Efficient index-based KNN join processing for high-dimensional data

Cui Yu a,1, Bin Cui b,*, Shuguang Wang c, Jianwen Su d

a Department of Computer Science, Monmouth University, West Long Branch, NJ 07764, USA
b Department of Computer Science, Peking University, Beijing, China
c Department of Computer Science, National University of Singapore, Singapore
d Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

Received 10 October 2005; received in revised form 8 May 2006; accepted 25 May 2006
Available online 7 July 2006

Abstract

In many advanced database applications (e.g., multimedia databases), data objects are transformed into high-dimensional points and manipulated in high-dimensional space. One of the most important but costly operations is the similarity join that combines similar points from multiple datasets. In this paper, we examine the problem of processing K-nearest neighbor similarity join (KNN join). KNN join between two datasets, R and S, returns for each point in R its K most similar points in S. We propose a new index-based KNN join approach using the iDistance as the underlying index structure. We first present its basic algorithm and then propose two different enhancements. In the first enhancement, we optimize the original KNN join algorithm by using approximation bounding cubes. In the second enhancement, we exploit the reduced dimensions of data space. We conducted an extensive experimental study using both synthetic and real datasets, and the results verify the performance advantage of our schemes over existing KNN join algorithms.

Keywords: High-dimensional data; KNN; Similarity join

1. Introduction

Many emerging database applications such as CAD, image, time series, molecular biology and scientific databases represent their data as multi/high-dimensional feature vectors. Each feature vector consists of d values which can be interpreted as coordinates in a d-dimensional space plus some associated content data. The distance between two feature vectors is commonly used to measure the degree of (dis-)similarity between the original high-dimensional objects (with regard to the feature represented).

Most existing work focuses on window and similarity queries, however, there is increasing need to combine similar tuples (points) of two datasets for many applications. The operation of generating all pairs is in essence similarity join. If each point of R is combined with the similar point(s) in the same dataset R, the operation is self similarity join. A KNN similarity join (KNN join), between two datasets, R and S, returns K most similar points in S for each point in R [6,7,26]. If R and S is the same dataset, then the KNN join is a self KNN join. The work on similarity join is motivated by the observation that one-time computing the neighbors for all data points can dramatically accelerate the operations of the traditional one-point-based similarity search. It can be used to support various applications where multidimensional data are involved. In particular, it is identified that many algorithms in almost all stages of knowledge discovery process can be accelerated by using KNN join as a primitive operation. For example, data mining algorithms can be implemented on the top of one join operation, instead of many similarity queries. Other applications include K-nearest neighbor classification [12], K-means clustering [16], sample assessment and sample postprocessing, missing value imputation, k-distance diagrams, etc. [8].
In this paper, we study the problem of processing \textit{KNN join} and propose a novel index-based KNN join solution. In our design, we use \textit{iDistance} \cite{25,18} as its underlying index structure for several good reasons.

- \textit{iDistance} \cite{25,18} has been shown to be one of the most efficient access techniques with respect to \textit{KNN} queries.
- With B**'-tree as its base structure, \textit{iDistance} can be easily integrated into existing commercial database systems. Algorithms built on top of \textit{iDistance} can inherit the same benefit of \textit{iDistance}, that is, the potential of being easily crafted into existing DBMSs. It is highly desirable in practice.
- Additionally, since the KNN join algorithms trigger a great amount of tree traversals, the flexible data partition strategies of \textit{iDistance} can significantly reduce this cost compared with other high-dimensional indexes.

In this paper, we propose a new index-based KNN join approach named as \textit{iJoin}, which exploits the \textit{iDistance} index structure maintained for efficient KNN retrievals. \textit{iJoin} has significant performance advantage, because it can make great use of the feature of data set partition in \textit{iDistance}. Such feature can help the \textit{iJoin} algorithms efficiently filter out false join answer candidates during processing. Several algorithms are designed in our work. We call the basic algorithm of \textit{iJoin} as \textit{iJoin}.

To further reduce disk I/O and computational costs, we propose two variations with different enhancements for \textit{iJoin}. The first enhancement applies the concept of approximation bounding cube to reduce unnecessary KNN computation and disk access. We name this enhanced algorithm \textit{iJoinAC}. The second enhancement is motivated by the observation \cite{10} that high-dimensional data points can be approximated and indexed based on a smaller number of dimensions. We name this enhanced algorithm \textit{iJoinDR}. \textit{iJoinDR} first sorts the dimensions of data points in the order of dimension extents, then build an \textit{iDistance} index with a subset of important dimensions, and finally exploits an efficient filter-and-refine mechanism using partial distance. With the reduced dimensions in the index structure and efficient filtering, \textit{iJoinDR} sets to remove as many false positives as possible in order to optimize both disk I/O and CPU costs.

To evaluate the performance of the proposed algorithms, we conducted an extensive performance study using both synthetic and real datasets. The experimental results show that the proposed algorithms are superior to the existing KNN join methods, with the performance gap increasing with the number of dimensions. In addition, using \textit{iDistance} as the underlying index structure also make \textit{iJoin} and its variations with unique features that other join algorithms do not have.

The remainder of our paper is organized as follows: In Section 2, we review the existing similarity join algorithms and the \textit{iDistance} structure. In Section 3, we introduce and discuss our new approach: KNN join with \textit{iDistance}. The experimental evaluation of our approach is presented in Section 4, and Section 5 concludes the paper.

