iDistance: An Adaptive B^+-Tree Based Indexing Method for Nearest Neighbor Search

H. V. JAGADISH
University of Michigan
BENG CHIN OOI and KIAN-LEE TAN
National University of Singapore
CUI YU
Monmouth University
and
RUI ZHANG
National University of Singapore

In this article, we present an efficient B^+-tree based indexing method, called iDistance, for K-nearest neighbor (KNN) search in a high-dimensional metric space. iDistance partitions the data based on a space- or data-partitioning strategy, and selects a reference point for each partition. The data points in each partition are transformed into a single dimensional value based on their similarity with respect to the reference point. This allows the points to be indexed using a B^+-tree structure and KNN search to be performed using one-dimensional range search. The choice of partition and reference points adapts the index structure to the data distribution.

We conducted extensive experiments to evaluate the iDistance technique, and report results demonstrating its effectiveness. We also present a cost model for iDistance KNN search, which can be exploited in query optimization.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Indexing, KNN, nearest neighbor queries

Authors' addresses: H. V. Jagadish, Department of Computer Science, University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109; email: jag@eecs.umich.edu; B. C. Ooi, K.-L. Tan, and R. Zhang, Department of Computer Science, National University of Singapore Kent Ridge, Singapore 117543; email: {ooibc,tankl,zhangru1}@comp.nus.edu.sg; C. Yu, Department of Computer Science, Monmouth University, 400 Cedar Avenue, West Long Branch, NJ 07764-1898; email: cyu@monmouth.edu.

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. Existing multidimensional indexes such as R-trees [Guttman 1984] have been shown to be inefficient even for supporting range queries in high-dimensional databases; however, they form the basis for indexes designed for high-dimensional databases [Katamaya and Satoh 1997; White and Jain 1996]. To reduce the effect of high dimensionality, use of larger fanouts [Berchtold et al. 1996; Sakurai et al. 2000], dimensionality reduction techniques [Chakrabarti and Mehrotra 2000, 1999], and filter-and-refine methods [Berchtold et al. 1998b; Weber et al. 1998] have been proposed. Indexes have also been specifically designed to facilitate metric based query processing [Bozkaya and Ozsoyoglu 1997; Ciaccia et al. 1997; Traina et al. 2000; Filho et al. 2001]. However, linear scan remains an efficient search strategy for similarity search [Beyer et al. 1999]. This is because there is a tendency for data points to be nearly equidistant to query points in a high-dimensional space. While linear scan is effective in terms of sequential read, every point incurs expensive distance computation, when used for the nearest neighbor problem. For quick response to queries, with some tolerance for errors (i.e., answers may not necessarily be the nearest neighbors), *approximate* nearest neighbor (NN) search indexes such as the P-Sphere tree [Goldstein and Ramakrishnan 2000] have been proposed. The P-Sphere tree works well on static databases and provides answers with assigned accuracy. It achieves its efficiency by duplicating data points in data clusters based on a sample query set. Generally, most of these structures are not *adaptive* to data distributions. Consequently, they tend to perform well for some datasets and poorly for others.

In this article, we present iDistance, a new technique for KNN search that can be adapted to different data distributions. In our technique, we first partition the data and define a reference point for each partition. Then we 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, a classical B^+-tree can be used to index this distance. As such, it is easy to graft our technique on top of an existing commercial relational database. This is important as most commercial DBMSs today do not support indexes beyond the B^+-tree and the R-tree (or one of its variants). The effectiveness of iDistance depends on how the data are partitioned, and how reference points are selected.

For a KNN query centered at q, a range query with radius r is issued. The iDistance KNN search algorithm searches the index from the query point outwards, and for each partition that intersects with the query sphere, a range query is resulted. If the algorithm finds K elements that are closer than r from

q at the end of the search, the algorithm terminates. Otherwise, it extends the search radius by Δr, and the search continues to examine the unexplored region in the partitions that intersects with the query sphere. The process is repeated till the stopping condition is satisfied. To facilitate efficient KNN search, we propose partitioning and reference point selection strategies as well as a cost model to estimate the page access cost of iDistance KNN searching.

This article is an extended version of our earlier paper [Yu et al. 2001]. There, we present the basic iDistance method. Here, we have extended it substantially to include a more detailed discussion of the technique and algorithms, a cost model, and comprehensive experimental studies. In this article, we conducted a whole new set of experiments using different indexes for comparison. In particular, we compare iDistance against sequential scan, the M-tree [Ciaccia et al. 1997], the Omni-sequential [Filho et al. 2001] and the bd-tree structure [Arya et al. 1994] on both synthetic and real datasets. While the M-tree and the Omni-sequential schemes are disk-based structures, the bd-tree is a main memory based index. Our results showed that iDistance is superior to these techniques for a wide range of experimental setups.

The rest of this article is organized as follows. In the next section, we present the background for metric-based KNN processing, and review some related work. In Section 3, we present the iDistance indexing method and KNN search algorithm, and in Section 4, its space- and data-based partitioning strategies. In Section 5, we present the cost model for estimating the page access cost of iDistance KNN search. We present the performance studies in Section 6, and finally, we conclude in Section 7.

2. BACKGROUND AND RELATED WORK

In this section, we provide the background for metric-based KNN processing, and review related work.

2.1 KNN Query Processing

In our discussion, we assume that DB is a set of points in a d-dimensional data space. A *K-nearest neighbor query* finds the K objects in the database closest in distance to a given query object. More formally, the KNN problem can be defined as follows:

Given a set of points DB in a d-dimensional space DS, and a query point $q \in DS$, find a set S that contains K points in DB such that, for any $p \in S$ and for any $p' \in DB - S$, $dist(q, p) < dist(q, p')$.

Table I describes the notation used in this article.

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 can be employed (see Figure 1). The search starts with a query sphere about q, with a small initial radius, which can be set according to historical records. We maintain a candidate answer set that contains points that could be the K nearest neighbors of q. Then the query sphere is enlarged step by step and the candidate answer set

Table I. Notation

Notation	Meaning
C_{eff}	Average number of points stored in a page
d	Dimensionality of the data space
DB	The dataset
DS	The data space
m	Number of reference points
K	Number of nearest neighbor points required by the query
p	A data point
q	A query point
S	The set containing K NNs
r	Radius of a sphere
$dist\_max_i$	Maximum radius of partition P_i
O_i	The ith reference point
P_i	The ith partition
$dist(p_1, p_2)$	Metric function returns the distance between points p_1 and p_2
$querydist(q)$	Query radius of q
$sphere(q, r)$	Sphere of radius r and center q
$furthest(S, q)$	Function returns the object in S furthest in distance from q

KNN Basic Search Algorithm

```
1.      start with a small search sphere centered at query point
2.      search and check all partitions intersecting the current query space
3.      if K nearest neighbors are found
4.          exit;
5.      else
6.          enlarge search sphere;
7.          goto 2;
end KNN;
```

Fig. 1. Basic KNN algorithm.

is updated accordingly until we can make sure that the K candidate answers are the true K nearest neighbors of q.

2.2 Related Work

Many multi-dimensional structures have been proposed in the literature, including various KNN algorithms [Böhm et al. 2001]. Here, we briefly describe a few relevant methods.

In Weber et al. [1998], the authors describe a simple vector approximation scheme, called VA-file. The VA-file divides the data space into 2^b rectangular cells where b denotes a user specified number of bits. 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 compact, geometric approximations. Nearest neighbor searches are performed by scanning the entire approximation file, and by excluding the vast majority of vectors from the search (filtering step) based only on these approximations. After the filtering step, a small set of candidates remains. These candidates are then visited and the actual distances to the query point q are determined. VA-file reduces the number of disk accesses, but it incurs higher computational cost to decode the bit-string, compute all the lower and some upper bounds on the distance to the query point, and determine the actual distances of candidate points. Another problem with the VA-file is that it works well for uniform data, but for skewed data, the pruning effect of the approximation vectors becomes very bad. The IQ-tree [Berchtold et al. 2000] extends the notion of the VA-file to use a tree structure where appropriate, and the bit-encoded file structure where appropriate. It inherits many of the benefits and drawbacks of the VA-file discussed above and the M-tree discussed next.

In Ciaccia et al. [1997], the authors proposed the height-balanced M-tree to organize and search large datasets from a generic metric space, where object proximity is only defined by a distance function satisfying the positivity, symmetry, and triangle inequality postulates. In an M-tree, leaf nodes store all indexed (database) objects, represented by their keys or features, 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 the covering tree of O_r, are within the distance r(O_r) from O_r, r(O_r) > 0, which is called the covering radius of O_r. Finally, a routing object O_r, is associated with a distance to P(O_r), its parent object, that is, the routing object that references the node where the O_r entry is stored. Obviously, 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, but no covering radius is needed. The strength of M-tree lies in maintaining the pre-computed distance in the index structure. However, the node utilization of the M-tree tends to be low due to its splitting strategy.

Omni-concept was proposed in Filho et al. [2001]. The scheme chooses a number of objects from a database as global 'foci' and gauges all other objects based on their distances to each focus. If there are l foci, each object will have l distances to all the foci. These distances are the Omni-coordinates of the object. The Omni-concept is applied in the case where the correlation behaviors of the database are known beforehand and the intrinsic dimensionality (d_2) is smaller than the embedded dimensionality d of the database. A good number of foci is $\lceil d_2 \rceil + 1$ or $\lceil d_2 \rceil \times 2 + 1$, and they can either be selected or efficiently generated. Omni-trees can be built on top of different indexes such as the B$^+$-tree and the R-tree. Omni B-trees used l B$^+$-trees to index the Omni-coordinates of the objects. When a similarity range query is conducted, on each B$^+$-tree, a set of candidate objects is obtained and intersection of all the l candidate sets will be checked for the final answer. For the KNN query, the query radius is estimated by some selectivity estimation formulas. The Omni-concept improves the performance of similarity search by reducing the number of distance calculations during search operation. However, multiple sets of ordinates for each point increases the page access cost, and searching multiple B-trees (or R-trees) also increases CPU time. Finally, the intersection of the l candidate sets incurs additional cost. In iDistance, only one set of ordinates is used and also only one B$^+$-tree is used to index them, therefore iDistance has less page accesses while still reducing the distance computation. Besides, the choice of reference points in iDistance is quite different from the choice of foci bases in Omni-family techniques.

The P-Sphere tree [Goldstein and Ramakrishnan 2000] is a two level structure, the root level and leaf level. The root level contains a series of <*sphere descriptor, leaf page pointer*> pairs, while each leaf of the index corresponds to a sphere (we call it the *leaf sphere* in the following) and contains all data points that lie within the sphere described in the corresponding sphere descriptor. The leaf sphere centers are chosen by sampling the dataset. The NN search algorithm only searches the leaf with the sphere center closest to the query point q. It searches the NN (we denote it as p) of q among the points in this leaf. When finding p, if the query sphere is totally contained in the leaf sphere, then we can confirm that p is the nearest neighbor of q; otherwise, a second best strategy is used (such as sequential scan). A data point can be within multiple leaf spheres, so the points are stored multiple times in the P-Sphere tree. This is how it trades space for time. A variant of the P-Sphere tree is the nondeterministic (ND) P-Sphere tree, which returns answers with some probability of being correct. The ND P-Sphere tree NN search algorithm searches k leaf spheres whose centers are closest to the query point, where k is a given constant (note that this k is different from the K in KNN). A problem arises in high-dimensional space for the deterministic P-Sphere tree search, because the nearest neighbor distance tends to be very large. It is hard for the nearest leaf sphere of q to contain the whole query sphere when finding the NN of q within this sphere. If the leaf sphere contains the whole query sphere, the radius of the leaf sphere must be very large, typically close to the side length of the data space. In this case, where the major portion of the whole dataset is within this leaf, scanning a leaf is not much different from scanning the whole dataset. Therefore, the authors also hinted that using deterministic P-Sphere trees for medium to high dimensionality is impractical. In Goldstein and Ramakrishnan [2000], only the experimental results of ND P-Sphere are reported, which is shown to be better than sequential scan at the cost of space. Again, iDistance only uses one set of ordinates and hence has no duplicates. iDistance is meant for high dimensional KNN search; which P-Sphere tree cannot address efficiently. The ND P-Sphere tree has better performance in high-dimensional space, but our technique, iDistance is looking for exact nearest neighbors.

