with the point contains all points that are nearest to it. Figure 5(a) shows an example in a 2-dimensional space of a query region and the affected space.

For a query point along the central axis, the set of points retrieved along the axis is exactly the same as in a pyramid tree. When dealing with query and data points off the axis, the sets of points are not exactly identical, due to the curvature of the hypersphere as compared to the partitioning along axial hyperplanes in the case of the pyramid tree: nonetheless, these sets are likely to have considerable overlap.

Centroid of hyperplane, Farthest Distance. The centroid of each hyperplane can be used as a reference point, and the partition associated with the point contains all points that are farthest from it. Figure 5(b) shows an example in 2-dimensional space. As shown, the affected area can be greatly reduced (as compared to the closest distance counterpart).

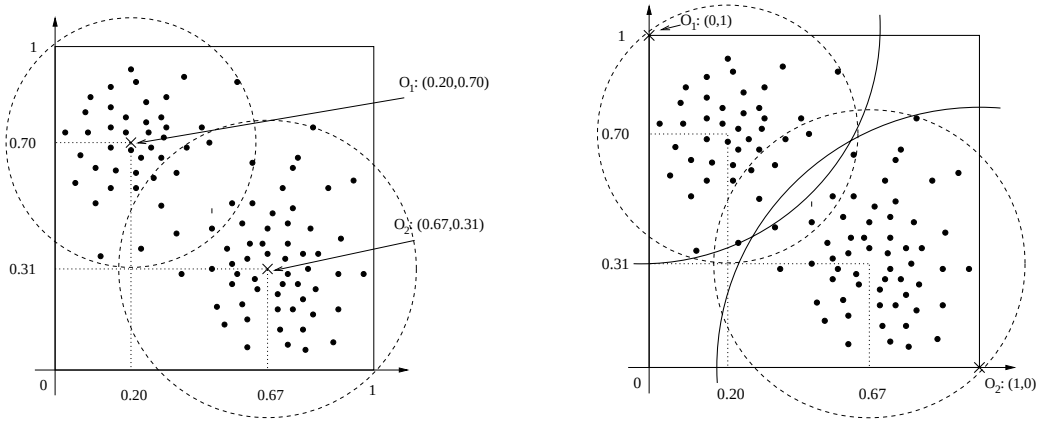
External point. Any point along the line formed by the centroid of a hyperplane and the centroid of the corresponding data space can also be used as a reference point.[2] By *external point*, we refer to a reference point that falls out of the data space. This heuristics is expected to perform well when the affected area is quite large, especially when the data are uniformly distributed. We note that both closest and farthest distance can also be supported. Figures 5(c) shows an example of closest distance for 2-dimensional space. Again, we observe that the affected space under external point is reduced (compared to using the centroid of the hyperplane). The generalization here conceptually corresponds to the manner in which the iMinMax tree generalizes the pyramid tree. The iDistance metric can perform approximately like the iMinMax tree, as we shall see in our experimental study.

5.2 Cluster Based Partitioning

As mentioned, equi-partitioning is expected to be effective only if the data are uniformly distributed. However, data points are often correlated. In these cases, a more appropriate partitioning strategy would be used to identify clusters from the data space. However, in high-dimensional data space, the distribution of data points is mostly sparse, and hence clustering is not as straightforward as in low-dimensional databases. There are several existing clustering schemes in the literature such as BIRCH [20] and CURE [10]. While our metric based indexing is not dependent on the underlying clustering method, we expect the clustering strategy to have an influence on retrieval performance. In our experiment, we adopt a sampling-based approach. The method comprises four steps. First, we obtain a sample of the database. Second, from the sample, we can obtain the statistics on the distribution of data in each dimension. Third, we select k_i values from dimension i. These k_i values are those values whose frequencies exceed a certain threshold value. (Histograms can be used to observe these values directly.) We can then form $\prod k_i$ centroids from these values. For example, in a 2-dimensional data set, we can pick 2 high frequency values, say 0.2 and 0.8, on one dimension, and 2 high frequency values, say 0.3 and 0.6, on another dimension. Based on this, we can predict the clusters could be around (0.2,0,3), (0.2,0.6), (0.8,0.3) or (0.8,0.6), which can be treated as the clusters' centroids. Fourth, we count the data that are nearest to

mapping function are different; in the Pyramid method, a d-dimensional data point is associated with a pyramid based on an attribute value, and is represented as a value away from the centroid of the space.