2. Related work

Most existing work on high-dimensional join processing is based on spatial join. The spatial join was originally defined for two-dimensional objects, where the join predicate is typically the intersection between two objects. Many spatial join techniques have been proposed in the literature \cite{9,17,20,2,22}.

More recently, there have been some similarity join algorithms \cite{4,23,19,1} designed to fetch all pairs of objects with a distance not exceeding a user-given parameter \(\epsilon\). The join algorithm based on the \(\epsilon\)-dB-tree \cite{23} is particularly suited for similarity self join. Its basic idea is to partition the datasets perpendicularly to one selected dimension and into stripes of width \(\epsilon\), so as to restrict the join to pairs of subsequential stripes. Epsilon Grid Order \cite{4} was proposed to explicitly deal with massive datasets and it tries to avoid scalability problems by defining a strict order on data points. The order is obtained by laying an equidistance grid with cell length \(\epsilon\) over the data space and then comparing the grid cells lexicographically.

The above algorithms were proposed to evaluate all pairs of points with a distance not exceeding a user-given parameter \(\epsilon\). Therefore, these kinds of indexes have to be built on the fly for dynamic \(\epsilon\). On the other hand, the parameter \(\epsilon\) is not easy to estimate in most cases, and the size of the answer of the similarity join cannot be predefined. These approaches to processing \(\epsilon\) similarity join cannot not be easily extended to efficiently support KNN similarity join due to the difficulty in pre-determining the search radius in KNN join.

The MuX KNN join algorithm \cite{7,8} was specifically designed for KNN join. MuX is essentially an R-tree based method designed to satisfy the conflicting optimization requirements of CPU and I/O costs. It employs large-sized pages (the hosting page) to optimize I/O time and uses the secondary structure – the buckets which are MBRs (minimum bounding rectangles) of much smaller size – to partition the data with finer granularity so that CPU cost can be reduced. MuX iterates over \(R\) pages, and for the \(R\) page in the memory, potential KNN-joinable pages in \(S\) are retrieved through the MuX index on \(S\) and searched for \(K\)-nearest neighbors. MuX makes use of an index to reduce the number of data pages retrieved, however it suffers as an R-tree based join algorithm. First, like the R-tree, its performance is expected to degenerate with the increase of data dimensionality. Second, the memory overhead of the MuX index structure is high for large high-dimensional data due to the space requirement of high-dimensional minimum bounding boxes. Both constraints restrict the scalability of the MuX KNN join method in terms of dimensionality and data size. For the above reason, we try to use a non-R-tree based indexing structure in our design.
In [25], iDistance was proposed to support KNN search for high-dimensional data. 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, and the points which are near in high-dimensional space are expected to have similar distance to the reference point. Third, distance is essentially a single value. Hence, high-dimensional data can be represented in single-dimensional space, thereby enabling the reuse of existing single-dimensional indexes such as the B\(^+\)-tree.

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, suppose we have \( m \) partitions, \( P_1, P_2, \ldots, P_m \); their corresponding reference points, \( O_1, O_2, \ldots, O_m \), are selected either based on a pre-defined space partitioning strategy or data partitioning strategy. In the third step, all data points are represented in a single dimensional space as follows. A data point \( p(v_1, v_2, \ldots, v_d) \), where \( 0 \leq v_j \leq 1 \) and \( 1 \leq j \leq d \), has an index key \( y \) based on the distance from the nearest reference point \( O_i \): \( y = i \cdot c + \text{Dist}(p, O_i) \). Here, \( \text{Dist}(p, O_i) \) 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 dimensional space into regions so that all points in Partition \( P_i \) will be mapped to the range of \([i \cdot c, (i + 1) \cdot c] \).

In iDistance, two data structures are employed:

- A B\(^+\)-tree is used to index the transformed points to facilitate speedy retrieval. The B\(^+\)-tree is used for its wide availability in commercial systems.
- An array is used to store the \( m \) reference points and their respective nearest and furthest radii that define the data space.

Searching in iDistance begins with the scanning of the auxiliary structure to identify the reference points whose data space (spherical 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. We first locate the leaf node which covers or is nearest to \( q \), and later search its sibling leaf nodes if necessary. 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 the 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 \( \text{queryDist} \). This occurs when the distance of the furthest object in the answer set 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.

With appropriate choice of partition schemes, iDistance is one of the most efficient access techniques with respect to KNN search. It is true that the power of pruning methods deteriorates with increasing dimensionality and parameter \( K \), but it is less dramatic for iDistance. In [25,18], it shows the improvement factor of iDistance with increasing data space dimensionality, compared with other techniques.

Because of its special characteristics and advantages, we choose to use the B\(^+\)-tree based iDistance in our design as the underlying index structure.

3. iJoin: KNN join with iDistance

It has recently been recognized that the single similarity join operation can form the basis of many algorithms of similarity search queries. Thus, we expect that an index supporting efficient KNN search can support efficient KNN join processing as well. In this section, we introduce our KNN similarity join approach iJoin, which uses iDistance indexes in join processing. Three algorithms will be presented, including the basic iJoin and its variations iJoin-AC and iJoinDR.

3.1. Basic algorithm of iJoin

In many existing data mining algorithms, multiple similarity search queries are explored by iteratively considering the neighbors of visited objects. Most of these techniques ignore the relationship among the data points. In our design, we exploit such relationship features via iDistance. For the rest of this section, we will introduce the basic scheme of iJoin that supports multiple similarity search queries and KNN joins.