Another metric based index is the Slim-tree [Traina et al. 2000], which is a height balanced and dynamic tree structure that grows from the leaves to the root. The structure is fairly similar to that of the M-tree, and the objective of the design is to reduce the overlap between the covering regions in each level of the metric tree. The split algorithm of the Slim-tree is based on the concept of minimal spanning tree [Kruskal 1956], and it distributes the objects by cutting the longest line among all the closest connecting lines between objects. If none exits, an uneven split is accepted as a compromise. The slim-down algorithm is a post-processing step applied on an existing Slim-tree to reduce the overlaps between the regions in the tree.

Due to the difficulty of processing exact KNN queries, some studies, such as Arya et al. [1994, 1998] turn to approximate KNN search. In these studies, a relative error bound ϵ is specified so that the approximate KNN distance is at most $(1+\epsilon)$ times the actual KNN distance. We can specify ϵ to be 0 so that exact answers are returned. However, the algorithms in Arya et al. [1994, 1998] are based on a main memory indexing structure called bd-tree, while the problem we are considering is when the data and indexes are stored on secondary memory. Main memory indexing requires a slightly different treatment since optimization on the use of L2 cache is important for speed-up. Cui et al. [2003, 2004] show that existing indexes have to be fine-tuned for exploiting L2 cache efficiently. Approximate KNN search has recently been studied in the data stream model [Koudas et al. 2004], where the memory is constrained and each data item could be read only once.

While more indexes have been proposed for high-dimensional databases, other performance speedup methods such as dimensionality reduction have also been performed. The idea of dimensionality reduction is to pick the most important features to represent the data, and an index is built on the reduced space [Chakrabarti and Mehrotra 2000; Faloutsos and Lin 1995; Lin et al. 1995; Jolliffe 1986; Pagel et al. 2000]. To answer a query, it is mapped to the reduced space and the index is searched based on the dimensions indexed. The answer set returned contains all the answers and some false positives. In general, dimensionality reduction can be performed on the datasets before they are indexed as a means to reduce the effect of the dimensionality curse on the index structure. Dimensionality reduction is lossy in nature; hence the query accuracy is affected as a result. How much information is lost, depends on the specific technique used and on the specific dataset at hand. For instance, Principal Component Analysis (PCA) [Jolliffe 1986] is a widely used method for transforming points in the original (high-dimensional) space into another (usually lower dimensional) space. 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. When the dataset is globally correlated, principal component analysis is an effective method for reducing the number of dimensions with little or no loss of information. However, in practice, the data points tend not to be globally correlated, and the use of global dimensionality reduction may cause a significant loss of information. As an attempt to reduce such loss of information, and also to reduce query processing due to false positives, a local dimensionality reduction (LDR) technique was proposed in Chakrabarti and Mehrotra [2000]. It exploits local correlations in data points for the purpose of indexing.

3. THE IDISTANCE

In this section, we describe 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 or representative 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.

Moreover, false drops can be efficiently filtered without incurring expensive distance computation.

3.1 An Overview

Consider a set of data points *DB* in a unit d-dimensional metric space *DS*, which is a set of points with an associated distance function *dist*. Let $p_1 : (x_0, x_1, \ldots, x_{d-1})$, $p_2 : (y_0, y_1, \ldots, y_{d-1})$ and $p_3 : (z_0, z_1, \ldots, z_{d-1})$ be three data points in *DS*. The distance function *dist* has the following properties:

$$dist(p_1, p_2) = dist(p_2, p_1) \quad \forall p_1, p_2 \in DB \tag{1}$$

$$dist(p_1, p_1) = 0 \quad \forall p_1 \in DB \tag{2}$$

$$0 < dist(p_1, p_2) \quad \forall p_1, p_2 \in DB; p_1 \neq p_2 \tag{3}$$

$$dist(p_1, p_3) \leq dist(p_1, p_2) + dist(p_2, p_3) \quad \forall p_1, p_2, p_3 \in DB \tag{4}$$

The last formula defines the triangular inequality, and provides a condition for selecting candidates based on metric relationship. Without loss of generality, we use the Euclidean distance as the distance function in our article, although other distance functions also apply for iDistance. For Euclidean distance, the distance between p_1 and p_2 is defined as

$$dist(p_1, p_2) = \sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2 + \cdots + (x_{d-1} - y_{d-1})^2}.$$

As in other databases, a high-dimensional database can be split into partitions. Suppose a point, denoted as O_i, is picked as the reference point for a data partition P_i. As we shall see shortly, O_i need not be a data point. A data point, p, in the partition can be referenced via O_i in terms of its distance (or proximity) to it, $dist(O_i, p)$. Using the triangle inequality, it is straightforward to see that

$$dist(O_i, q) - dist(p, q) \leq dist(O_i, p) \leq dist(O_i, q) + dist(p, q).$$

When we are working with a search radius of $querydist(q)$, we are interested in finding all points p such that $dist(p, q) \leq querydist(q)$. For every such point p, by adding this inequality to the above one, we must have:

$$dist(O_i, q) - querydist(q) \leq dist(O_i, p) \leq dist(O_i, q) + querydist(q).$$

In other words, in partition P_i, we need only examine candidate points p whose distance from the reference point, $dist(O_i, p)$, is bounded by this inequality, which in general specifies an annulus around the reference point.

Let $dist\_max_i$ be the distance between O_i and the point furthest from it in partition P_i. That is, let P_i have a radius of $dist\_max_i$. If $dist(O_i, q) - querydist(q) \leq dist\_max_i$, then P_i has to be searched for NN points, else we can eliminate this partition from consideration altogether. The range to be searched within an affected partition in the single dimensional space is $[dist(0_i, q) - querydist(q), min(dist\_max_i, dist(O_i, q) + querydist(q))]$. Figure 2 shows an example where the partitions are formed based on data clusters (the data partitioning strategy will be discussed in detail in Section 4.2). Here, for query point q and query radius r, partitions P_1 and P_2 need to be searched, while partition P_3 need not.

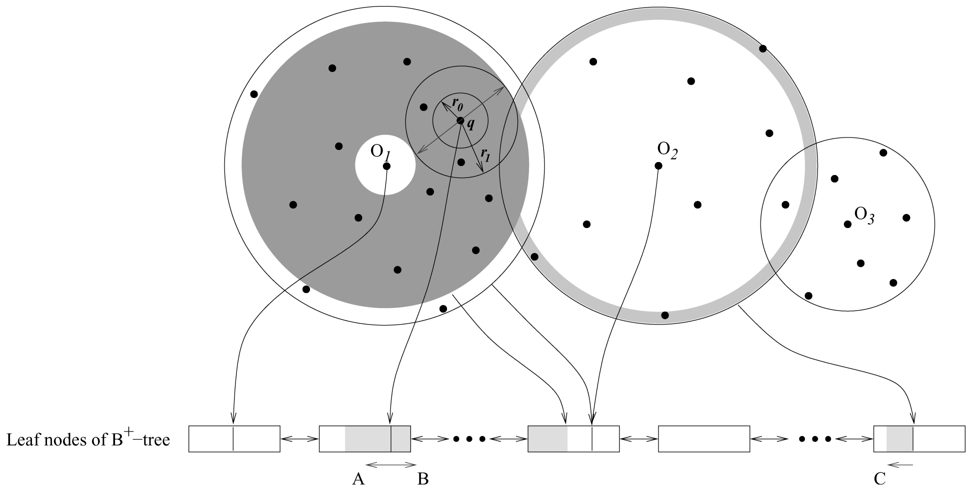


Fig. 2. Search regions for NN query q.

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.

To facilitate efficient metric-based KNN search, we have identified two important issues that have to be addressed:

(1) What index structure can be used to support metric-based similarity search?
(2) How should the data space be partitioned, and which point should be picked as the reference point for a partition?

We focus on the first issue here, and will turn to the second issue in the next section. In other words, for this section, we assume that the data space has been partitioned, and the reference point in each partition has been determined.

3.2 The Data Structure

In iDistance, high-dimensional points are transformed into points in 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. Suppose that we have m partitions, $P_0, P_1, \dots, P_{m-1}$ and their corresponding reference points, $O_0, O_1, \dots, O_{m-1}$.

Finally, in the third step, all data points are represented in a single dimensional space as follows. A data point $p : (x_0, x_1, \dots, x_{d-1}), 0 \leq x_j \leq 1, 0 \leq j < d$, has an index key, y, based on the distance from the nearest reference point O_i as follows:

$$y = i \times c + dist(p, O_i) \tag{5}$$

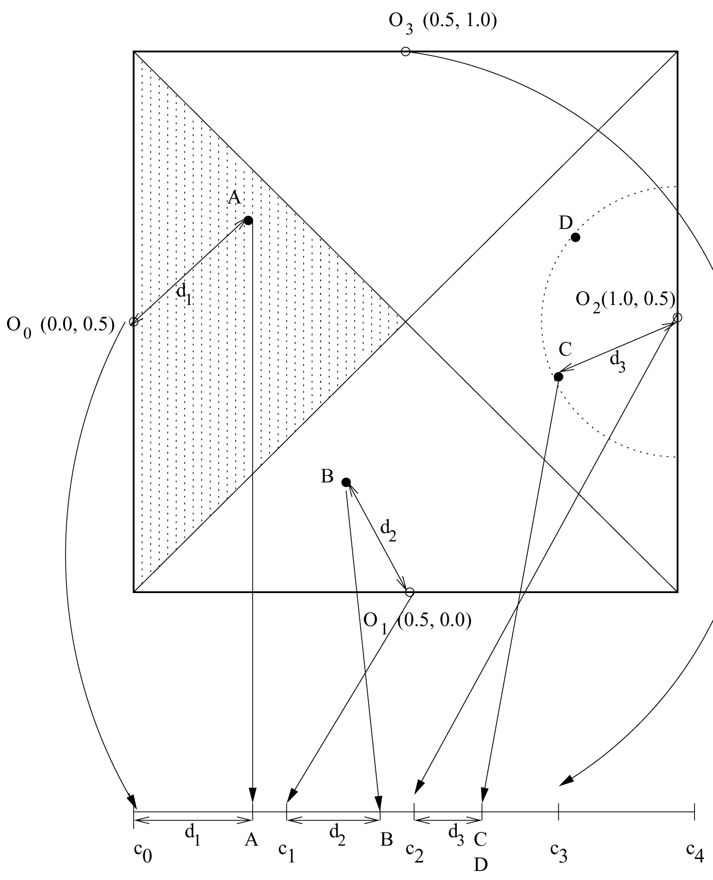


Fig. 3. Mapping of data points.

where c is a constant used 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 $[i \times c, (i+1) \times c)$. c must be set sufficiently large in order to avoid the overlap between the index key ranges of different partitions. Typically, it should be larger than the length of diagonal in the hypercube data space.