[2]We note that the other two reference points are actually special cases of this.



(a) Cluster centroid.

(b) Cluster edge.

Figure 6: Cluster-based space partitioning.

each of the centroids; if there are sufficient number of data around a centroid, then we can estimate that there is a cluster there.

We note that the third step of the algorithm is crucial since the number of clusters can have an impact on the search area and the number of traversals from the root to the leaf nodes. When the number of clusters is small, more points are likely to have similar distance to a given reference point. On the other hand, when the number of clusters is large, more data spaces, defined by spheres with respect to centroid of clusters, are likely to overlap, and incur additional traversal and searching. Our solution is simple: if the number of clusters is too many, we can merge those whose centroids are closest; similarly, if the number of clusters is too few, we can split a large cluster into two smaller ones. We expect the number of clusters to be a tuning parameter, and may vary for different applications and domains.

Once the clusters are obtained, we need to select the reference points. Again, we have several possible options when selecting reference points.

Centroid of cluster. The centroid of a cluster is a natural candidate as a reference point. Figure 6(a) shows an example with two clusters in 2-dimensional space.

Edge of cluster. As shown in Figure 6(a), when the centroid is used, the sphere area of both clusters have to be enlarged to include outlier points, leading to significant overlap in the data space. To minimize the overlap, we can select points on the edge of the hyperplanes as reference points. Figure 6(b) is an example of 2-dimensional data space. There are two clusters and the edge points are $O_1 : (0, 1)$ and $O_2 : (1, 0)$. As shown, the overlap of the two partitions is smaller than that using cluster centroids as reference points.

6 Performance Study

In this section, we report the results of an extensive performance study that we have conducted to evaluate the performance of iDistance. Variant indexing strategies of iDistance are tested on different data sets, varying data set dimension, data set size and data distribution.

We also extended the iMinMax(θ) scheme [13] to support KNN queries, and to return approximate answers progressively [19]. iMinMax(θ) maps a high-dimensional point to either the maximum or minimum value of the values among the various dimensions of the point, and a range query requires d subqueries.

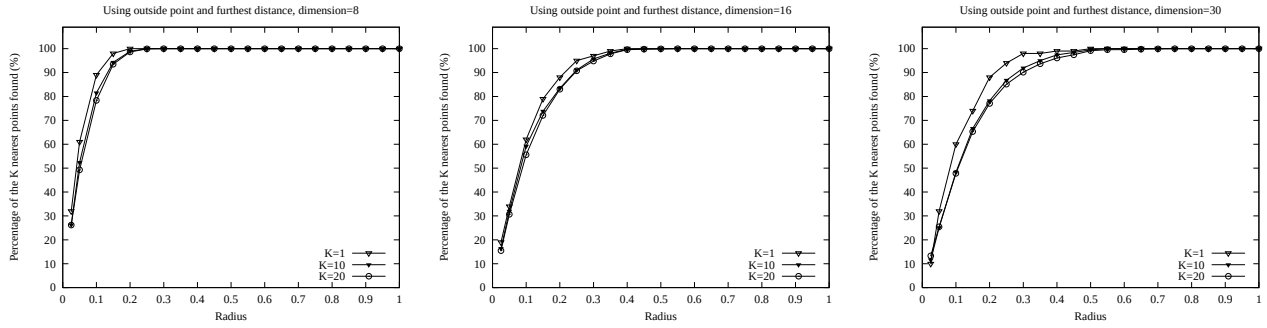
The comparative study was done between iDistance, iMinMax, and the A-tree [15]. We also compare iDistance against linear scan which has been shown to be effective for KNN queries in high-dimensional data space.

6.1 Experiment Setup

We implemented iMinMax(θ) and the iDistance technique and their search algorithms in C, and used the B^+-tree as the single dimensional index structure. We obtained the source codes of the A-tree from the authors [15]. For all indexes, we index page size to 4 KB. All the experiments are performed on a 300-MHz SUN Ultra 10 with 64 megabytes main memory, running SUN Solaris.

We conducted many experiments using various data sets, with some deriving from LUV color histograms of 20,000 images. Here, we report some of the more interesting results on KNN queries; more results can be found in [18]. For each query, a d-dimensional point is used. We issue five hundred points, and take the average I/O cost as the performance metric.