We observe that the choice of reference points in iDistance can affect performance in terms of both CPU time and I/O page accesses. To support more efficient processing, we propose a clustering method to initiate reference points when the index tree is constructed. For each dataset, we use K-means clustering [16] method to partition the dataset and each cluster center is used as reference point of the partition. The KNN join algorithm can benefit more from iDistance than other index structures because iDistance reference points for two join datasets can be the same regardless of different data distributions. For example, we only apply K-means clustering on the dataset \( R \), and use the reference points of \( R \) for dataset \( S \) as well. A certain Partition \( P \) in \( R \) has a related Partition \( P' \) in \( S \) which has the same reference point as \( P \), and almost overlaps with \( P \). Hence, the range of the keys can also overlap in the transformed single-dimensional space, i.e., \( P' \) usually contains most of the KNN join mates of the points in \( P \).

A more general join query example is shown in Fig. 1. As illustrated, we have a partition \( R_1 \) of dataset \( R \) and three partitions \( S_1, S_2 \) and \( S_3 \) of dataset \( S \), where join sets \( R \) and \( S \) have different reference points. To get all the join mates of \( R_1 \), we extend the radius \( r \) of \( R_1 \) to Search Radius,
and the overall query space is the dotted area of $S_1$ and $S_2$. $S_3$ can be excluded directly as it is far away.

After the process of index construction, two B+-trees are built for two datasets whose leaf nodes store high-dimensional vector points. Now we present how KNN join can be conducted using the iDistance index structure. Figs. 2–4 show the details of the basic iJoin algorithm: \textit{iJoin}.

The function \textit{iJoin} takes three arguments. $R$ and $S$ are two datasets indexed with iDistance and $K$ is the number of nearest neighbors in $S$ to be joined for each point in $R$. It returns all join pairs from $R$ and $S$ such that for each point in $R$, $K$ nearest neighbors are found from $S$. The outer set $R$ is traversed page by page and every page is stored in the variable $noder$, where each page represents a leaf node. $min$ and $max$ are the minimum and maximum key values (the first and last keys) in $noder$. If the first and the last entries in $noder$ belong to the same partition, we can proceed to join $noder$ with $S$ starting with the nearest node of a data partition; otherwise, we need to divide $noder$ into partitions, and join each of them with partitions of $S$. $r$ _radius is the distance of entry to its $K$th nearest neighbor candidate, which is used to filter the points in Partition $p_r$.

The purpose of the algorithm \textit{iTraverse} is quite straightforward. In Line 1, we determine the range of $ps$ which may contain the answer, by calling the function \textit{GetBoundary} in Fig. 4. For all the points in the range, we check whether they are the $K$ nearest neighbor of the entry. If the answer is yes, we insert the candidate pairs into the output list. When new NN candidates are inserted, the current $K$th NN might be changed. If this happens, the value of searching boundary $r$ _radius needs to be updated. Note, search space enlarges with the accesses of leaf nodes in each affected data partitions, however $r$ _radius reduces each time when a nearer point as the $K$th NN is found. The search stops when the completed search space has radius (search radius) equal to the value of $r$ _radius.

In Figs. 5 and 6, we use a simple example to demonstrate the process of \textit{iJoin}. Suppose we want to join sets $R$ and $S$, both are partitioned into two clusters with the same reference points. Two B+-tree based iDistance indexes are built to index the datasets $R$ and $S$, where $R$ has five leaf nodes and $S$ has six leaf nodes, respectively. We start the join with partition $R_11$. Given a search radius, we can prune partitions $S_13$, $S_21$, $S_22$ and $S_23$. Then we join $R_11$ with two nodes $S_11$ and $S_12$. If all the join pairs are found, we can stop processing on $R_11$ and continue to operate on other nodes. If not, we increase the radius and search the other nodes of $S$. Fig. 6 shows the join pairs, e.g., $R_11$ with $S_11$ and $S_12$, $R_23$ with $S_22$, $S_23$ and $S_13$.

### 3.2. Minimizing candidate set by using approximation bounding cube

Typically, the consumption of CPU time is proportional to the number of floating point distance computations required for computing the join condition (i.e., the test whether two data points are similar enough). Hence, in the first enhancement, our primary goal is to reduce the number of distance computations for every traversal in the leaf level of the index.

---

**Algorithm iJoin**(entry, $p_r$, $r$ _radius)$\\$

\[\text{Input: Entry entry, Partition $p_r$, current Kth NN distance $r$ _radius} \]

\[\text{Output: KNN Join answer set} \]

\[\text{1. FOR each node, in $R$ DO} \]
\[\text{2. $min = MinKey(node,)$} \]
\[\text{3. $max = MaxKey(node,)$} \]
\[\text{4. IF $min$ and $max$ belong to two different partitions} \]
\[\text{5. Divide node, into two parts;} \]
\[\text{6. FOR each entry, of node, (in each part) } \]
\[\text{7. Init( neighbors, $K$);} \]
\[\text{8. $r$ _radius;} \]
\[\text{9. WHILE not all neighbors found DO} \]
\[\text{10. locate the nearest unprocessed Partition $p_r$ of $S$;} \]
\[\text{11. iTraverse(entry, $p_r$, $r$ _radius);} \]
\[\text{12. Append the data point of entry, and its neighbors to the answer set;} \]