Figure 3 shows a mapping in a 2-dimensional space. Here, O_0, O_1, O_2 and O_3 are the reference points; points A, B, C and D are data points in partitions associated with the reference points; and, c_0, c_1, c_2, c_3 and c_4 are range partitioning values that represent the reference points as well. For example O_0 is associated with c_0, and all data points falling in its partition (the shaded region) have their distances relative to c_0. 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. That is, different points within a partition that are equidistant from the reference point have the same transformed value. For example, data points C and D have the same mapping value, and as a result, false positives may exist during search.

```
KNN Search Algorithm iDistanceKNN($q$, $\Delta r$, $max\_r$)

1.    $r = 0$;
2.    Stopflag = FALSE;
3.    initialize $lp[\ ]$, $rp[\ ]$, $oflag[\ ]$;
4.    while Stopflag == FALSE
5.        $r = r + \Delta r$;
6.        SearchO($q$, $r$);
end iDistanceKNN;
```

Fig. 4. iDistance KNN main search algorithm.

```
SearchO($q$, $r$)

1.    $p_{furthest}$ = furthest(S,q)
2.    if dist($p_{furthest}$, $q$) $< r$ and $|S| == K$
3.        Stopflag = TRUE;
4.            /* need to continue searching for correctness sake before stop*/
5.    for $i = 0$ to $m - 1$
6.        $dis$ = dist($O_i$, $q$);
7.        if not $oflag[i]$ /* if $O_i$ has not been searched before */
8.            if $sphere(O_i, dist\_max_i)$ contains $q$
9.                $oflag[i]$ = TRUE;
11.               $lnode$ = LocateLeaf($btree$, $i * c + dis$);
12.               $lp[i]$ = SearchInward($lnode$, $i * c + dis - r$);
13.               $rp[i]$ = SearchOutward($lnode$, $i * c + dis + r$);
14.           else if $sphere(O_i, dist\_max_i)$ intersects $sphere(q, r)$
15.               $oflag[i]$ = TRUE;
16.               $lnode$ = LocateLeaf($btree$, $dist\_max_i$);
17.               $lp[i]$ = SearchInward($lnode$, $i * c + dis - r$);
18.       else
19.           if $lp[i]$ not nil
20.               $lp[i]$ = SearchInward($lp[i] \rightarrow leftnode$, $i * c + dis - r$);
21.           if $rp[i]$ not nil
22.               $rp[i]$ = SearchOutward($rp[i] \rightarrow rightnode$, $i * c + dis + r$);
end SearchO;
```

Fig. 5. iDistance KNN search algorithm: SearchO.

In iDistance, we employ two data structures:

—A B$^+$-tree is used to index the transformed points to facilitate speedy retrieval. We choose the B$^+$-tree because it is an efficient indexing structure for one-dimensional data and it is also available in most commercial DBMSs. In our implementation of the B$^+$-tree, leaf nodes are linked to both the left and right siblings [Ramakrishnan and Gehrke 2000]. This is to facilitate searching the neighboring nodes when the search region is gradually enlarged.

—An array is used to store the m data space partitions and their respective reference points. The array is used to determine the data partitions that need to be searched during query processing.

3.3 KNN Search in iDistance

Figures 4–6 summarize the algorithm for KNN search with the iDistance method. The essence of the algorithm is similar to the generalized search

```
SearchInward(node, ivalue)

1.  for each entry $e$ in $node$ ($e = e_j, j = 1, 2, \ldots, Number\_of\_entries$)
2.      if $|S| == K$
3.          $p_{furthest}$ = furthest(S,q);
4.          if dist($e$, $q$) < dist($p_{furthest}$, $q$)
5.              $S = S - p_{furthest}$;
6.              $S = S \cup e$;
7.      else
8.          $S = S \cup e$;
9.  if $e_1$.key > $ivalue$
10.     $node$ = SearchInward($node \rightarrow leftnode$, $i * c + dis - r$);
11. if end of partition is reached
12.     $node$ = nil;
13. return($node$);
end SearchInward;
```

Fig. 6. iDistance KNN search algorithm: SearchInward.

strategy outlined in Figure 1. 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 furthest object in S (answer set) from the query point q is less than or equal to the current search radius r.

Before we explain the main concept of the algorithm *iDistanceKNN*, let us discuss three important routines. Note that routines SearchInward and SearchOutward are similar to each other, so we shall only explain routine SearchInward. Given a leaf node, routine SearchInward examines the entries of the node towards the left 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 SearchOutward) 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 SearchOutward) may also be close to q. As such, the left (or right for SearchOutward) sibling is examined. In other words, SearchInward (SearchOutward) searches the space towards (away from) the reference point of the partition. Let us consider again the example shown in Figure 2. For query point q, the SearchInward search on the partition P_1 will search towards left sibling as shown by the direction of arrow A, while the SearchOutward will search towards right sibling as shown by the direction of arrow B. For partition P_2, we only search towards left sibling by SearchInward as shown by the direction of arrow C. The routine LocateLeaf is a typical B$^+$-tree traversal algorithm, which locates a leaf node given a search value, hence the detailed description of the algorithm is omitted. It locates the leaf node either based on the respective value of q or maximum radius of the partition being searched.

We now explain the search algorithm. Searching in iDistance begins by scanning the auxiliary structure to identify the reference points, O_i, whose data spaces intersect the query region. For a partition that needs to be searched, the starting search point must be located. If q is contained inside the data sphere, the iDistance value of q (obtained based on Equation 5) is used directly, else $dist\_max_i$ is used. The search starts with a small radius. In our implementation, we just use Δr as the initial search radius. Then the search radius is increased by Δr, step by step, to form a larger query sphere. For each enlargement, there are three cases to consider.

(1) The partition contains the query point, q. In this case, we want to traverse the partition sufficiently to determine the K nearest neighbors. This can be done by first locating the leaf node whereby q may be stored (Recall that this node does not necessarily contain points whose distance is closest to q compared to its sibling nodes), and searching inward or outward of the reference point accordingly. For the example shown in Figure 2, only P_1 is examined in the first iteration and q is used to traverse down the B^+-tree.
(2) The query point is outside the partition but the query sphere intersects the partition. In this case, we only need to search inward. Partition P_2 (with reference point O_2) in Figure 2 is searched inward when the search sphere enlarged by Δr intersects P_2.
(3) The partition does not intersect the query sphere. Then, we do not need to examine this partition. An example in point is P_3 of Figure 2.

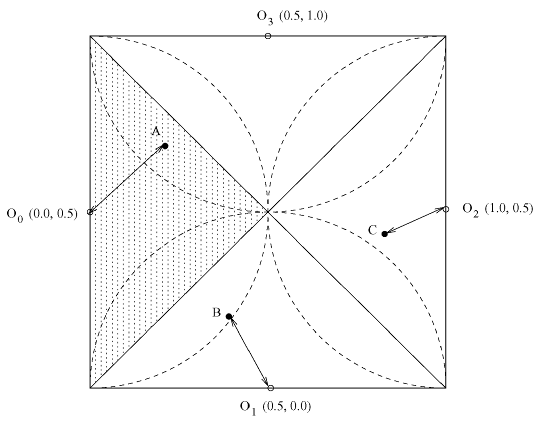
The search stops when the K nearest neighbors have been identified from the data partitions that intersect with the current query sphere and when further enlargement of the query sphere does not change the K nearest list. In other words, all points outside the partitions intersecting with the query sphere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

4. SELECTION OF REFERENCE POINTS AND DATA SPACE PARTITIONING

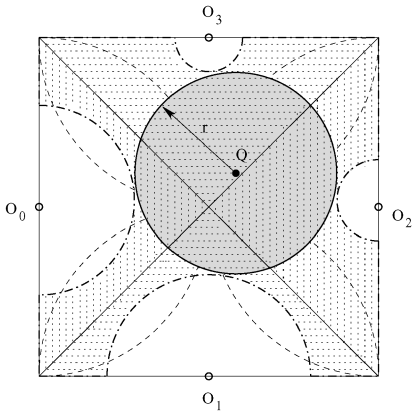
To support distance-based similarity search, we need to split the data space into partitions and for each partition, we need a reference point. In this section we look at some choices. For ease of exposition, we use 2-dimensional diagrams for illustration. However, we note that the complexity of indexing problems in a high-dimensional space is much higher; for instance, the distance between points larger than one (the full normalized range in a single dimension) could still be considered close since points are relatively sparse.

4.1 Space-Based Partitioning

A straightforward approach to data space partitioning is to subdivide the space into equal partitions. In a d-dimensional space, we have $2d$ hyperplanes. The method we adopted is to partition the space into $2d$ pyramids with the center of the unit cube space as their top, and each hyperplane forming the base of



(a) Space partitioning



(b) Effective search space

Fig. 7. Using (centers of hyperplanes, closest distance) as reference point.

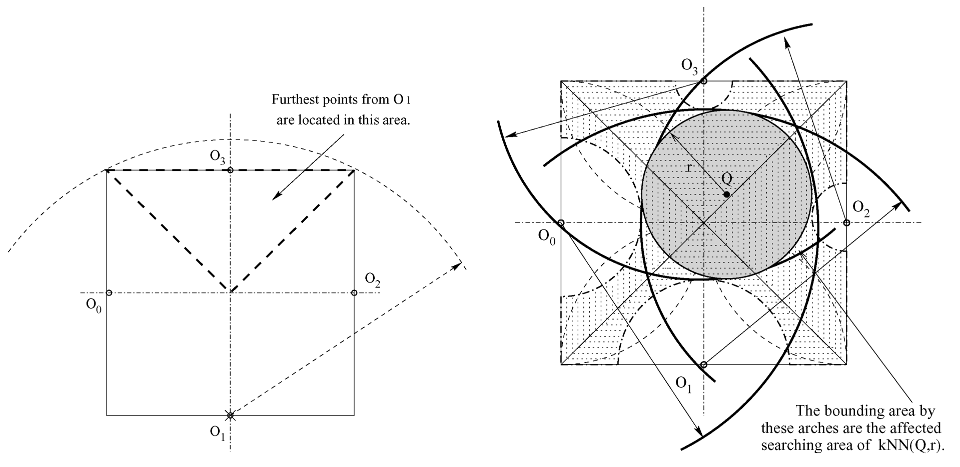
each pyramid.[1] We study the following possible reference point selection and partition strategies.

(1) **Center of Hyperplane, Closest Distance.** The center 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 7(a) shows an example in a 2-dimensional space. Here, O_0, O_1, O_2 and O_3 are the reference points, and point A is closest to O_0 and so belongs to the partition associated with it (the shaded region). Moreover, as shown, the actual data space is disjoint though the hyperspheres overlap. Figure 7(b) shows an example of a query region, which is the dark shaded area, and the affected space of each pyramid, which is the shaded area bounded by the pyramid boundary and the dashed curve. For each partition, the area not contained by the query sphere does not contain any answers for the query. However, since the mapping is lossy, the corner area outside the query region has to be checked since the data points have the same mapping values as those in the area intersecting with the query region.