In the experiment, we generated 8, 16, 30-dimensional uniform data sets. The data size ranges from 100,000 to 500,000. For the clustered data sets, we used a clustering algorithm similar to BIRCH to



(a) Dimension = 8 (b) Dimension = 16 (c) Dimension = 30

Figure 7: Effect of search radius on query accuracy.

generate the data sets.

6.2 Effect of Search Radius on Accuracy

In high-dimensional KNN search, the search around the neighborhood is required to find K nearest neighbors. Typically, a small search sphere is used and enlarged when the search condition cannot be met. Hence, it is important to study the effect of search radius on the proposed index methods.

In this experiment, we used 8-dimensional, 16-dimensional and 30-dimensional, 100K tuple uniformly distributed data sets. We use only the (external points,farthest distance) combination in this experiment. Figures 7(a)-(c) show the average percentage of KNN answers returned over multiple query points with respect to the search radius (*querydist*), of the 8,16 and 30 dimensional datasets respectively. The results show that as radius increases, the accuracy improves and hits 100% at certain query distance. A query with smaller K requires less searching to retrieve the required answers. As the number of dimension increases, the radius required to obtain 100% also increases due to increase in possible distance between two points and sparsity of data space in higher-dimensional space. However, we should note that the seemingly large increase is not out of proportion with respect to the total possible dissimilarity. We also observe that iDistance can return a significant number of nearest neighbors with a small query radius.

In Figure 8, we see the retrieval efficiency of iDistance for 10-NN queries. First, we note that we have stopped at radius around 0.5. This is because the algorithm is able to detect all the nearest neighbors once the radius reaches that length. As shown, iDistance can provide fast initial answers quickly (compared to linear scan). Moreover, iDistance can produce the complete answers much faster than linear scan for reasonable number of dimensions (i.e., 8 and 16). When the number of dimensions reaches 30, iDistance takes a longer time to produce the complete answers. This is expected since the data are uniformly distributed. However, because of its ability to produce approximate

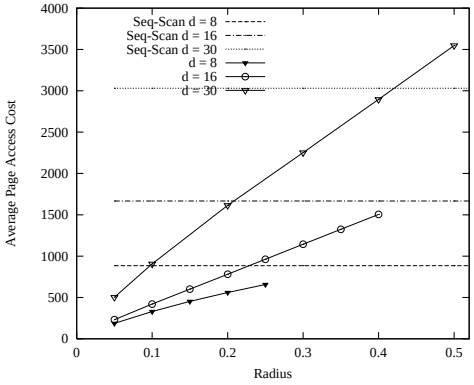


Figure 8: Retrieval Efficiency.

answers, iDistance is a promising strategy to adopt.

6.3 Effect of Reference Points on Equi-Partitioning Schemes

In this experiment, we evaluate the efficiency of equi-partitioning-based iDistance schemes using one of the previous data sets.

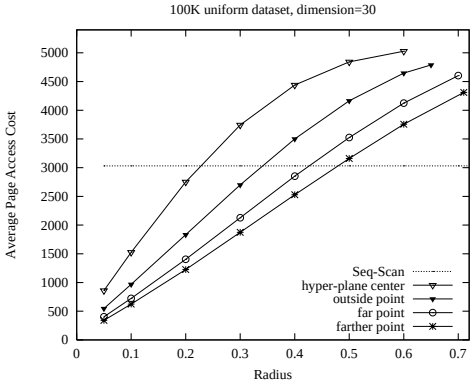
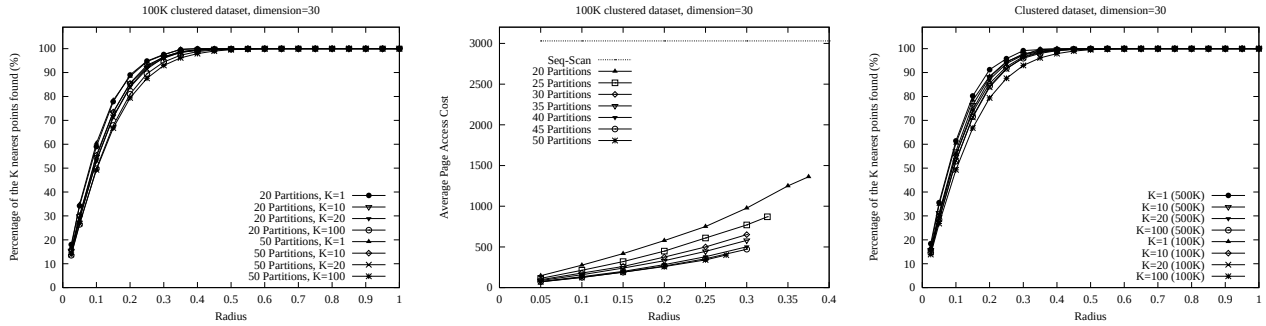


Figure 9: Effect of reference points.

