LDC: Enabling Search By Partial Distance In A Hyper-Dimensional Space

Nick Koudas[1] Beng Chin Ooi[2] Heng Tao Shen[2] Anthony K. H. Tung[2]

[1] AT&T Labs Research
Shannon Laboratory Florham Park, USA

[2] Department of Computer Science
National University of Singapore, Singapore

Abstract

Recent advances in research fields like multimedia and bioinformatics have brought about a new generation of **hyper-dimensional** *databases which can contain hundreds or even thousands of dimensions. Such hyper-dimensional databases pose significant problems to existing high-dimensional indexing techniques which have been developed for indexing databases with (commonly) less than a hundred dimensions. To support efficient querying and retrieval on hyper-dimensional databases, we propose a methodology called Local Digital Coding (LDC) which can support k-nearest neighbors (KNN) queries on hyper-dimensional databases and yet co-exist with ubiquitous indices, such as B^+-trees. LDC extracts a simple bitmap representation called Digital Code(DC) for each point in the database. Pruning during KNN search is performed by dynamically selecting only a subset of the bits from the DC based on which subsequent comparisons are performed. In doing so, expensive operations involved in computing L-norm distance functions between hyper-dimensional data can be avoided. Extensive experiments are conducted to show that our methodology offers significant performance advantages over other existing indexing methods on both real life and synthetic hyper-dimensional datasets.*

1. Introduction

Hyper-dimensional[1] databases are databases which contain hundreds or even thousands of dimensions. Recent advances in several research fields including multimedia, bioinformatics, data mining on audio, images and text, as well as networking, have resulted in such databases which pose significant challenges to existing high-dimensional indexing techniques, that are usually capable of handling databases (commonly) up to tens of dimensions.

The problem of indexing and searching in a hyper- dimensional database is a challenging one, due to three main reasons:

- First, according to several studies (e.g., [3]), the expected minimal distance between any two points in a hyper-dimensional space is very large (becoming larger with increasing dimensionality) while the difference between the minimal and maximal distance to a point is expected to be small (becoming smaller with increasing dimensionality). These two characteristics of a hyper-dimensional space mean that the search radius for a k-nearest neighbor query is expected to be large. This in turn results in a large number of 'false positives' since most points are expected to have almost equal distance to the query point. This phenomenon leads to significant deterioration of the query performance in most existing indexing methods.
- Second, due to the extremely high dimensionality, the fanout for most indexes built on a hyper-dimensional space is typically very small, resulting in an increase in the height of the indexes (e.g., in a 200 dimensional space, we can't expect more that ten entries in an 8K page if 4 bytes are needed for each dimension).
- Finally, the computation of the distance (e.g., Euclidean distance) between two points in a hyper-dimensional space, becomes processor intensive as the dimensionality increases. This implies that the processor time is expected to become a significant portion of the overall query response time for a hyper-dimensional database. Proposed techniques for optimizing the performance of most indexing techniques do not take this into consideration.

In this paper, we propose an effective methodology called Local Digital Coding (LDC) for finding k-nearest neighbors in a hyper-dimensional space. LDC is developed to address the problems mentioned above and provide a substantial reduction on both I/O and processor time when searching on hyper-dimensional datasets consisting of hundreds of dimensions. It is compatible with ubiquitous indices, such as B^+-trees and thus can be easily deployed.

Given a cluster of points in a high-dimensional data space LDC transforms each point into a bitmap which we refer to as the point's Digital Code (DC). Each dimension of the point is represented by a single bit in its DC. The DC of a point is generated by comparing the coordinates of the point with the coordinates of the cluster center the point belongs to. A bit is set to 1, if the value of the dimension it corresponds to, is larger than the value of the corresponding dimension of the cluster center, and 0 otherwise.

1 The term hyper-dimensional is used to differentiate the problem we are addressing from the present norm of 30- to 50- (high) dimensional space

Since there is a bit in the DC for each dimension, indexing a D-dimensional space will result in DCs with D bits. The data points in a cluster can thus be separated into 2^D partitions with points in each partition sharing the same DC. Based on LDC, we propose a novel searching algorithm, called **Searching on-the-fly by PArtial-distance** (SPA). Given the DCs of both the query point and a partition, SPA **dynamically** selects a **subset** from the DCs (say n bits) to perform matching. A partition is pruned off if the number of matching bits in the two DCs is less than m bits. The intuition behind such an approach is that the points in the pruned partition are on different sides of some cutting planes with respect to the query point and thus are too far away to be in the answer set.