For reference points along the central axis, the partitions look similar to those of the Pyramid tree. When dealing with query and data points, the sets of points are however 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.

(2) **Center of Hyperplane, Furthest Distance.** The center of each hyperplane can be used as a reference point, and the partition associated with the point contains all points that are furthest from it. Figure 8(a) shows an example in a 2-dimensional space. Figure 8(b) shows the affected search

[1]We note that the space is similar to that of the Pyramid technique [Berchtold et al. 1998a]. However, the rationales behind the design and the 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 center of the space.


Furthest points from O 1
are located in this area.
O_3
O_2
O_0
O_1
r
Q
The bounding area by
these arches are the affected
searching area of kNN(Q,r).

(a) Space partitioning (b) Effect of reduction on query space

Fig. 8. Using (center of hyperplane, furthest distance) as reference point.

area for the given query point. The shaded search area is that required by the previous scheme, while the search area caused by the current scheme is bounded by the bold arches. As can be seen in Figure 8(b), the affected search area bounded by the bold arches is now greatly reduced as compared to the closest distance counterpart. We must however note that the query search space is dependent on the choice of reference points, partition strategy and the query point itself.

(3) **External Point.** Any point along the line formed by the center of a hyperplane and the center 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 outside the data space. This heuristic 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 furthest distance can be supported. Figure 9 shows an example of the closest distance scheme for a 2-dimensional space when using external points as reference points. Again, we observe that the affected search space for the same query point is reduced under an external point scheme (compared to using the center of the hyperplane).

4.2 Data-Based Partitioning

Equi-partitioning may seem attractive for uniformly distributed data. However, data in real life are often clustered or correlated. Even when no correlation exists in all dimensions, there are usually subsets of data that are locally correlated [Chakrabarti and Mehrotra 2000; Pagel et al. 2000]. In these cases, a more appropriate partitioning strategy would be used to identify clusters from the data space. There are several existing clustering schemes in the literature such

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

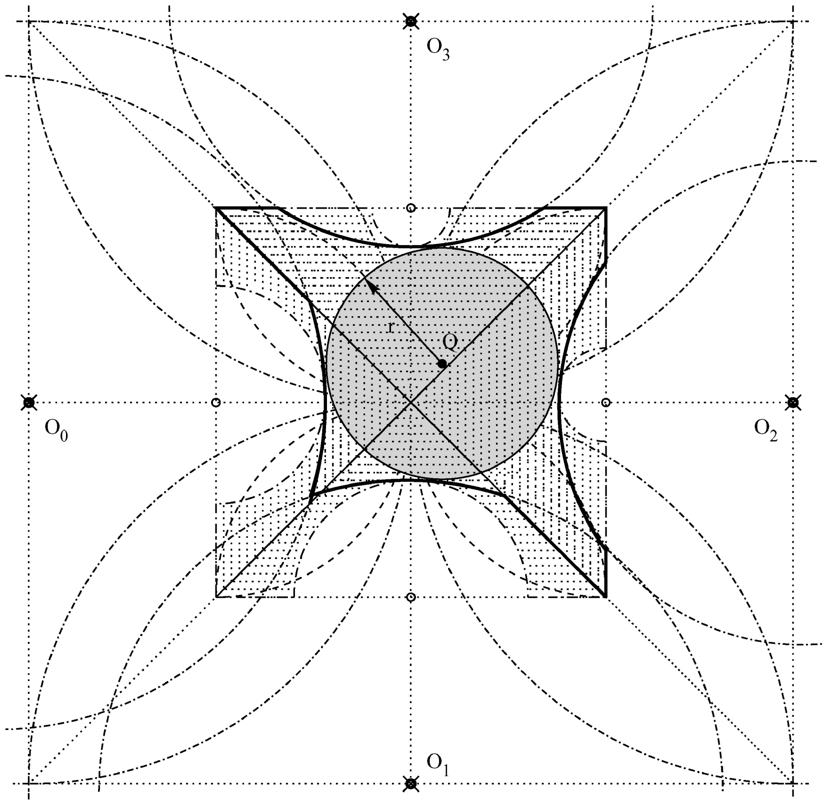


Fig. 9. Space partitioning under (external point, closest distance)-based reference point.

as K-Means [MacQueen 1967], BIRCH [Zhang et al. 1996], CURE [Guha et al. 1998], and PROCLUS [Aggarwal et al. 1999]. 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 implementation, we adopted the K-means clustering algorithm [MacQueen 1967]. The number of clusters affects the search area and the number of traversals from the root to the leaf nodes. We expect the number of clusters to be a tuning parameter, which may vary for different applications and domains.

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

(1) *Center of cluster.* The center of a cluster is a natural candidate as a reference point. Figure 10 shows a 2-dimensional example. Here, we have 2 clusters, one cluster has center O_1 and another has center O_2.
(2) *Edge of cluster.* As shown in Figure 10, when the cluster center is used, the sphere areas 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 partition as reference points, such as points on hyperplanes, data space corners, data points at one side of a cluster and away from other clusters, and so on. Figure 11 is an example of selecting

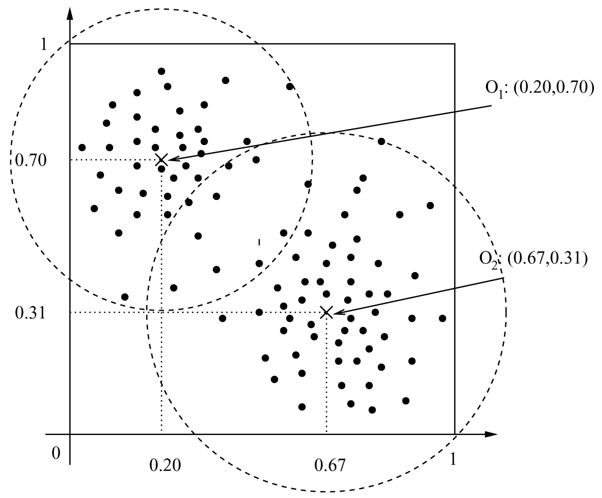


Fig. 10. Cluster centers and reference points.

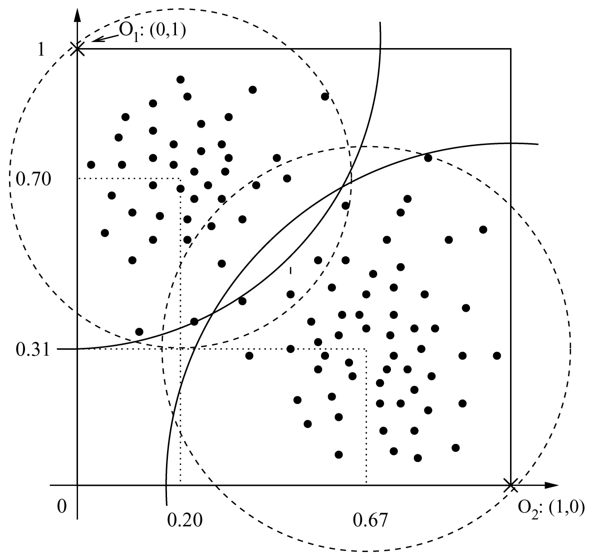


Fig. 11. Cluster edge points as reference points.

the edge points as the reference points in a 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 centers as reference points.

In short, overlap of partitioning spheres can lead to more intersections by the query sphere, and more points having the same similarity (distance) value will cause more data points to be examined if a query region covers that area. Therefore, when we choose a partitioning strategy, it is important to avoid or

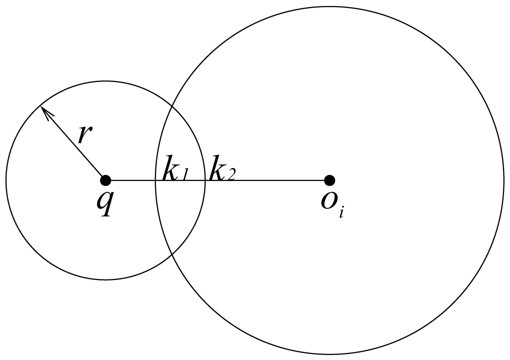


Fig. 12. Histogram-based cost model.

reduce such partitioning sphere overlap and large number of points with close similarity, as much as possible.

5. A COST MODEL FOR IDISTANCE

iDistance is designed to handle KNN search efficiently. However, due to the complexity of very high-dimensionality or the very large K used in the query, iDistance is expected to be superior for certain (but not all) scenarios. We therefore develop cost models to estimate the page access cost of iDistance, which can be used in query optimization (for example, if the iDistance has the number of page accesses less than a certain percentage of that of sequential scan, we would use iDistance instead of sequential scan). In this section, we present a cost model based on both the Power-method [Tao et al. 2003] and a histogram of the key distribution. This histogram-based cost model applies to all partitioning strategies and any data distribution, and it predicts individual query processing cost in terms of page accesses instead of average cost. The basic idea of the Power-method is to precompute the local power law for a set of representative points and perform the estimation using the local power law of a point close to the query point. In the key distribution histogram, we divide the key values into buckets and maintain the number of points that are in each bucket.

Figure 12 shows an example of how to estimate the page access cost for a partition P_i, whose reference point is O_i. q is the query point and r is the query radius. k_1 is on the line qO_i and with the largest key in the partition P_i. k_2 is the intersection of the query sphere and the line qO_i. First, we use the Power-method to estimate the Kth nearest neighbor distance r, which equals the query radius when the search terminates. Then we can calculate the key of k_2, $|qO_i| - r + i \cdot c$, where i is the partition number and c is the constant to stretch the key values. Since we know the boundary information of each partition and hence the key of k_1, we know the range of the keys accessed in partition P_i, that is, between the keys of k_2 and k_1. By checking the keys distribution histogram, we know the number of points accessed in this key range, $N_{a,i}$; then the number of pages accessed in the partition is $\lceil N_{a,i}/C_{eff} \rceil$. The summation of the number of page accesses of all the partitions provides us the number of page accesses for the query.

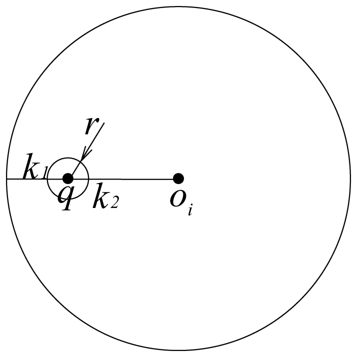


Fig. 13. Histogram-based cost model, query sphere inside the partition.

Note that, if the query sphere is inside a partition as shown in Figure 13, both k_1 and k_2 are intersections of the query sphere and the line $q\,O_i$. Different from the above case, the key of k_1 is $|q\,O_i| + r + i \cdot c$ here. The number of page accesses is derived in the same way as above.

The costs estimated by the techniques described above turn out to be very close to actual costs observed, as we will show in the experimental section that follows.

In Jagadish et al. [2004], we also present cost models purely based on formula derivations. They are less expensive to maintain and compute, in that no summary data structures need be maintained, but they assume uniform data distribution and therefore are not accurate for nonuniform workloads. Where data distributions are known, these or similar other formulae may be used to advantage.