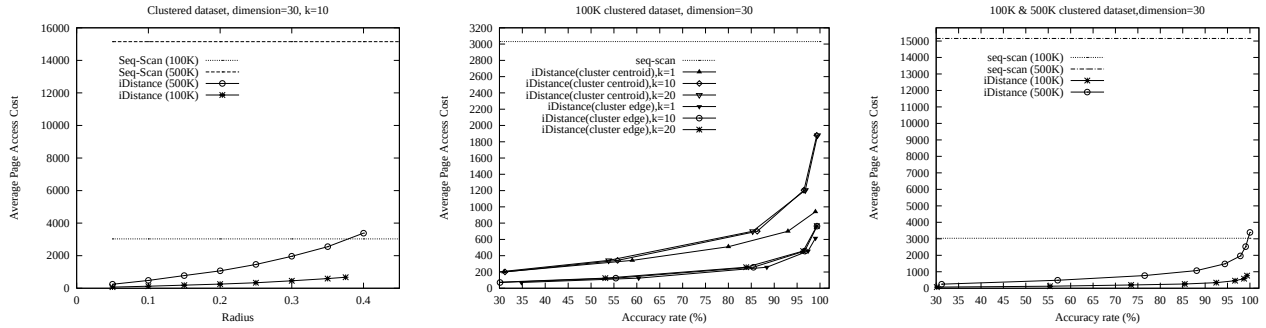
Figure 9 shows the results for (centroid,closest) combination and 3 (external points, closest) schemes. Each of the external points is farther away from the



(a) Percentage trend with variant searching radius.

(b) Effect of number of partitions on iDistance.

(c) Effect of data size on search radius.



(d) Effect of data size on I/O cost.

(e) Effect of reference points in clustered data sets.

(f) Effect of clustered data size.

Figure 10: On cluster-based schemes.

hyperplane centroid than the others. First, we note that the I/O cost increases with radius when doing KNN search. This is expected since a larger radius would mean increasing number of false hits and more data are examined. We also notice that iDistance-based schemes are very efficient in producing fast first answers, as well as the complete answers. Moreover, we note that the farther away the reference point from the hyperplane centroid, the better is the performance. This is because the data space that is required to be traversed is smaller in these cases as the point gets farther away.

6.4 Performance of Cluster-based Schemes

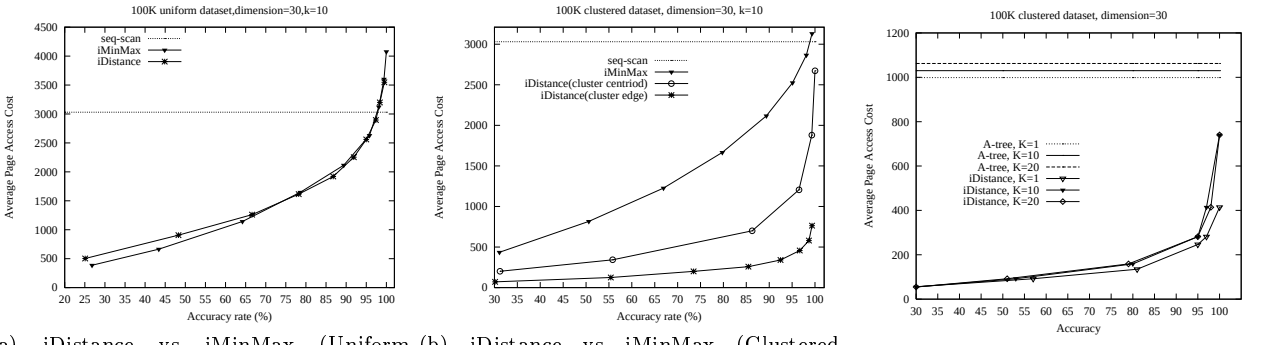
In this experiment, we tested a data set with 100K data points of 20 and 50 clusters, some of which overlapped each other. To test the effect of the number of partitions on KNN, we merge some number of close clusters and treat them as one cluster.