A summary of our contributions is as follows:

- We present LDC, an efficient methodology for indexing a hyper-dimensional space. LDC has the following advantages over existing methods for handling hyper-dimensional data:
  - LDC is optimized to take both processor and I/O time into account. This is important since processor time for comparing vectors becomes significant in a hyper-dimensional space.
  - LDC's method of spatial inference for pruning the search space is based on cutting-planes instead of direct distance computation. Because of this, the proposed algorithm, SPA, that performs KNN search, has significantly less 'false positives' compared to methods using direct distance computation.
  - LDC adopts a bit representation instead of storing the actual dimensional values, thus, its storage requirements are the smallest possible (without considering the option of using compression methods) for representing the location of any D-dimensional partition. This effectively reduces the number of I/Os that are required to search the index.
- We identify the parameters that affect the performance of LDC and present the results of a detailed analysis identifying how these parameters can be tuned for certain objectives given a query point.
- We present the results of a detailed experimental evaluation of our methodology supporting our claims that LDC is capable of handling hyper-dimensional databases more efficiently, compared to other applicable methods.

The rest of the paper is organized as follows. Section 2 reviews related work. LDC is presented in Section 3. In Section 4, the novel SPA algorithm is introduced. An extensive performance study on our method is presented in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

Indexing techniques have been the focus of extensive research both in low as well as high-dimensional databases [4]. With the demand for even higher-dimensional databases, consisting of hundreds of or more dimensions, earlier high-dimensional indexes face significant challenges. Indexing techniques have been designed typically for 30-50 dimensions, and fail to improve the performance of sequential scan [11] due to the known 'dimensionality curse'. To tackle this phenomenon recent proposals adopt one of the three approaches: (1) Dimensionality reduction, (2) Data approximation, and (3) One dimensional Transformation.

Dimensionality reduction methods [9, 6] map the high dimensional space into a low dimensional space which can be indexed efficiently using existing multi-dimensional indexing techniques. The main idea is to condense the original space into a few dimensions along which the information is maximized. Such methods report approximate nearest neighbors however, since dimensionality reduction incurs information loss.

Representations of the original data points using smaller, approximate representations have also been proposed, as a means of aiding high dimensional indexing and searching. Such proposals include, the VA-file [11], the IQ-tree [1] and the A-tree [10]. The VA-file (Vector Approximation file) represents the original data points by much smaller vectors. The main drawback of the VA-file however, is that it defaults in assessing the full distance between the approximate vectors, which imposes a significant overhead, especially if the underlying dimensionality is very large. Moreover, the VA-file does not adapt gracefully to highly skewed data. The IQ-tree was proposed recently. It maintains a flat directory which contains the minimum bounding rectangles of the approximate data representations.

One dimensional transformations provide another direction for high-dimensional indexing. Such techniques include the Pyramid technique[2] and iDistance[12]. They suffer however, from the fact that any meaningful search operation involves assessing distances between the full high dimensional representation of the data points; thus, pruning during search becomes problematic as the dimensionality increases. High dimensional search techniques, assuming specific storage models, have been recently proposed in [7].

While LDC exploits bit representation, it is fundamentally different from the VA-file in three ways: 1) LDC uses one bit for each dimension, while the VA-file uses multiple bits. 2) Pruning in LDC is achieved dynamically by selecting a subset of DC for bitwise operation, while the VA-file involves full distance computation between approximation vectors. 3) LDC is adaptive to skewed data distributions while the VA-file has been shown to be effective mainly for uniform datasets [10]. This is further confirmed in our experiments.

3. Local Digital Coding (LDC)

In this section, we start by presenting the LDC methodology. Since LDC can easily co-exist with B^+-tree indices, we refer to a B^+-tree employing LDC as an *LDC tree*. Then, we will discuss how to efficiently construct an LDC tree.

3.1. Structure of the LDC tree

The structure of the LDC tree can be essentially separated into three levels. Figure 1 presents an example. The

first level of the LDC tree is a B^+-tree which is formed by simply transforming each point in the hyper-dimensional space into a one-dimensional value. Several methods can be utilized to perform this transformation; we choose the method proposed in [12], due to its simplicity. In [12], the whole data space is first grouped into clusters with each point in the data space belonging to one of the clusters.