6. A PERFORMANCE STUDY

In this section, we present results of an experimental study performed to evaluate iDistance. First we compare the space-based partitioning strategy and the data-based partitioning strategy and find that the data-based partitioning strategy is much better. Then we focus our study on the behavior of iDistance using the data-based partitioning strategy with various parameters and under different workloads. At last we compare iDistance with other metric based indexing methods, the M-tree and the Omni-sequential, as well as a main memory bd-tree [Arya et al. 1994]. We have also evaluated iDistance against iMinMax [Ooi et al. 2000] and A-tree [Sakurai et al. 2000], and our results, which have been reported in Yu et al. [2001] showed the superiority of iDistance over these schemes. As such, we shall not duplicate the latter results here.

We implemented the iDistance technique and associated search algorithms in C, and used the B^+-tree as the single dimensional index structure. We obtained the M-tree, Omni-sequential, and the bd-tree from the authors or their web sites, and standardized the codes as much as we could for fair comparison. Each index page is 4096 Bytes. Unless stated otherwise, all the experiments

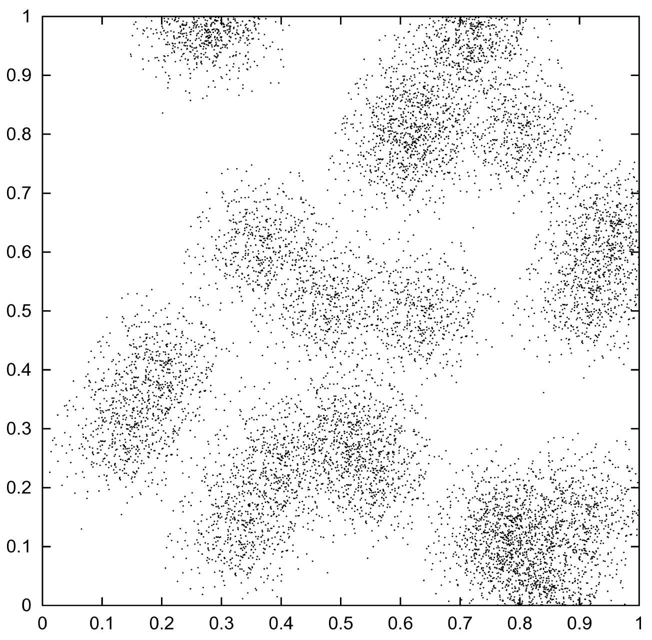


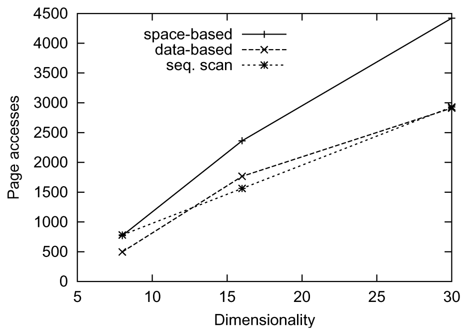
Fig. 14. Distribution of the clustered data.

were performed on a computer with Pentium(R) 1.6 GHz CPU and 256 MB RAM except the comparison with bd-tree (the experimental setting for this comparison would be specified later). The operating system running on this computer is RedHat Linux 9. We conducted many experiments using various datasets. Each result we show was obtained as the average (number of page accesses or total response time) over 200 queries that follow the same distribution of the data.

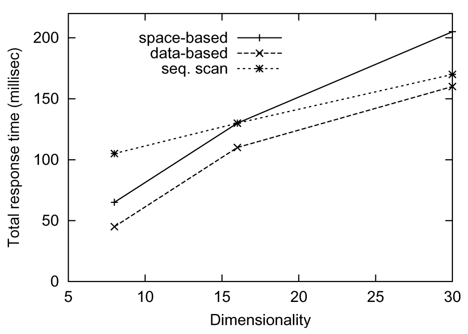
In the experiment, we generated 8, 16, 30-dimensional uniform, and clustered datasets. The dataset size ranges from 100,000 to 500,000 data points. For the clustered datasets, the default number of clusters is 20. The cluster centers are randomly generated and in each cluster, the data follow the normal distribution with the default standard deviation of 0.05. Figure 14 shows a 2-dimensional image of the data distribution.

We also used a real dataset, the Color Histogram dataset. This dataset is obtained from http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html. It contains image features extracted from a Corel image collection. HSV color space is divided into 32 subspaces (32 colors: 8 ranges of H and 4 ranges of S). And the value in each dimension in a Color Histogram of an image is the density of each color in the entire image. The number of records is 68,040. All the data values of each dimension are normalized to the range [0, 1].

In our evaluation, we use the number of page accesses and the total response time as the performance metric. Default value of Δr is 0.01, that is, 1% of the side length of the data space. The initial search radius is just set as Δr.



(a) Page accesses



(b) Total response time

Fig. 15. Space-based partitioning vs. data-based partitioning, uniform data.

6.1 Comparing Space-Based and Data-Based Partitioning Strategies

We begin by investigating the relative performance of the partitioning strategies. Note that the number of reference points is always $2d$ for the space-based partitioning approach, and for a fair comparison, we also use $2d$ reference points in the data-based partitioning approach. Figure 15 shows the result of 10NN queries on the 100,000 uniform dataset. The space-based partitioning has almost the same page accesses as sequential scan when dimensionality is 8 and more page accesses than sequential scan in high dimensionality. The data-based partitioning strategy has fewer page accesses than sequential scan when dimensionality is 8, more page accesses when dimensionality is 16, and almost the same page accesses when dimensionality is 30. This is because the pruning effect of the data-based strategy is better in low dimensionality than in high dimensionality. The relative decrease (compared to sequential scan) of page accesses when dimensionality is 30 is because of the larger number of reference points. While iDistance's page access performance is not attractive relative to sequential scan, the total response time performance is better because of its ability to filter data using a single dimensional key. The total response time of the space-based partitioning is about 60% that of sequential scan when dimensionality is 8, same as sequential scan when dimensionality is 16, but worse than sequential scan when dimensionality is 30. The total response time of the data-based partitioning is always less than both of the others, while its difference from the sequential scan decreases as dimensionality increases.

Figure 16 shows the result of 10NN queries on the 100,000 clustered dataset. Both partitioning strategies are better than sequential scan in both page accesses and total response time. This is because for clustered data, the Kth nearest neighbor distance is much smaller than that in the uniform data. In this case, iDistance can prune a lot of data points in the searching. The total response time of the space-based partitioning is about 20% that of sequential scan. The total response time of data-based partitioning is less than 10% that of sequential scan. Again, the data-based partitioning is better than both of the others.

In Section 4.1, we discussed using external point as the reference points of the space-based partitioning. A comparison between using external points and

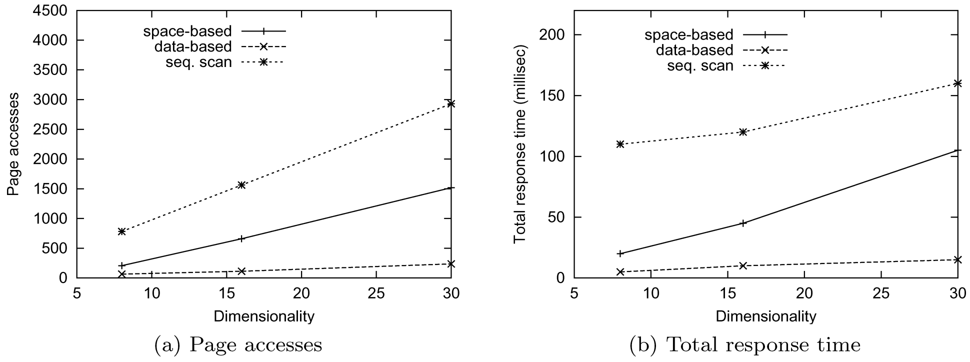


Fig. 16. Space-based partitioning vs. data-based partitioning, clustered data.

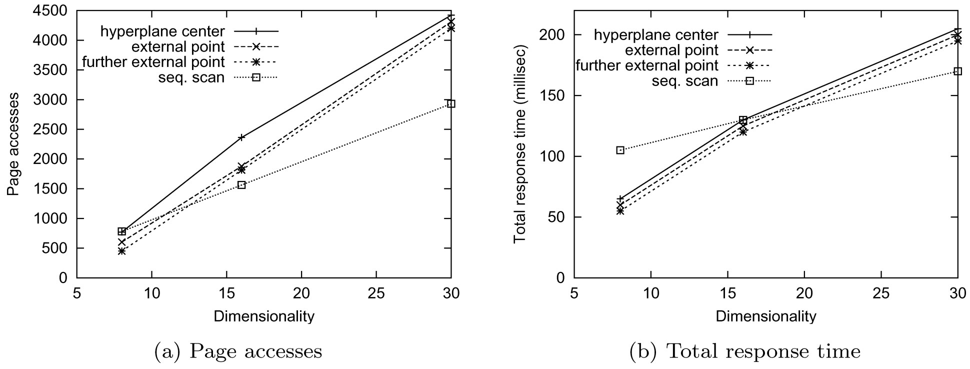


Fig. 17. Effect of reference points in space-based partitioning, uniform data.

the center point as the reference point on the uniform datasets is shown in Figure 17.

Using an external point as the reference point has slightly better performance than using the center point, and using a farther external point is slightly better than using the external point in turn, but the difference between them is not big, and all of them are still worse than the data-based partitioning approach (compare with Figure 15). Here, the farther external point is already very far (more than 10 times the side length of the data space) and the performance using even farther points almost does not change, therefore they are not presented.

From the above results, we can see that the data-based partitioning scheme is always better than the space-based partitioning approach. Thus, for all subsequent experimental study, we will mainly focus on the data-based partitioning strategy. However, we note that the space-based partitioning is always better than sequential scan in low and medium dimensional spaces (less than 16). Thus, it is useful for these workloads. Moreover, the scheme incurs much less overhead since there is no need to cluster data to find the reference points as in the data-based partitioning.

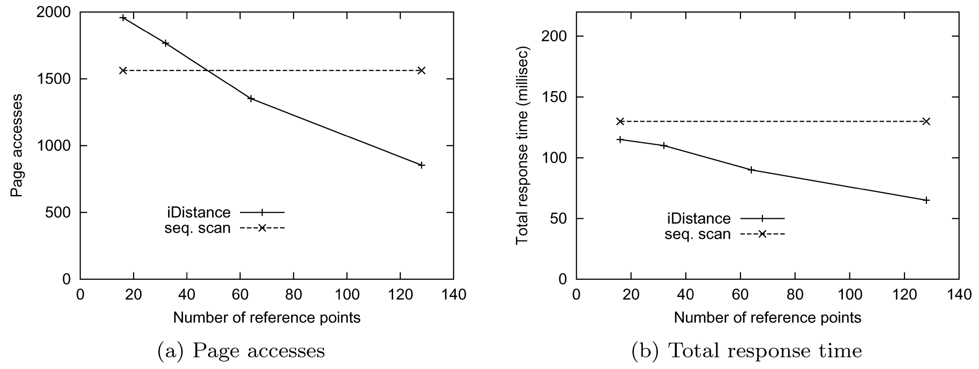


Fig. 18. Effects of number of reference points, uniform data.