---

**Algorithm iTraverse**(entry, $p_r$, $r$ _radius)$\\$

\[\text{//Note: Modification of $r$ _radius will be done on the original} \]

\[\text{Input: Entry entry, Partition $p_r$, current Kth NN distance $r$ _radius} \]

\[\text{Output: new search range [start, end] in $p_r$} \]

\[\text{1. $value = iDistance$ value of the point of entry, as it belongs to $p_r$;} \]
\[\text{2. $start = max(value, r$ _radius, $MinKey(p_r)$);} \]
\[\text{3. $end = min(value, + r$ _radius, $MaxKey(p_r)$);} \]
\[\text{4. return [start,end];} \]

---

**Algorithm GetBoundary**(entry, $p_r$, $r$ _radius)$\\$

\[\text{Input: Entry entry, Partition $p_r$, current Kth NN distance $r$ _radius} \]

\[\text{Output: new search range [start, end] in $p_r$} \]

\[\text{1. $value = iDistance$ value of the point of entry, as it belongs to $p_r$;} \]
\[\text{2. $start = max(value, r$ _radius, $MinKey(p_r)$);} \]
\[\text{3. $end = min(value, + r$ _radius, $MaxKey(p_r)$);} \]
\[\text{4. return [start,end];} \]
Our solution is to represent each data point in both $R$ and $S$ by an approximation bounding cube, then process these approximation cubes before we compute the similarity of candidate pairs [11]. iJoin with this enhancement is called iJoinAC. To facilitate this process, we introduce a new level in the iDistance structure which indexes the approximation of real feature vectors as shown in Fig. 7.

Processing approximation cubes is much more efficient than processing actual feature vectors because approximation cubes are usually much smaller in size. With approximation cubes, we can filter out most false join answers before we access the real vectors [24]. The approximation cubes we use do not refer to a specific structure, but rather, to a generic template which can be integrated with many quantization/coding techniques. We apply a simple quantization method in iJoinAC. Let $V(v_1, \ldots, v_d)$ be a $d$-dimensional vector, where $v_i$ (in $[0,1]$) is the float value of the feature vector on dimension $i$. The corresponding approximation cube can then be represented as $A(a_1, \ldots, a_d)$, where $a_i$ is the length (integer) of the side of the cube on dimension $i$. To define it more specifically, $a_i$ can easily be computed from $v_i$, using the following formula, where $q$ ($q \geq 1$) is an integer.

$$a_i = \left\lceil v_i \times 2^q \right\rceil$$

It is not difficult to verify that the value of $a_i$ falls in $[0,2^q)$. This is a very simple quantization function to implement the approximation cube. However, many other coding techniques can be very effective as well. Other quantization methods such as run length coding, Huffman coding and entropy coding are also applicable after the floating point vectors are transformed into integer domain with Fourier Transform.

The iJoinAC algorithm is quite similar to the iJoin algorithm except for the leaf level traversal. As shown in Fig. 8, as different from the original traversal in iJoin,
Algorithm $i$TraverseAC($entry_y$, $p_s$, $r_{radius}$)

Input: Entry $entry_y$, Partition $p_s$, current $K$th NN distance $r_{radius}$

1. $\{\text{start, end}\} = GetBoundary(entry_y, p_s, r_{radius})$
2. FOR each $entry_y$ in $p_s$, DO
3. IF $entry_y$, key $\in$ $\{\text{start, end}\}$
4. IF $UBound(entry_y, entry_y) \leq r_{radius}$
5. InsertCandidate(neighbors, $entry_y$);
6. ELSE IF $LBound(entry_y, entry_y) > r_{radius}$
7. IF $Dist(entry_y, entry_y) \leq r_{radius}$
8. InsertCandidate(neighbors, $entry_y$);
9. IF current $K$th NN is changed
10. update $r_{radius} = Dist(Kth\text{-}NN, entry_y)$;
11. $\{\text{start, end}\} = GetBoundary(entry_y, p_s, r_{radius})$;

Fig. 8. Traverse algorithm for $i$JoinAC.

$i$TraverseAC mainly processes approximation cubes (not real feature vectors). Function GetBoundary calculates the starting and ending points in a partition to traverse. $UBound$ and $LBound$ are two functions that compute the upper and lower bounds of the distance between two feature vectors. These two bounds, $D_{ub}$ and $D_{lb}$, are carefully calculated with approximations such that the real distance $D_{real}$ between two corresponding vectors will have the following property:

$$D_{lb} \leq D_{real} \leq D_{ub} \quad (2)$$

Intuitively, $D_{lb}$ is simply the shortest distance among points represented by two approximation cubes. Similarly, $D_{ub}$ is the longest distance. $D_{real}$ is the classical $L_p$ metrics, specifically the Euclidean metric in our case.

For points $p_1(v_{1,1}, v_{1,2}, \ldots, v_{1,d})$ and $p_2(v_{2,1}, v_{2,2}, \ldots, v_{2,d})$, we have

$$D_{real} = \left(\sum_{i=1}^{d} |v_{1,i} - v_{2,i}|^p\right)^{\frac{1}{p}} \quad (3)$$