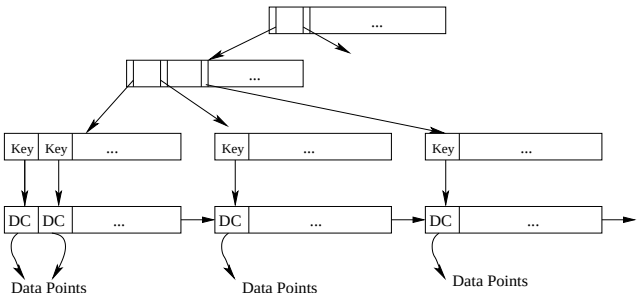


Figure 1. The overall structure of an LDC tree.

Assuming each cluster is assigned a unique ID, the one-dimensional value for a data point P in cluster i is computed as:

$$key(P) = i * c + d(P, O_i)$$

where O_i is the center of cluster i, c a constant and $d(.)$ a distance function, assessing the distance between two points in the hyper- dimensional space. The minimum and maximum value of $Key()$ among all points in a cluster, define a range of values, which we refer to as the *range* of cluster i. Constant c, which we refer to as the *scaling constant* is chosen as to increase the range of each cluster, so that the ranges of different clusters are disjoint and can thus be distinguished easily. This way, the range of each cluster i, is $[i * c, (i + 1) * c]$. An additional auxiliary array is used to store the center and range for each cluster. Given a D-dimensional point P, the cluster with center O, that P belongs to, can be determined by computing $Key(P)$ and checking membership of the value $Key(P)$ in a cluster's range.

The second level of the LDC tree is the Digital Code (DC) level. The basic idea of a DC is to produce an effective single bit representation for each dimension of a data point. Given a D-dimensional data space, we encode each data point into a D-dimensional DC representation by applying the algorithm in Figure 2. Given a D-dimensional point P and a cluster center O, initially, the DC is set to 0 (line 1). For each dimension, if its value is equal or greater than the value of the corresponding dimension of the cluster center, we set the corresponding bit of the DC representing that dimension to 1 and 0 otherwise (lines 3 and 4). The DCs are stored separately from the index in order to enable fast sequential scanning of DCs in the SPA algorithm.

The third level of the tree is the data level where the actual data points are stored.

3.2. Constructing the LDC Tree

The LDC tree is built in three steps. In the first step, the data space is partitioned into clusters. A plethora of

Local Digital Coding Algorithm
LDC(P,O)
Input: P, O;
Output: DC;
1. unsigned int DC=0;
2. for i=0 to D-1
3. if (P[i] $\geq$ O[i])
4. DC $|= 1 <<$ i;
5. return DC;

Figure 2. Local Digital Coding

proposals exist for clustering high dimensional data sets [8, 13]. For hyper-dimensional datasets, this process can be computationally expensive even as a pre-processing step. A variety of approaches exist for clustering high dimensional datasets approximately, trading accuracy for speed [5]; such techniques can be readily utilized in our setting, significantly improving the performance of this preprocessing step.

For simplicity we adopt the following approach for partitioning the data set into clusters. We first select the edge points as done in [12] and then group the data points into clusters by assigning them to the nearest edge point. While doing so, the centroid of each cluster can be approximated. This is done by estimating the median of the cluster on each dimension through the construction of a histogram along each dimension. The centroid of each cluster is then used as the cluster center.

The $Key()$ value for each point (relative to its cluster center) is then computed. A single B^+-tree index is then built on these values. The second step links each distance value in the leaf nodes of the B^+-tree to the corresponding point's DC representation in order. This way the DC level of LDC is populated. Pages containing DC's are sequentially stored so that a sequential scan on DCs can be performed.

Finally, the third step links each DC to the corresponding data point in order. This way, for each point in the data space its corresponding DC value is under its corresponding $Key()$ value in the leaf nodes of the B^+-tree, and above the actual coordinates of the point.

Dynamic maintenance operations arising from insertions/deletions on an LDC Tree can be easily performed. When a new point is added in the dataset, its cluster membership is first determined by identifying the nearest cluster center. Then its one-dimensional distance and DC representation is computed, followed by the standard insertion operation in B^+-trees. Correspondingly, the point's linked DC and data are inserted into the correct position in the DC level and data level of the tree respectively. Deletion operations are quite similar. After the point's one-dimensional distance value is computed, the standard deletion operation in a B^+-tree is then performed, followed by removing the corresponding linked DC and data point.

4. KNN Search in an LDC Tree

We now turn our attention on hyper-dimensional searching and show how a k-nearest neighbor (KNN) search can be effectively performed on the LDC tree.