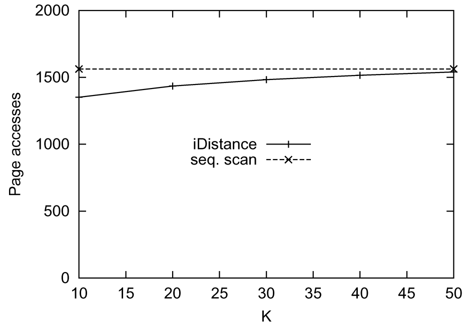
6.2 iDistance Using Data-Based Partitioning

In this subsection, we further study the performance of iDistance using a data-based partitioning strategy (iDistance for short in the sequel). We study the effects of different parameters and different workloads. As reference, we compare the iDistance with sequential scan. Although iDistance is better than sequential scan for the 30-dimensional uniform dataset, the difference is small. To see more clearly the behavior of iDistance, we use 16-dimensional data when we test on uniform datasets. For clustered data, we use 30-dimensional datasets since iDistance is still much better than sequential scan for such high dimensionality.

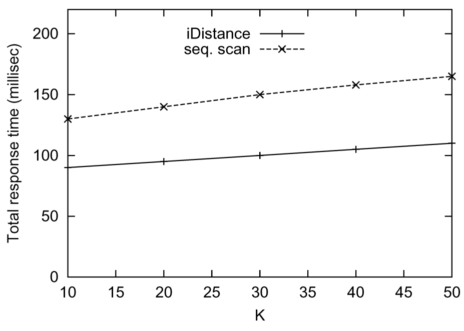
Experiments on Uniform Datasets

In the first experiment, we study the effect of the number of reference points on the performance of iDistance. The results of 10NN queries on the 100,000 16-dimensional uniform dataset are shown in Figure 18. We can see that as the number of reference points increases, both the number of page accesses and total response time decrease. This is expected, as smaller and fewer clusters need to be examined (i.e., more data are pruned). The amount of the decrease in time also decreases as the number of reference points increases. While we can choose a very large number of reference points to improve the performance, this will increase (a) the CPU time as more reference points need to be checked, and (b) the time for clustering to find the reference points. Moreover, there will also be more fragmented pages. So a moderate number of reference points is fine. In our other experiments, we used 64 as the default number of reference points.

The second experiment studies the effect of K on the performance of iDistance. We varied K from 10 to 50 at the step of 10. The results of queries on the 100,000 16-dimensional uniform dataset are shown in Figure 19. As expected, as K increases, iDistance incurs a larger number of page accesses. However, it remains superior over sequential scan. In terms of total response time, while both iDistance's and sequential scan's response times increase linearly as K increases, the rate of increase for iDistance is slower. This is because

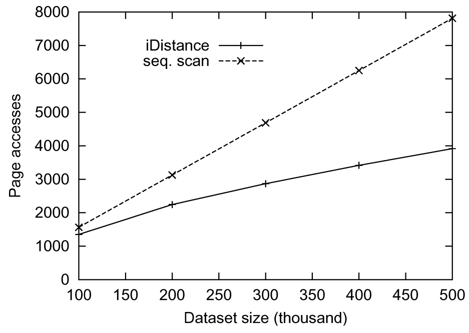


(a) Page accesses

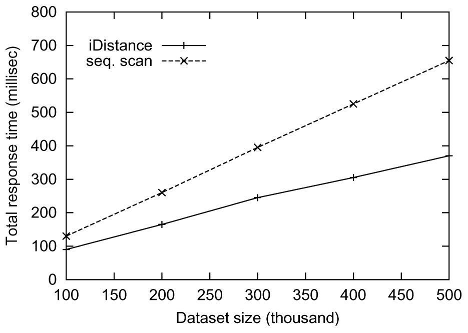


(b) Total response time

Fig. 19. Effects of K, uniform data.



(a) Page accesses



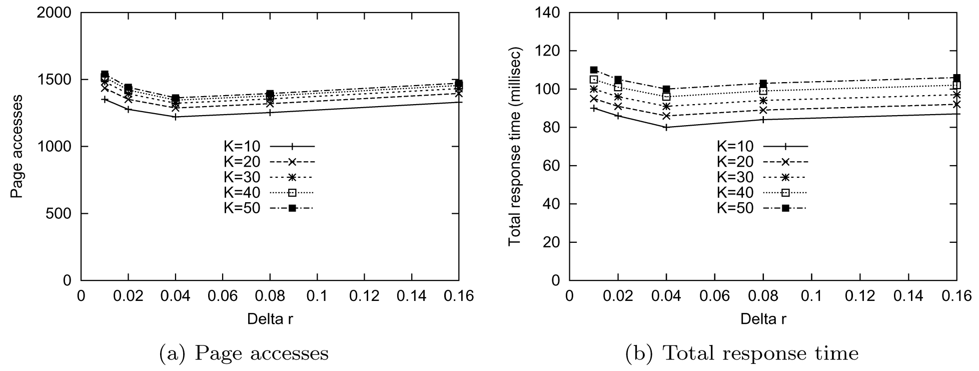
(b) Total response time

Fig. 20. Effects of dataset size, uniform data.

as K increases, the number of distance computations also increases for both iDistance and sequential scan. But, iDistance not only has fewer distance computations, the rate of increase in the distance computation is also smaller (than sequential scan).

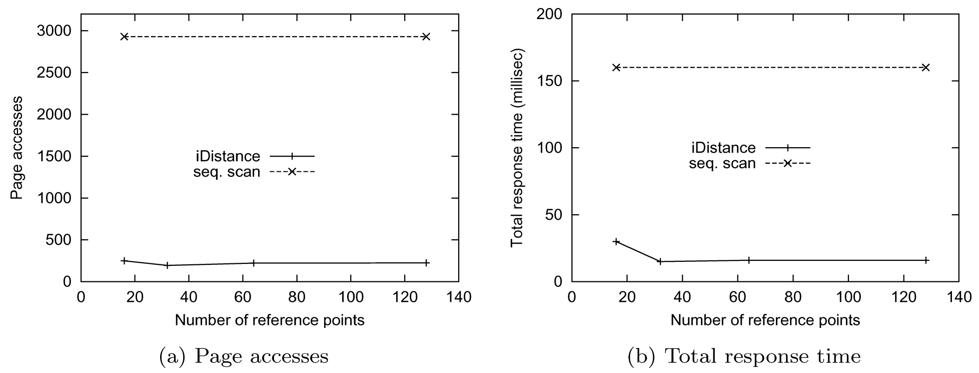
The third experiment studies the effect of the dataset size. We varied the number of data points from 100,000 to 500,000. The results of 10NN queries on five 16-dimensional uniform datasets are shown in Figure 20. The number of page accesses and the total response time of both iDistance and sequential scan increase linearly as the dataset size increases, but the increase for sequential scan is much faster. When the dataset size is 500,000, the number of page accesses and the total response time of iDistance are about half of that of sequential scan.

The fourth experiment examines the effect of the Δr in the iDistance KNN Search Algorithm presented in Figure 4. Figure 21 shows the performance when we varied the values of Δr. We can observe that, as Δr increases, both the number of page accesses and total response time decrease at first but then increase. For a small Δr, there will be more iterations to reach the final query radius and consequently, more pages are accessed and more CPU time is incurred. On the other hand, if Δr is too large, the query radius may exceed the KNN distance at



(a) Page accesses (b) Total response time

Fig. 21. Effects of Δr, uniform data.



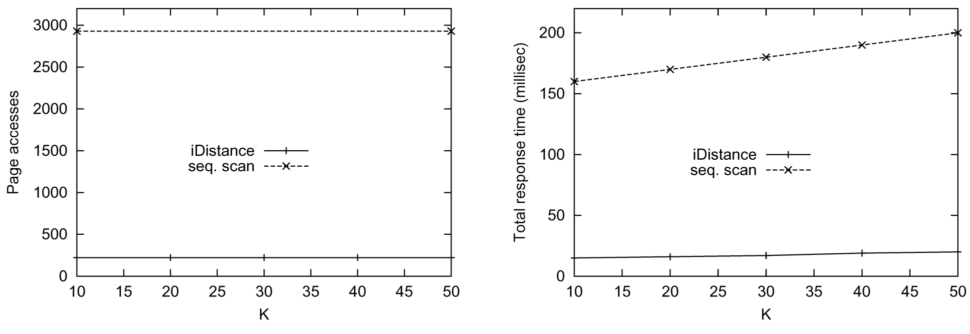
(a) Page accesses (b) Total response time

Fig. 22. Effects of number of reference points, clustered data.

the last iteration and redundant data pages are fetched for checking. We note that it is very difficult to derive an optimal Δr since it is dependent on the data distribution and the order in which the data points are inserted into the index. Fortunately, the impact on performance is marginal (less than 10%). Considering that, in practice, small K may be used in KNN search, which implies a very small KNN distance. Therefore, in all our experiments, we have safely set $\Delta r = 0.01$, that is, 1% of the side length of the data space.

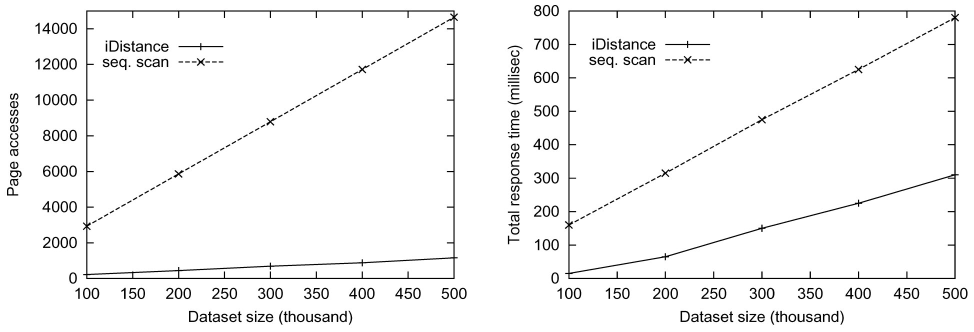
Experiments on Clustered Datasets

For the clustered datasets, we also study the effect of the number of the reference points, K, and dataset size. By default, the number of reference points is 64, K is 10 and dataset size is 100,000. Dimensionality of all these datasets is 30. The results are shown in Figures 22, 23 and 24 respectively. These results exhibit similar characteristics to those of the uniform datasets except that iDistance has much better performance compared to sequential scan. The speedup factor is as high as 10. The reason is that for clustered data, the Kth nearest neighbor distance is much smaller than that in uniform data, so many more data points can be pruned from the search. Figure 22 shows that after the number of reference points exceeds 32, the performance gain becomes



(a) Page accesses (b) Total response time

Fig. 23. Effects of K, clustered data.



(a) Page accesses (b) Total response time

Fig. 24. Effects of dataset size, clustered data.

almost constant. For the rest of the experiments, we use 64 reference points as default.

Each of the above clustered datasets consists of 20 clusters, each of which has a standard deviation of 0.05. To evaluate the performance of iDistance on different distributions, we tested three other datasets with different numbers of clusters and different standard deviations, while other settings are kept at the default values. The results are shown in Figure 25. Because all these datasets have the same number of data points but only differ in distribution, the performance of sequential scan is almost the same for all of them, hence we only plot one curve for sequential scan on these datasets. We observe that the total response time of iDistance remains very small for all the datasets with standard deviation σ less than or equal to 0.1 but increases a lot when the standard deviation increases to 0.2. This is because as the standard deviation increases, the distribution of the dataset becomes closer to uniform distribution, which is when iDistance becomes less efficient (but is still better than sequential scan).