If the approximations of $p_1$ and $p_2$ are $A_1(a_{1,1}, a_{1,2}, \ldots, a_{1,d})$ and $A_2(a_{2,1}, a_{2,2}, \ldots, a_{2,d})$, respectively, $D_{lb}$ and $D_{ub}$ can be computed as the following:

$$D_{lb} = \left(\sum_{i=1}^{d} (D_{lb,i})^p\right)^{\frac{1}{p}},$$

where $D_{lb,i} = \begin{cases} a_{1,i} - a_{2,i} & \text{if } a_{1,i} > a_{2,i} \\ 0 & \text{if } a_{2,i} = a_{1,i} \\ a_{2,i} - a_{1,i} & \text{if } a_{2,i} > a_{1,i} \end{cases} \quad (4)$

$$D_{ub} = \left(\sum_{i=1}^{d} (D_{ub,i})^p\right)^{\frac{1}{p}},$$

where $D_{ub,i} = \begin{cases} \frac{a_{1,i} - a_{2,i} + 1}{2} & \text{if } a_{1,i} > a_{2,i} \\ \frac{1}{2} & \text{if } a_{2,i} = a_{1,i} \\ \frac{a_{2,i} - a_{1,i} + 1}{2} & \text{if } a_{2,i} > a_{1,i} \end{cases} \quad (5)$

As shown in Formulae $4$ and $5$, $D_{lb}$ and $D_{ub}$ can be derived from two approximation cubes, $A_1(a_{1,1}, \ldots, a_{1,d})$ and $A_2(a_{2,1}, \ldots, a_{2,d})$. Note that, $D_{lb}$ and $D_{ub}$ are indeed the lower and upper bounds for $D_{real}$.

Returning to the algorithm $i$TraverseAC, after reading a pair of approximations, $D_{lb}$ and $D_{ub}$ are computed. If $D_{ub}$ is less than the $r_{radius}$, the real feature vector corresponding to those approximations will be read and output as an answer pair. In the case that $D_{lb}$ is greater than the $r_{radius}$, processing will proceed to the next pair of approximations. Otherwise, the feature vectors will be read and the real distance between them will be computed. If the real distance is less than the $r_{radius}$, they are an answer pair.

Processing approximations can be much faster than processing real feature vectors, depending on the relative size between the approximations and vectors. Most of the candidates can be pruned without being visited. We can further see the effectiveness of this technique from the performance evaluation in Section 4.

3.3. Similarity join with index supporting dimensionality reduction

In the basic $i$Join algorithm, $i$Join, we present the technique to process similarity joins between two datasets organized by iDistance in their metric space. In this section, we propose another enhancement, which exploits dimensionality reduction. The idea of this enhancement is based on two observations.

First, information is not evenly distributed in every dimension of a high-dimensional dataset. A small number of important dimensions may contain most of the information of the data objects. Recently, many researchers [10] have realized this property and have developed some effective techniques to solve various problems. The Principal Component Analysis (PCA) algorithm [14] is one of the methods to perform dimensionality reduction, extracting useful dimensions one by one according to importance.

Second, the sorting technique is a very effective solution to many problems. The result can even be surprisingly good in some cases. In [21], in order to define multiple minimum supports for different categories of frequent sets, sorting acts as the key step to make downward closure property hold, to allow the rest of the data mining process to proceed. In [4], sorting is the first step of processing similarity join in the algorithm. With these two interesting findings, we extend the $i$Join algorithm to support dimensionality reduction. We call it $i$JoinDR.

During the construction of iDistance, the most important dimensions of the dataset $R$ is determined. Similar to the approximation level described in Section 3.2, we introduce a new level which indexes only the important dimensions as shown in Fig. 9. The original iDistance insertion algorithm can be used to construct the index. Once the index is constructed, data points in each partition can be represented by the reduced important dimensions since iDistance uses the $B^-$tree as the underlying index. This property enables us to develop a very efficient join algorithm.

We apply the same main routine as in $i$Join with a new function $i$TraverseDR to traverse the leaf level. In Fig. 10,
the algorithm \(iTraverseDR\) is similar to the algorithm \(iTraverse\) except that here, partial vectors comprised of the important dimensions are used to approximate the data points and work as a filter. \(entry'_r\) and \(entry'_s\) are the approximate forms of \(entry_r\) and \(entry_s\), respectively. If \(\text{Dist}(entry'_r, entry'_s) \leq r_{radius}\), the real feature vectors will be read and output as an answer pair, if the distance between them is less than \(r_{radius}\). The calculation of partial distance is more efficient than that of real feature vectors, depending on the number of important dimensions chosen. Besides efficiency, \(iJoinDR\) also ensures the completeness of the join results, because the distance \(\text{Dist}(entry'_r, entry'_s)\) based on partial vectors is a lower-bound of the real distance between \(entry_r\) and \(entry_s\).

Furthermore, the choice of important dimensions may affect the performance of \(iJoinDR\) as well. Many popular techniques have been proposed to select important dimensions. Principal Component Analysis (PCA) [14], random projection [3] and dimensional extent sorting [5] can perform the task. In our implementation, we use PCA to reduce dimensionality.

PCA is a widely used method for transforming points in original (high-dimensional) space into another (usually lower dimensional) space [10]. It examines the variance structure in the dataset and determines the directions along which the data exhibits high variance. The first principal component (or dimension) accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. Using PCA, most of the information in the original space is condensed into a few dimensions along which

variances in data distribution are the largest. A single example is illustrated in Fig. 11, where a single dimension (the first PC) can capture the variation of data in two-dimensional space after PCA processing. In such cases, it is possible to eliminate some dimensions (the later PCs) with little loss of distance information. PCA is very effective in capturing important information in a dataset, e.g., for the 41-dimensional KDD data from the UCI data repository which we use in our experimental study, 10 important dimensions can capture more than 80% variance of the dataset after PCA processing.