We use the edge near to the cluster as its reference point for the partition. We notice in Figure 10(a), as with the other experiments, that the complete answer set can be obtained with a reasonably small radius. We also notice that a smaller number of partitions performs better in returning the K points. This is probably due to the larger partitions for small number of partitions.

The I/O results in Figure 10(b) show a slightly different trend. Here, we note that a smaller number of partitions incurs higher I/O cost. This is reasonable since a smaller number of partitions would mean that each partition is larger, and the number of false drops being accessed is also higher. As before, we observe that the cluster-based scheme can obtain the complete set of answers in a short time.

We also repeated the experiments for a larger data set of 500K points of 50 clusters using the edge of cluster strategy in selecting the reference points. Figure 10(c) shows the searching radius required for locating K (K=1, 10, 20, 100) nearest neighbors when 50 partitions were used. The results show that searching radius does not increase (compared to small data set) in order to get good percentage of KNN. However, the data size does have great impact on the query cost. Figure 10(d) shows the I/O cost for 10-NN queries. We can see that iDistance has a speedup factor of 4 over linear scan when all 10 NNs were retrieved.

Figure 10(e) and Figure 10(f) show how the I/O cost is affected as the nearest neighbors are being returned. Here, a point (x, y) in the graph means that x percent of the K nearest points are obtained after y number of I/Os. Here, we note that all the proposed schemes can produce 100% answers at a much lower cost than linear scan. In fact, the improvement can be as much as five times. The results also show that picking an edge point to be the reference point is generally better because it can reduce the amount of overlap.



(a) iDistance vs iMinMax (Uniform dataset)

(b) iDistance vs iMinMax (Clustered dataset)

(c) iDistance vs A-tree (Clustered dataset)

Figure 11: A comparative study.

6.5 Comparative Study

In this study, we compare iDistance cluster-based schemes with iMinMax and A-tree.

Our first experiment uses a 100K 30-dimensional uniform data set. The query is a 10-NN query. For iDistance, we use the (external point, farthest) scheme. Figure 11(a) shows the result of the experiment. First, we note that both iMinMax and iDistance can produce quality approximate answers very quickly compared to linear scan. As shown, the I/O cost is lower than linear scan with up to 95% accuracy. However, because the data is uniformly distributed, to retrieve all the 10 NN takes a longer time than linear scan since all points are almost equidistant to one another. Second, we note that iMinMax and iDistance perform equally well.

In another set of experiments, we use a 100K 30-dimensional clustered data set. The query is still a 10-NN query. Here, we study two versions of cluster-based iDistance – one that uses the edge of the cluster as a reference point, and another that uses the centroid of the cluster. Figure 11(b) summarizes the result. First, we observe that among the two cluster-based schemes, the one that employs the edge reference points performs best. This is because of the smaller overlaps in space of this scheme. Second, as in earlier experiments, we see that the cluster-based scheme can return initial approximate answer quickly, and can eventually produce the final answer set much faster than the linear scan. Third, we note that iMinMax can also produce approximate answers quickly. However, its performance starts to degenerate as the radius increases, as it attempts to search for exact K NNs. Unlike iDistance which terminates once the K nearest points are determined, iMinMax cannot terminate until the entire data set is examined. As such, to obtain the final answer set, iMinMax performs poorly. Finally, we see that the relative performance between iMinMax and iDistance for clustered data set is different from that of uniform data set. Here, iDistance outperforms iMinMax by a wide margin because of the larger number of false drops produced by iMinMax.

We also compare iDistance cluster-based schemes with the recently proposed A-tree structure [15], which has been shown to be far more efficient than the SR-tree [12] and significantly more efficient than the VA-file [16]. Upon investigation, we note that *the gain is not only due to the use of virtual bounding boxes, but also the smaller sized logical pointers. Instead of storing actual pointers in the internal nodes, it has to store a memory resident mapping table. In particular, the size of the table is very substantial.* To have a fair comparison with the A-tree, we also implemented a version of the iDistance method in a similar fashion – the resultant effect is that the fan-out is greatly increased.