We also studied the effect of different Δr on the clustered datasets. Like the results on the uniform datasets, the performance change is very small.

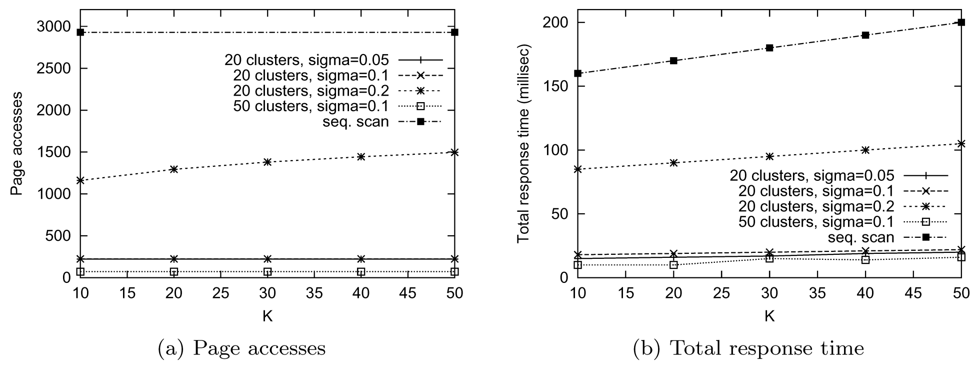


Fig. 25. Effects of different data distribution, clustered data.

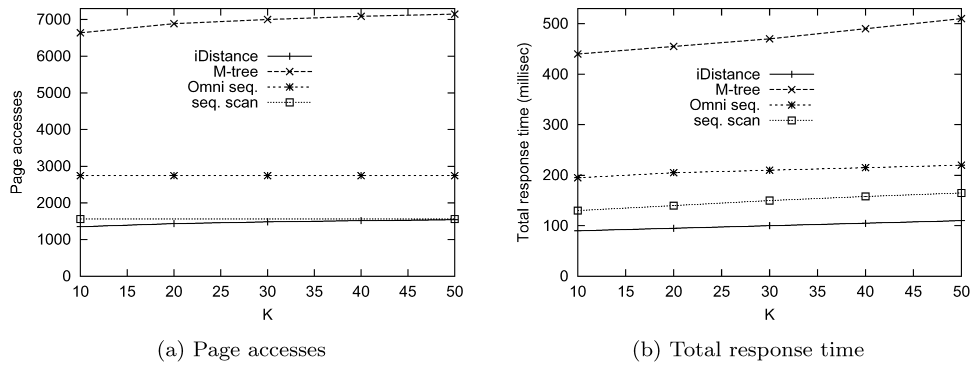
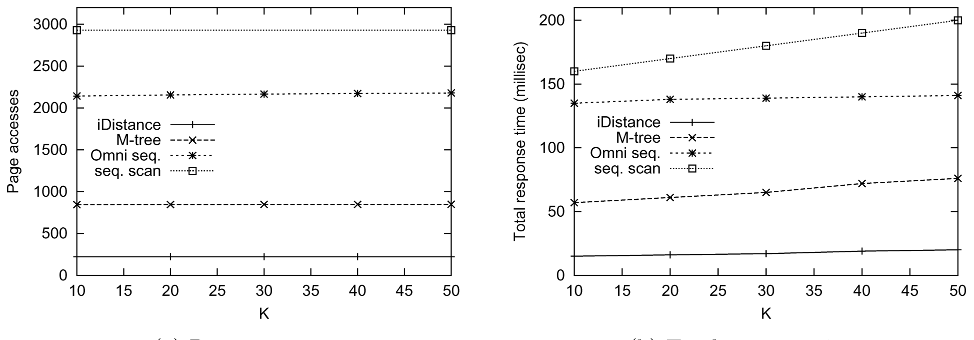


Fig. 26. Comparative study, 16-dimensional uniform data.

6.3 Comparative Study of iDistance and Other Techniques

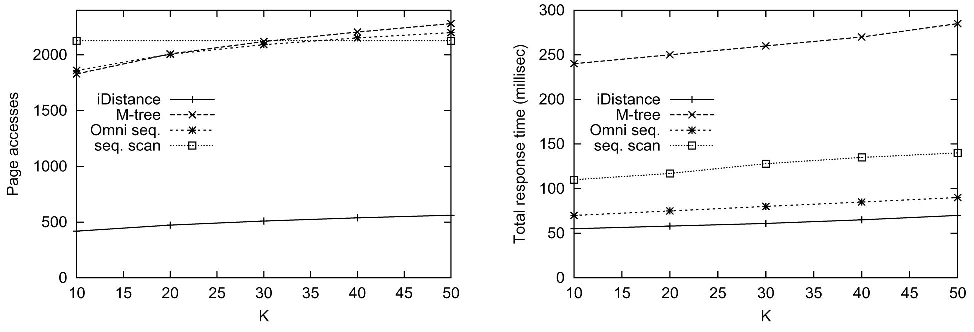
In this subsection, we compare iDistance with sequential scan and two other metric based indexing methods, the M-tree [Ciaccia et al. 1997] and the Omni-sequential [Filho et al. 2001]. Both the M-tree and the Omni-sequential are disk-based indexing schemes. We also compare iDistance with a main memory index, the bd-tree [Arya et al. 1994] in the environment of constrained memory. In Filho et al. [2001], several indexing schemes of the Omni-family were proposed, and the Omni-sequential was reported to have the best average performance. We therefore pick the Omni-sequential from the family for comparison. The Omni-sequential needs to select a good number of foci bases to work efficiently. In our comparative study, we tried the Omni-sequential for several numbers of foci bases and only presented the one giving the best performance in the sequel. We still use 64 reference points for iDistance. Datasets used include 100,000 16-dimensional uniformly distributed points, 100,000 30-dimensional clustered points and 68040 32-dimensional real data. We varied K from 10 to 50 at the step of 10.

First we present the comparison between the disk-based methods. The results on the uniform dataset are shown in Figure 26. Both the M-tree and the Omni-sequential have more page accesses and longer total response time than



(a) Page accesses (b) Total response time

Fig. 27. Comparative study, 30-dimensional clustered data.

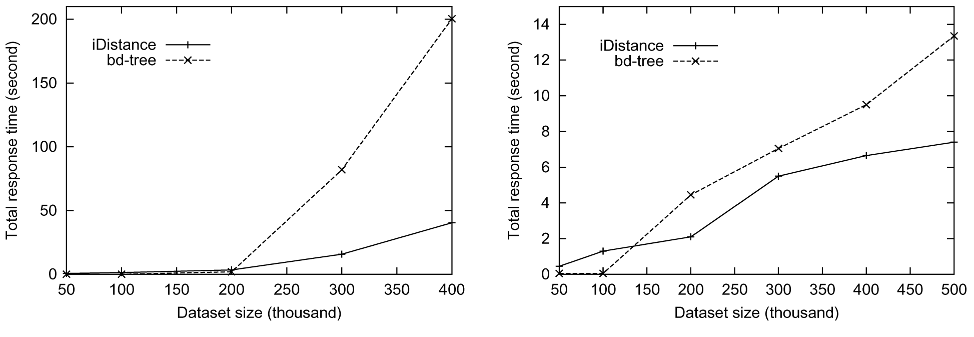


(a) Page accesses (b) Total response time

Fig. 28. Comparative study, 32-dimensional real data.

sequential scan. iDistance has similar page accesses to sequential scan, but shorter total response time than sequential scan. The results on the clustered dataset are shown in Figure 27. The M-tree, the Omni-sequential and iDistance are all better than sequential scan because the smaller Kth nearest neighbor distance enables more effective pruning of the data space for these metric based methods. iDistance performs the best. It has a speedup factor of about 3 over the M-tree and 6 over the Omni-sequential. The results on the real dataset are shown in Figure 28. The M-tree and the Omni-sequential have similar page accesses as sequential scan while the number of page accesses of iDistance is about 1/3 those of the other techniques. The Omni-sequential and iDistance have shorter total response times than sequential scan while the M-tree has a very long total response time. The Omni-sequential can reduce the number of distance computations, so it takes less time while having the same page accesses as sequential scan. The M-tree accesses the pages randomly, therefore it is much slower. iDistance has significantly fewer page accesses and distance computations, hence it has the least total response time.

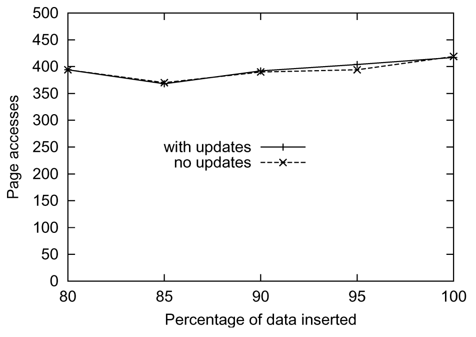
Next we compare the iDistance with the bd-tree [Arya et al. 1994]. The bd-tree was proposed to process approximate KNN queries, but it is able to return exact KNNs when the error bound ϵ is set to 0. All other parameters



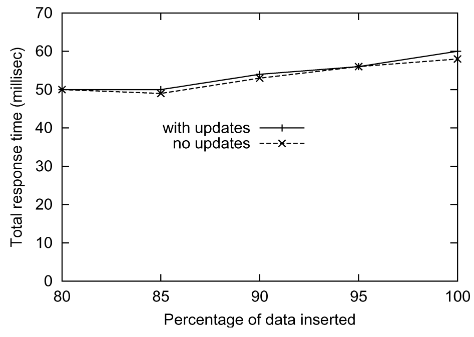
(a) 16-dimensional uniform data (b) 30-dimensional clustered data

Fig. 29. Comparison with a main memory index: bd-tree.

used in the bd-tree are set to the values suggested by the authors. The bd-tree is a memory resident index that loads the full index and data in memory, while iDistance reads in index and data pages from disk as and when they are required. To have a sensible comparison, we conducted this set of experiments on a computer with a small memory, whose size is 32M bytes. The CPU of the computer is a Pentium 266 MHz and the operating system is RedHat Linux 9. When the bd-tree runs out of memory, we let the operating system do the paging. As the performance of a main memory structure is affected more by the size of the dataset, we study the effect of dataset size instead of K. Since the main memory index has no explicit page access operation, we only present the total response time as the performance measurement. Figure 29(a) shows the results on the 16-dimensional uniform datasets. When the dataset is small (less than 200,000), the bd-tree is slightly better than iDistance; however, as the dataset grows beyond certain size (greater than 300,000), the total response time increases dramatically. When the dataset size is 400,000, the total response time of the bd-tree is more than 4 times that of iDistance. The reason is obvious. When the whole dataset can fit into memory, its performance is better than the disk-based iDistance, but when the data size goes beyond the available memory, thrashing occurs and impairs the performance considerably. In fact, the response time deteriorates significantly when the dataset size hits 300,000 data points or 19M bytes. The reason is that the operating system also uses up a fair amount of memory so the memory available for the index is less than the total. Figure 29(b) shows the results on the 30-dimensional clustered datasets. As before, the bd-tree performs well when the dataset size is small and degrades significantly when the dataset size increases. However, the trend is less intense than that of the uniform datasets, as the index takes advantage of the locality of the clustered data and hence less thrashing happens. The results on the 32-dimensional real dataset are similar to that of the 30-dimensional clustered dataset up to the point of dataset size of 50,000. Since the real dataset has a much smaller size than the available memory, the bd-tree performs better than iDistance. However, in practice, we probably would not have so much memory available for a single query processing process. Therefore, an efficient index must be scalable in terms of data size and be main memory efficient.