Notation	Description
Q, P, O	Query, data point, cluster center
$Q[i]$, $P[i]$, $O[i]$	The i^{th} dimension value for Q,P,O
DC_Q, DC_P	The DC for Q and P
$DC_Q[i]$, $DC_P[i]$	The corresponding bit for the i^{th} dimension in the DC of Q and P
$rank$	A dimension ranking array
$rank[i]$	The i^{th} element of the dimension ranking array
D	Dimensionality of the database
DIM	The set of dimensions for the database, $\|DIM\| = D$
DIM'	A proper subset of DIM
s, t	Number of dimensions
m	Pruning length
n	Candidate pruning length
pd	Partial distance
Θ	Threshold for search space or $\frac{n}{m}$
Φ	Maximal number of candidates allowed

Table 1. A Table of Notations.

Without loss of generality and to simplify our presentation we will assume that the distance function adopted in the search is the Euclidean distance. Thus, in the remainder of the paper $d(.)$ refers to the Euclidean distance between a pair of points. We summarize the notation used in Table 1 for quick reference. We first present the theoretical foundation of our algorithm.

4.1. Partial Distance

We begin by formally defining *partial distance*:

Definition 4.1 (Partial Distance $pd(Q, P, DIM')$) *Let Q and P be two points in a D-dimensional space and let $Q[i]$, $P[i]$ denote the values of dimension i in Q and P respectively. We denote the set of D dimensions as DIM. Given $DIM' \subset DIM$, the partial distance between Q and P is defined as*

$$pd(Q, P, DIM') = \sqrt{\sum_{i \in DIM'} (Q[i] - P[i])^2}$$

Thus, the partial distance between two D-dimensional points P and Q is in fact the Euclidean distance computed on a subset of the D dimensions. In what follows, when the parameters of partial distance are clearly implied by the context, we will denote partial distance simply using pd. We thus have the following corollary:

Corollary 4.1 Let $d(Q, P)$ denote the Euclidean distance between Q and P, then $pd(Q, P, DIM') \leq d(Q, P)$.

Next, we derive a lower bound on the distance of a query point Q and a data point P, from their corresponding DC representations:

Theorem 4.1 *Let Q be a query point and P be a data point. We use DC_Q and DC_P to denote the DCs that are computed with respect to a cluster center O. In addition, we use $DC_Q[i]$ and $DC_P[i]$ to denote the corresponding bit in dimension i in both DCs. Let DIM' be a set of dimensions such that $\forall i \in DIM', DC_Q[i] \neq DC_P[i]$, then*

$$d(P, Q) \geq \sqrt{\sum_{i \in DIM'} (Q[i] - O[i])^2}$$

Proof:

Since DC_Q and DC_P differ on the i^{th} bit, this implies that Q and P are located on different sides of a plane that is parallel to the i^th dimension and passes through O. We can thus infer that $|Q[i]-O[i]| \leq |Q[i]-P[i]|$ for all $i \in DIM'$. Thus, we conclude that

$$\sqrt{\sum_{i \in DIM'} (Q[i] - O[i])^2} \leq pd(Q, P, DIM')$$

Corollary 4.1 completes the proof.

We now provide an **upper bound on the number of mismatches** that can occur between DC_Q and DC_P (computed with respect to O) if the distance between Q and P is less than a pruning distance - pd. To do so, we first introduce the notion of a **dimension ranking array**.

Definition 4.2 (Dimension Ranking Array) *Let Q be a query point and O be a cluster center. The* **dimension ranking array** *for Q and O is an array, $rank$, of D positions such that:*

1. *Each dimension in the data space is uniquely represented by one element $rank[i]$ in the array.*
2. *$(Q[rank[0]] - O[rank[0]]) \geq Q[rank[1]] - O[rank[1]]) \geq ..... \geq Q[rank[D-1]] - O[rank[D-1]])$*

Thus, each element $rank[i]$ in the dimension ranking array for O and Q is the dimension in which $|Q[rank[i]] - O[rank[i]]|$ is the i^{th} largest among all the D dimensions.

Theorem 4.2 *Let DC_Q and DC_P be the DCs of two D-dimensional points which are computed with respect to a cluster center O. Let $rank$ be the dimension ranking array for Q and O. Given a pruning distance pd, we use s to denote the largest number such that*

$$\sqrt{\sum_{i=D-s}^{D-1} (Q[rank[i]] - O[rank[i]])^2} \leq pd$$

If $d(P, Q) \leq pd$, then the number of bit mismatches between DC_Q and DC_P cannot be more than s.