In this set of experiments, we use a 100K 30-dimensional clustered data set, and 1NN, 10NN and 20NN queries. The page size we used is 4K bytes. Figure 11(c) summarizes the result. The results show that iDistance is clearly more efficient than the A-tree for the three types of queries we used. Moreover, it is able to produce answers progressively which A-tree is unable to achieve. The result also shows that the difference in performance for different K values is not significant when K gets larger.

6.6 CPU Cost

While linear scan incurs less seek time, linear scan of a feature file entails examination of each data point and calculation of distance between each data point and the query point. This will result in high CPU cost for linear scan. Figure 12 shows the CPU time of linear scan and iDistance for the same experiment as in Figure 10(b). It is interesting to note that the performance in terms of CPU time approximately reflects the trend in page accesses. The results show that the best iDistance method achieves about a seven fold increase in speed. We omit iMinMax in our comparison as iMinMax has to search the whole index in order to ensure 100% accuracy, and its CPU time at that point is much higher than linear scan.

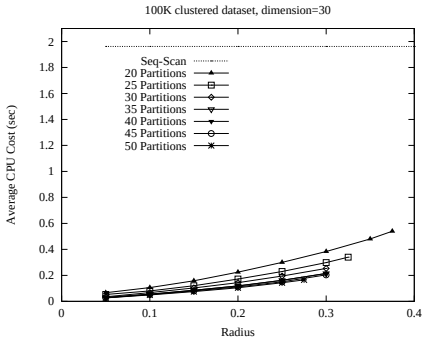


Figure 12: CPU Time.

7 Conclusion

In this paper, we have presented a simple and efficient approach to nearest neighbor processing, called iDistance. Extensive experiments were conducted and the results show that iDistance is both effective and efficient. In fact, iDistance achieves a speedup factor of seven over linear scan without blowing up in storage requirement and compromising on the accuracy of the results (unless it is done with the intention for even faster response time). Moreover, it is well suited for integration into existing DBMSs. iDistance also outperforms the A-tree and iMinMax schemes, and is robust and adaptive to different data distributions. As for the extension of current work, we are implementing a similarity join algorithm using the iDistance.

Acknowledgement

We would like to thank the authors of [15] for providing us with the source code of the A-tree structure that we used in our comparative study. The work of H.V.Jagadish was supported in part by NSF under grant number IIS-0002356.

References

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

[2] S. Berchtold, C. Bohm, H.P. Kriegel, J. Sander, and H.V. Jagadish. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proc. 16th International Conference on Data Engineering*, pages 577–588. 2000.

[3] S. Berchtold, D.A. Keim, and H.P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 28–37. 1996.

[4] E. Bertino and et. al. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic, 1997.

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbors meaningful? In *Proc. International Conference on Database Theory*, 1999.

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

[7] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: a new approach to indexing high dimensional spaces. In *Proc. 26th International Conference on Very Large Databases*, pages 89–100, 2000.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-trees: An efficient access method for similarity search in metric space. In *Proc. 23rd International Conference on Very Large Data Bases*, pages 426–435. 1997.

[9] J. Goldstein and R. Ramakrishnan. Contrast plots and p-sphere trees: space vs. time in nearest neighbor searches. In *Proc. 26th International Conference on Very Large Databases*, pages 429–440, 2000.

[10] S. Guha, R. Rastogi, and K. Shim. Cure: an efficient clustering algorithm for large databases. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*. 1998.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57. 1984.

[12] N. Katamaya and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. 1997 ACM SIGMOD International Conference on Management of Data*. 1997.

[13] B. C. Ooi, K. L. Tan, C. Yu, and S. Bressan. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 166–174. 2000.

[14] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[15] Y. Sakurai, M. Yoshikawa, and S. Uemura. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. 26th International Conference on Very Large Data Bases*, pages 516–526. 2000.

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

[17] D.A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th International Conference on Data Engineering*, pages 516–523. 1996.

[18] C. Yu. *High-Dimensional Indexing*. PhD thesis, Department of Computer Science, National University of Singapore, 2001.

[19] C. Yu, B. C. Ooi, and K. L. Tan. Progressive KNN search using B^+-trees. *http://www.comp.nus.edu.sg/∼ooibc/appimm.ps*, page 20p, 2001.

[20] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: an efficient data clustering method for very large databases. In *Proc. 1996 ACM SIGMOD International Conference on Management of Data*. 1996.