(a) Page accesses



(b) Total response time

Fig. 30. iDistance performance with updates.

6.4 On Updates

We use clustering to choose the reference points from a collection of data points, and fix them from that point onwards. It is therefore important to see whether a dynamic workload would affect the performance of iDistance much. In this experiment, we first construct the index using 80% of the data points from the real dataset. We run 200 10NN queries and record the average number of page accesses and total response time. Then we insert 5% of the data to the database and rerun the same queries. This process is repeated until the other 20% of the data are inserted. Separately, we also run the same queries on the index built based on the reference points chosen for 85%, 90%, 95% and 100% of the dataset. We compare the average number of page accesses and total response time of the two as shown in Figure 30. The difference between them is very small. The reason is that real data from the same source tends to follow a similar distribution, so the reference points chosen at different times are similar. Of course, if the distribution of the data changes too much, we will need choose the reference points again and rebuild the index.

6.5 Evaluation of the Cost Models

Since our cost model estimates page accesses of each individual query, we show the actual number of page accesses and the estimated page accesses from 5 randomly chosen queries on the real dataset in Figure 31. Estimation of each of these 5 queries has the relative error below 20%. For all the tested queries, the estimations of more than 95% of them achieve a relative error below 20%. Considering that iDistance often has a speedup factor of 2 to 6 over other techniques, the 20% error will not affect the query optimization result greatly.

We also measured the time needed for computing the cost model. The average computation time (including the time for retrieving the number from the histogram) is less than 3% of the average KNN query processing time. So this cost model is still a practical approach for query optimization.

6.6 Summary of the Experimental Results

The data-based partitioning approach is more efficient than the space-based partitioning approach. The iDistance using the data-based partitioning is

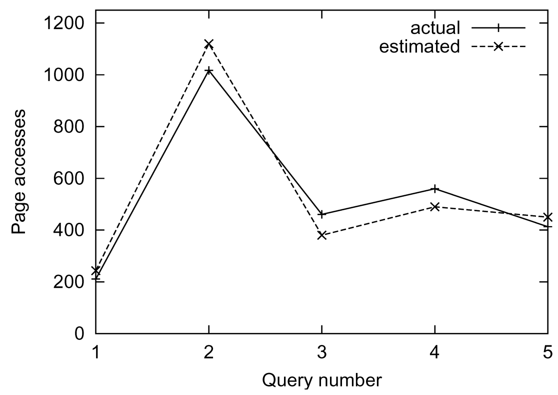


Fig. 31. Evaluation of the histogram-based cost model.

always better than the other techniques in all our experiments on various workloads. For uniform data, it beats sequential scan in dimensionality as high as 30. Of course, due to the intrinsic characteristics of the KNN problem, we expect iDistance to lose out to sequential scan in much higher dimensionality on uniform datasets. However, for more practical data distributions, where data are skew and clustered, iDistance shows much better performance compared with sequential scan. Its speedup factor over sequential scan is as high as 10.

The number of reference points is an important tunable parameter for iDistance. Generally, the more the number of reference points, the better the performance, and at the same time, the longer the time needed for clustering to determine these reference points. Too many reference points also impairs performance because of higher computation overhead. Therefore, a moderate number is fine. We have used 64 as the number of reference points in most of our experiments (the others are because we need to study the effects of number of reference points) and iDistance performs better than sequential scan and other indexing techniques in these experiments. For a dataset with unknown data distribution, we suggest 60 to 80 reference points. Usually iDistance achieves a speedup factor of 2 to 6 over the other techniques. We can use a histogram-based cost model in query optimization to estimate the page access cost of iDistance, which usually has a relative error below 20%.

The space-based partitioning is simpler and can be used in low and medium dimensional space.

7. CONCLUSION

Similarity search is of growing importance, and is often most useful for objects represented in a high dimensionality attribute space. A central problem in similarity search is to find the points in the dataset nearest to a given query point. In this article we have presented a simple and efficient method, called iDistance, for K-nearest neighbor (KNN) search in a high-dimensional metric space.

Our technique partitions the data and selects one reference point for each partition. The data in each cluster can be described based on their similarity with respect to a reference point, hence they can be transformed into a single dimensional space based on such relative similarity. This allows us to index the data points using a B^+-tree structure and perform KNN search using a simple one-dimensional range search. As such, the method is well suited for integration into existing DBMSs.

The choice of partition and reference points provides the iDistance technique with degrees of freedom that most other techniques do not have. We described how appropriate choices here can effectively adapt the index structure to the data distribution. In fact, several well-known data structures can be obtained as special cases of iDistance suitable for particular classes of data distributions. A cost model was proposed for iDistance KNN search to facilitate query optimization.

We conducted an extensive experimental study to evaluate iDistance against two other metric based indexes, the M-tree and the Omni-sequential, and the main memory based bd-tree structure. As a reference, we also compared iDistance against sequential scan. Our experimental results showed that iDistance outperformed the other techniques in most of the cases. Moreover, iDistance can be incorporated into existing DBMS cost effectively since the method is built on top of the B^+-tree. Thus, we believe iDistance is a practical and efficient indexing method for nearest neighbor search.

REFERENCES

AGGARWAL, C., PROCOPIUC, C., WOLF, J., YU, P., AND PARK, J. 1999. Fast algorithm for projected clustering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

ARYA, S., MOUNT, D., NETANYAHU, N., SILVERMAN, R., AND WU, A. 1994. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 573–582.

ARYA, S., MOUNT, D., NETANYAHU, N., SILVERMAN, R., AND WU, A. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM 45*, 6, 891–923.

BERCHTOLD, S., BÖHM, C., JAGADISH, H., KRIEGEL, H., AND SANDER, J. 2000. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proceedings of the International Conference on Data Engineering*. 577–588.

BERCHTOLD, S., BÖHM, C., AND KRIEGEL, H.-P. 1998a. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 142–153.

BERCHTOLD, S., ERTL, B., KEIM, D., KRIEGEL, H.-P., AND SEIDL, T. 1998b. Fast nearest neighbor search in high-dimensional space. In *Proceedings of the International Conference on Data Engineering*. 209–218.

BERCHTOLD, S., KEIM, D., AND KRIEGEL, H. 1996. The X-tree: An index structure for high-dimensional data. In *Proceedings of the International Conference on Very Large Data Bases*. 28–37.

BEYER, K., GOLDSTEIN, J., RAMAKRISHNAN, R., AND SHAFT, U. 1999. When is nearest neighbors meaningful? In *Proceedings of the International Conference on Database Theory*.

BÖHM, C., BERCHTOLD, S., AND KEIM, D. 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv. 33*, 3, 322–373.

BOZKAYA, T. AND OZSOYOGLU, M. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 357–368.

CHAKRABARTI, K. AND MEHROTRA, S. 1999. The hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the International Conference on Data Engineering*. 322–331.

CHAKRABARTI, K. AND MEHROTRA, S. 2000. Local dimensionality reduction: a new approach to indexing high dimensional spaces. In *Proceedings of the International Conference on Very Large Databases*. 89–100.

CIACCIA, P., PATELLA, M., AND ZEZULA, P. 1997. M-trees: An efficient access method for similarity search in metric space. In *Proceedings of the International Conference on Very Large Data Bases*. 426–435.

CUI, B., OOI, B. C., SU, J. W., AND TAN, K. L. 2003. Contorting high dimensional data for efficient main memory processing. In *Proceedings of the ACM SIGMOD Conference*. 479–490.

CUI, B., OOI, B. C., SU, J. W., AND TAN, K. L. 2004. Indexing high-dimensional data for efficient in-memory similarity search. *In IEEE Trans. Knowl. Data Eng*. to appear.

FALOUTSOS, C. AND LIN, K.-I. 1995. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 163–174.

FILHO, R. F. S., TRAINA, A., AND FALOUTSOS, C. 2001. Similarity search without tears: The omni family of all-purpose access methods. In *Proceedings of the International Conference on Data Engineering*. 623–630.

GOLDSTEIN, J. AND RAMAKRISHNAN, R. 2000. Contrast plots and p-sphere trees: space vs. time in nearest neighbor searches. In *Proceedings of the International Conference on Very Large Databases*. 429–440.

GUHA, S., RASTOGI, R., AND SHIM, K. 1998. Cure: an efficient clustering algorithm for large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 47–57.

JAGADISH, H., OOI, B. C., TAN, K.-L., YU, C., AND ZHANG, R. 2004. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. Tech. Rep. www.comp.nus.edu.sg/∼ooibc, National University of Singapore.

JOLLIFFE, I. T. 1986. *Principle Component Analysis*. Springer-Verlag.

KATAMAYA, N. AND SATOH, S. 1997. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

KOUDAS, N., OOI, B. C., TAN, K.-L., AND ZHANG, R. 2004. Approximate NN queries on streams with guaranteed error/performance bounds. In *Proceedings of the International Conference on Very Large Data Bases*. 804–815.

KRUSKAL, J. B. 1956. On the shortest spanning subtree of a graph and the travelling salesman problem. In *Proceedings of the American Mathematical Society* 7, 48–50.

LIN, K., JAGADISH, H., AND FALOUTSOS, C. 1995. The TV-tree: An index structure for high-dimensional data. *VLDB Journal 3*, 4, 517–542.

MACQUEEN, J. 1967. Some methods for classification and analysis of multivariate observations. In *Fifth Berkeley Symposium on Mathematical statistics and probability*. University of California Press, 281–297.

OOI, B. C., TAN, K. L., YU, C., AND BRESSAN, S. 2000. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 166–174.

PAGEL, B.-U., KORN, F., AND FALOUTSOS, C. 2000. Deflating the dimensionality curse using multiple fractal dimensions. In *Proceedings of the International Conference on Data Engineering*.

RAMAKRISHNAN, R. AND GEHRKE, J. 2000. *Database Management Systems*. McGraw-Hill.

SAKURAI, Y., YOSHIKAWA, M., AND UEMURA, S. 2000. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proceedings of the International Conference on Very Large Data Bases*. 516–526.

TAO, Y., FALOUTSOS, C., AND PAPADIAS, D. 2003. The power-method: A comprehensive estimation technique for multi-dimensional queries. In *Proceedings of the Conference on Information and Knowledge Management*.

TRAINA, A., SEEGER, B., AND FALOUTSOS, C. 2000. Slim-trees: high performance metric trees minimizing overlap between nodes. In *Advances in Database Technology—EDBT 2000, International*

Conference on Extending Database Technology, Konstanz, Germany, March 27–31, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1777. Springer-Verlag, 51–65.

WEBER, R., SCHEK, H., AND BLOTT, S. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the International Conference on Very Large Data Bases*. 194–205.

WHITE, D. AND JAIN, R. 1996. Similarity indexing with the SS-tree. In *Proceedings of the International Conference on Data Engineering*. 516–523.

YU, C., OOI, B. C., TAN, K. L., AND JAGADISH, H. 2001. Indexing the distance: an efficient method to knn processing. In *Proceedings of the International Conference on Very Large Data Bases*. 421–430.

ZHANG, T., RAMAKRISHNAN, R., AND LIVNY, M. 1996. Birch: an efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Received March 2003; revised March and September 2004; accepted October 2004