Proof:

Assume DC_Q and DC_P have t mismatches, $t \geq s$ and let DIM' represent the set of dimensions in which the t mismatches between DC_Q and DC_P occurred. According to Theorem 4.1,

$$d(P, Q) \geq \sqrt{\sum_{i \in DIM'} (Q[i] - O[i])^2}$$

To minimize the R.H.S of the inequality, we should pick DIM' to be the set of t dimensions in which $|Q[i]-O[i]|$ are

the smallest i.e. the dimensions represented by $rank[D-t]$ *to* $rank[D-1]$. *Since* s *is the largest value such that*

$$\sqrt{\sum_{i=D-s}^{D-1} (Q[rank[i]] - O[rank[i]])^2} \leq pd$$

and $t > s$, *we can thus conclude that*

$$\sqrt{\sum_{i=D-t}^{D-1} (Q[rank[i]] - O[rank[i]])^2} > pd$$

By combining the inequalities, we will have $d(P,Q) > pd$.

Figure 3 illustrates the relationship between D, s and t on the ranking array. Notice that given a pruning distance pd, Theorem 4.2 enables pruning of candidate data points. Given a query Q, a cluster center O and a pruning distance pd, one can derive the suitable value of s according to Theorem 4.2 and all DCs, with more than s mismatches to the query DC, can be pruned away. Such a strategy for pruning has two main drawbacks: (a) it involves performing bitwise operations on vectors of D bits, for a possibly very large D and (b) it provides no way to control the amount of actual pruning performed.

To overcome these drawbacks, we will compare the DCs on *only those bits that correspond to the top (largest)* n *dimensions* in the dimension ranking array. Comparing on only $n < D$ dimensions effectively reduces the number of bits examined, offering additional savings in processor time. Assuming n is sufficiently large, we can still determine a value m based on Theorem 4.2, such that having more than m mismatches among the n bits will mean that $d(P,Q) > pd$. Comparing on the n largest dimensions, for a specific value of pd, assures that n is as small as possible. Moreover, as we will see, it enables a flexible strategy that allows the user to control the amount of pruning performed.

We note that selecting a good value for n can be difficult. A large value of n will not bring substantial savings in processor time. Even more importantly, a small value of n could result in a low probability of pruning off candidate data points. For example, in the extreme case where $n = 1$ and $m = 0$, only 1 bit of the DCs is compared and there is only a 50% chance of pruning off a candidate DC if we assume a uniform distribution of the data points. In the next section, we will illustrate the relationship between the probability of pruning a partition and the selection of n and m.

4.2. Selecting the values m and n

In our analysis on the relationship between n and m, we shall assume that data are uniformly distributed in the hyper-dimensional space.

Lemma 4.1 *A* D*-dimensional DC representation divides the portion of the hyper-dimensional space spanned by the points belonging to a cluster into* 2^D *partitions.*

Proof: *Since the* i^{th} *bit of a DC represents the two sides of a plane that is orthogonal to the* i^{th} *dimension and passes through a cluster center* O, *having* D *bits indicates that there are* D *orthogonal planes passing through* O. *Each of these mutually independent planes divides the original data space into half which gives us* 2^D *partitions.*

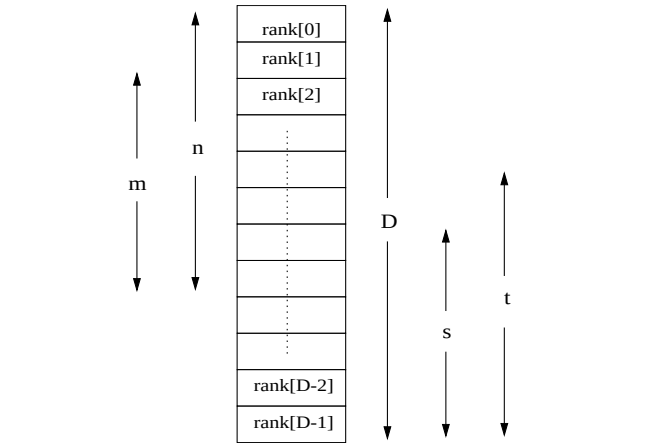


Figure 3. Dimensions Ranking Array.