We briefly review how principal components are computed. Let the dataset contain \(N \times d\)-dimensional points. Let \(A\) be the \(N \times d\) data matrix, where each row corresponds to a point in the dataset. We first compute the mean and covariance matrix of the dataset to get the \(eigenmatrix\), \(V\), which is a \(d \times d\) matrix. The first principal component is the eigenvector corresponding to the largest eigenvalue of the variance-covariance matrix of \(A\), the second component corresponds to the eigenmatrix with the second largest eigenvalue and so on. The second step is to transform the data points into new space. This is achieved by multiplying the vectors of each data point with the \(eigenmatrix\).

Formally, a point \(P(x_1, x_2, \ldots, x_D)\) is transformed into \(V \times P = (y_1, y_2, \ldots, y_D)\). To reduce the dimensionality of a dataset to \(k\), \(0 < k < d\), we only need to project the first \(k\) dimensions of the transformed points. The mapping (to reduced dimensionality) corresponds to the well-known Singular Value Decomposition (SVD) of data matrix \(A\) and can be done in \(O(N \cdot d^2)\) time [15].

In summary, our proposed schemes impose no constraint on any property of the applications and datasets it supports. It is adaptive and dynamic, accommodating various \(K\) values. As the experimental results in the performance evaluation section will show, our technique is highly efficient and effective.

4. Performance evaluation

In order to demonstrate the efficiency of \(iJoin\), we implement the proposed algorithms and perform an extensive evaluation using both synthetic and real datasets. The uniform datasets contain 10,000–100,000 randomly
distributed data points and the dimensionality varies of 4, 6, 8 and 16 dimensions. Our real dataset is a 41-dimensional KDD data with over 200,000 records from the UCI data repository [27]. When only one dataset is used, we perform self KNN join. All the experiments are conducted on a Sun Enterprise Server 450 with a Sun Sparc 480 Hz CPU. We compare our KNN join techniques with nested loop join and MuX join [8]. MuX join is the current state-of-art KNN join algorithm. We design the nested loop join algorithm based on the KNN search supported by Omni-Sequential [13], and name the nested loop join Omni-Sequential join. We choose Omni-Sequential join to benchmark our proposed approach as it is the more efficient form of sequential scan. The performance is measured by CPU time and disk I/O access. The page size we use in the experiments is 8 Kbytes.

4.1. Effect of reference points

As we have mentioned, the choice of reference points in iDistance can affect KNN query performance in terms of CPU time and disk access. In the first experiment, we will see the effect of reference point selection on KNN join performance. We use a synthetic dataset consisting of 8-dimensional uniform datasets of size varying from 10,000 to 100,000 records. The default number of nearest neighbors returned for each point is set at 5.

When we build the index structure for join sets, iDistance reference points for two join datasets can be the same regardless of different data distributions. Shared reference points are expected to yield better performance because of the similar partitions in the two datasets. KNN join can be performed partition by partition, and hence matched point pairs can be easily found. However, we can build the index for two join datasets independently. Fig. 12 shows the experimental results with different reference point selections.

It is clear that sharing reference points can provide better performance. Note that in our iJoin algorithm, join operation is based on the node, and each node represents some data space in a partition. If the same reference points are selected for the join sets, it is very likely that one node \( r \) of \( R \) occupies similar space as or overlaps with a node \( s \) in \( S \). Thus for the points in node \( r \), it is more possible to get near-exact KNN candidates after the examination of \( s \), which will help to improve prune efficiency in further node accesses. In contrast, the node in \( R \) typically needs to access more nodes in \( S \) to get the exact KNN answers if the partitions for two datasets are different. Therefore, we select the same reference points for join datasets throughout the performance study.

Number of reference points can affect the KNN query performance of iDistance. In this experiment, we see the effect of number of reference points on KNN join performance. We use 8-dimensional uniform datasets with 50,000 records. The experimental results with varying numbers of reference points are shown in Fig. 13.

The number of reference points (clusters) of iDistance affects search of the area and number of tree traversals from root to leaf nodes in the B+-tree. When the number of clusters is small (<50), more points are likely to have similar distance to the reference points, i.e., more points need to be examined, and hence, higher computational cost is incurred. On the other hand, when the number of clusters is large (>100), more spaces, defined by spheres with respect to centroid of clusters, are likely to overlap and incur additional traversal and search, i.e., more disk I/O accesses. Therefore, optimal performance is a compromise of these factors. However, number of reference points is a tuning parameter, and may vary for different datasets and applications. In the following experiments, we only report the optimal performance for KNN join algorithms.

4.2. Effect of dimensionality

In this experiment, we evaluate the effect of dimensionality on the proposed KNN join algorithms (iJoin, iJoinAC and iJoinDR) by varying the dimensionality of the dataset. The sizes of the databases used for these experiments are 50,000. There are two strategies to simulate KNN join query: pre-constructing indexes and building indexes on the fly. However, we found that index construction cost is almost negligible compared to join cost. Therefore, we assume that the indexes are already preconstructed and

![Fig. 12. Effect of reference point selection.](image)
do not take index construction cost into account in the experiments.

Fig. 14 describes CPU time and number of disk page accesses for the various datasets with dimensionality of 4, 6, 8, and 16. From the results, we can see that the Omni-Sequential join scheme is clearly more costly than other methods. Omni-Sequential join needs to access more data points, and hence, requires more distance calculations between the point pairs. Computational cost is the dominant cost for high-dimensional KNN join. In most cases, the Omni-Sequential join method is more than three times slower than its competitors. MUX performs better than Omni-Sequential join, however it is less effective than the proposed iJoin schemes. The main disadvantage of MUX is that the base index structure, i.e., the R-tree, is not scalable to high-dimensionality. MUX has to suffer the dimensionality curse on R-trees. The performance of iJoin is more than 80% better than that of MUX when the dimensionality is larger than 8. The gap between MUX and the iJoin algorithms widens as dimensionality increases. In another words, the improvement factor of iJoin grows with increasing data space dimensionality. This result reflects, as we anticipated, iJoin can keep superior when dimensionality is higher. It can be also observed in the next experiments using 41-dimensional real data from KDD Archive [27].

Distance computation cost and disk I/O accesses in our techniques are much lower due to the use of iDistance and the various pruning strategies employed. The iJoin algorithms have two good properties. First, iDistance organizes data points such that nearby points are stored in the same partition. When we process the KNN join, we treat each partition as one unit. Hence, we can constrain most of the operations between the corresponding partitions. Second, we apply the same reference point strategy to build the tree structure. It is very likely that the node in R occupies similar space as or overlaps with a node in S, which can significantly improve KNN search efficiency during node accesses.

The enhanced iJoinAC algorithm performs about 20% better than the basic iJoin. Approximation bounding cubes work well when the data is randomly distributed. Approximation cubes are much smaller in size and support faster computation in integer domain. Therefore, processing approximations can be faster than processing real feature vectors, and most of the candidates can be pruned without being visited.

Not surprisingly, the performances of iJoin and iJoinDR are not much different on the uniformly distributed random dataset. The reason is quite obvious. iJoinDR tries to compare partial distance on a subset of dimensions first. For randomly distributed data, all the dimensions have similar variance and there are no “important” dimensions that contain most of the information of the dataset. The number of reduced dimensions in iJoinDR must be large enough in order to filter efficiently. Therefore, in this case, we cannot save much computation cost or disk I/O cost.
For the rest of this paper, we shall only focus on iJoin, iJoinDR, iJoinAC and MUX. We disregard Omni-Seg-quential join as it performs much slower than the other methods.

4.3. Effect of data size

We now study the effect of data size on the various KNN join algorithms. Figs. 15 and 16 summarize the evaluation results for both uniform and real-world datasets. Fig. 15 shows the results for KNN join on 8-dimensional uniform datasets of size varying from 10,000 to 100,000 records. We observe that our techniques perform better as in the first experiment. When the size of the dataset increases, the execution times of these join methods increase, but the cost gap increases with the size of the datasets. The proposed iJoin algorithms are 80% better in most cases. The results show the good scalability of iJoin algorithms with respect to data size. The node-match mechanism and approximation cubes work well even when the dataset is large.

Fig. 16 shows the CPU time and disk access of KNN join in the KDD datasets. The maximum cost speed-up of our techniques as compared to MUX is over a factor of 2. The reason is that the KDD data is high-dimensional. The MUX algorithm inherits the problem of not being scalable in terms of dimensionality as it is based on the R-tree. Most of the MBR keys in the R-tree-like index structure overlap when dimensionality is high. The iJoin algorithms work well because they are based on iDistance, which was originally proposed to deal with high-di-men-sional data. iJoinDR is about 30% better than iJoinAC for KDD dataset. Real datasets are generally skewed, thus a small subset of dimensions is sufficient to capture most of the variation in the dataset. Distance computation can be faster and fewer disk I/Os are incurred because of the reduced dimensionality. For iJoinAC, more points may have similar approximation cubes when the data is skewed. To filter false KNN candidates efficiently, we need to use more bits to represent the approximation cubes for the skewed dataset. Hence, the benefit of approximation cubes is reduced. Although iJoinAC still performs better than iJoin, the gap is narrowed.

4.4. Effect of K

In this experiment, we study the effect of number of nearest neighbors K. Figs. 17 and 18 show the performance results of KNN join with varying K from 5 to 20 for different datasets. This set of experiments run on an 8-dimensional uniform dataset with 50,000 records and a KDD dataset with 200,000 records. In our experiments, we notice MUX cannot prune point pairs effectively as the number of neighbors K increases, while iJoin can organize data points such that nearby points are stored near each other. As a result, shown in the figures, the CPU cost of MUX
increases by a large amount while the iJoin schemes remain almost constant in terms of costs. Hence, the cost of our techniques is not heavily affected by $K$. Among all the iJoin schemes, $iJoinAC$ performs best for the uniformly distributed dataset while $iJoinDR$ performs best for the KDD dataset.

4.5. Effect of relative size of two join datasets

In this experiment, we join two datasets of different sizes and study how the size ratio of two datasets affects the performance of the join operations. The KDD dataset is used for this experiment. We have the join set $R$ with 200,000 records as the outer set, and vary the size of the inner set $S$ from 20,000 to 160,000, so that the relative size of $R:S$ is changed from 0.1 to 0.8. Fig. 19 shows the CPU cost and I/O cost for different relative size of two join sets. The CPU time speed-up of KNN join with our techniques as compared to MUX is over 2 in most cases. $iJoinDR$ is the most efficient compared with the other two mechanisms. As we have mentioned, as the KDD dataset is skewed, a small subset of dimensions can capture most of the variation in the dataset; hence, overall cost is reduced.

4.6. Summary of experiments

We have conducted extensive experiments to evaluate the performance of iJoin and its two variations. Table 1 lists the experiments we included in this paper.
Table 1
List of Experiments

<table>
<thead>
<tr>
<th>Experiment 1: Effect of reference points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Techniques:</td>
</tr>
<tr>
<td>iJoin</td>
</tr>
<tr>
<td>iJoinAC</td>
</tr>
<tr>
<td>iJoinDR</td>
</tr>
<tr>
<td>Mux</td>
</tr>
<tr>
<td>Omni-sequential</td>
</tr>
<tr>
<td>Comparison:</td>
</tr>
<tr>
<td>Shared reference points vs. independent reference points; Different number of reference points</td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td>Datasets:</td>
</tr>
<tr>
<td>10,000, 20,000, 50,000, 100,000 (8-dimensional uniform datasets); 200,000 (real data)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment 2: Effect of Dimensionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Techniques:</td>
</tr>
<tr>
<td>iJoin</td>
</tr>
<tr>
<td>iJoinAC</td>
</tr>
<tr>
<td>iJoinDR</td>
</tr>
<tr>
<td>Mux</td>
</tr>
<tr>
<td>Omni-sequential</td>
</tr>
<tr>
<td>Comparison:</td>
</tr>
<tr>
<td>Various dimensionality</td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td>Dataset size:</td>
</tr>
<tr>
<td>50,000 (uniform data) and 200,000 (real data)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment 3: Effect of Dataset size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Techniques:</td>
</tr>
<tr>
<td>iJoin</td>
</tr>
<tr>
<td>iJoinAC</td>
</tr>
<tr>
<td>iJoinDR</td>
</tr>
<tr>
<td>Mux</td>
</tr>
<tr>
<td>Comparison:</td>
</tr>
<tr>
<td>Various dataset sizes</td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td>Dataset size:</td>
</tr>
<tr>
<td>10,000, 20,000, 50,000, 100,000 (8-dimensional uniform data); 200,000 (41-dimensional real data)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment 4: Effect of K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Techniques:</td>
</tr>
<tr>
<td>iJoin</td>
</tr>
<tr>
<td>iJoinAC</td>
</tr>
<tr>
<td>iJoinDR</td>
</tr>
<tr>
<td>Mux</td>
</tr>
<tr>
<td>Comparison:</td>
</tr>
<tr>
<td>Various number of nearest neighbors to be combined</td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td>Dataset size:</td>
</tr>
<tr>
<td>50,000 8-dimensional uniform data; 200,000 41-dimensional real data; K = 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment 5: Relative dataset sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Techniques:</td>
</tr>
<tr>
<td>iJoin</td>
</tr>
<tr>
<td>iJoinAC</td>
</tr>
<tr>
<td>iJoinDR</td>
</tr>
<tr>
<td>Mux</td>
</tr>
<tr>
<td>Comparison:</td>
</tr>
<tr>
<td>Different relative sizes of two join datasets</td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td>Outer dataset:</td>
</tr>
<tr>
<td>R: 200,000 41-dimensional real data; Inner dataset: S: 20,000, 40,000, 100,000, 160,000 41-dimensional real data; K = 5</td>
</tr>
</tbody>
</table>

iJoin is adaptable and efficient because it exploits and benefits from iDistance’s flexible reference points selection and partition strategy. In Experiment 1, it shows that using the same (or similar) reference points for two join data sets can improve the efficiency of iJoin. Experiment 2 proves that iJoin is more efficient than MUX and Omni-Sequential join. More meaningfully, the performance gap widens with the increasing dimensionality. The result of Experiment 3 can further confirm that iJoin is more scalable to dimensionality. Experiments 3, 4 and 5 show that iJoin has much better scalability than MUX, in terms of the dataset sizes, the number of NNs that need to be found for each combination, and the increase of inner join dataset size. In addition, we can also see that iJoinAC and iJoinDR can mostly further improve the performance of iJoin.

iJoin can outperform the current up-to-date MUX technique, because iDistance used in iJoin is much more efficient than R-tree used in MUX, for high-dimensional data. Moreover, the flexible partition strategy of iDistance makes it possible to control the data partitioning in large granularity. With sharing reference points for both join data sets, iJoin can quickly bring the nodes with close distances together for join. However, the data regions of the R-tree nodes are hardly controllable. For the same reason, MUX needs big buffer space to support its join operation. However, the performance of iJoin does not rely much on buffering. It is certainly an additional advantage of iJoin, since the volume of buffered data is usually limited.

5. Conclusion

Many applications are based on the similarity joins of very large datasets. In this paper, we have investigated the problem of computing KNN similarity joins between pairs of high-dimensional point datasets and described several algorithmic approaches that can be applied to KNN joins. We have proposed an efficient KNN join approach using iDistance as the underlying index structure. Three algorithms are presented in this paper, including the basic iJoin and its two variations with different enhancements: iJoinAC and iJoinDR. iJoinAC makes use of approximation cubes to reduce unnecessary KNN computation on real feature vectors. iJoinDR indexes reduced dimensions to decrease disk I/O access and CPU cost. These strategies exploit the actual data distribution of input relations and can easily be integrated into many existing databases. We have presented the detailed study of the algorithms, and showed experimentally that our approach is a promising solution to the high-dimensional KNN join problem. In this paper, all our techniques assume the single processor computer architecture. For future work, we are particularly interested in a more detailed study of similarity KNN joins in different architectures, such as similarity KNN join on parallel and distributed computer systems.

References