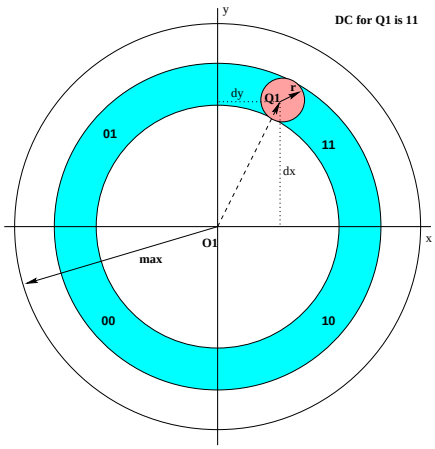


Figure 4. Searching space in a 2-d space

Figure 4 shows an example in 2-dimensional space. It is evident that the cluster space is divided into four partitions by two orthogonal planes. The partitions are identified by their DCs as 11, 10, 01, and 00 respectively.

Lemma 4.2 *Let* DC_Q *be a query DC and* DIM' *be a subset of any* n *dimensions. If comparisons are made only on the* n *bits of the DCs that correspond to* DIM', *then the average number of partitions which share exactly* m *common bits with* DC_Q *is*

$$\binom{n}{m} * 2^{(D-n)}$$

Proof: *We first notice that each axis-aligned hyperplane, in expectation, will partition the points in the cluster into half. Hence, each bit in a data point's DC has equal probability of matching or not matching the corresponding bit of* DC_Q. *The probability that a DC has exactly* m *common bits with* DC_Q *is thus* $\binom{n}{m} * (\frac{1}{2})^m * (\frac{1}{2})^{n-m} = \binom{n}{m} * \frac{1}{2^n}$. *Since we consider only* n *bits, there will be* 2^n *different combinations of the* n *bits. The number of DCs associated with*

each of these combination will be $2^{(D-n)}$. *Thus, the average number of DCs with exactly* m *common bits with the query DC is* $\begin{pmatrix} n \\ m \end{pmatrix} * \frac{1}{2^n} * 2^n * 2^{(D-n)} = \begin{pmatrix} n \\ m \end{pmatrix} * 2^{(D-n)}$.

From Lemma 4.2, the have the following corollary:

Corollary 4.2 *When comparing with a query DC on a subset of* n *bits, the average number of DCs which have no more than* m *mismatches with the query DC is:*

$$\sum_{k=n-m}^{n} \begin{pmatrix} n \\ k \end{pmatrix} * 2^{(D-n)}$$

or equivalently

$$\sum_{k=0}^{m} \begin{pmatrix} n \\ k \end{pmatrix} * 2^{(D-n)}$$

The next theorem provides an estimate on the average percentage of partitions that are accessed when pruning is performed based on a subset of n bits from the query DC.

Theorem 4.3 *If a bitmap of length* n, *extracted from the query DC, is compared with the partitions of a cluster and partitions are pruned if there are more than* m *mismatches between the query and partition DCs, then the percentage of partitions that are searched is:*

$$Prob(n,m) = \frac{\sum_{k=0}^{m} \begin{pmatrix} n \\ k \end{pmatrix}}{2^n}$$

Proof: *From Corollary 4.2, the number of partition DCs having no more than* m *mismatches with the query DC has an expected value of* $\sum_{k=0}^{m} \begin{pmatrix} n \\ k \end{pmatrix} * 2^{(D-n)}$. *Since the total number of partitions is* 2^D, *the percentage of partitions being searched will be* $\sum_{k=0}^{m} \begin{pmatrix} n \\ k \end{pmatrix} * 2^{(D-n)}/2^D =$

$$Prob(n,m) = \frac{\sum_{k=0}^{m} \begin{pmatrix} n \\ k \end{pmatrix}}{2^n}$$

The expression for $Prob(n,m)$ indicates that increasing n or reducing m results in a reduction of the search space. Hence by adjusting the values of n and m, we can control the number of candidates. We refer to such a value of m as the *pruning length* and n as the *candidate pruning length*.

The ideal case in a KNN search is to maximize pd and minimize $Prob(n,m)$. When pd is larger, the KNN can be discovered earlier, since a larger pruning distance will return more neighbors (hopefully K). However, when $Prob(n,m)$ is too small (a small m and a large n will result in a very small partial distance), a small fraction of a cluster is searched and thus, too few candidates are returned. In this case, there is a danger to filter out the KNN and further search has to be performed. So there is a trade-off between $Prob(n,m)$ and pd.

From Theorem 4.2, given specific values for n and pd, we derive that m is the *largest* number such that

$$\sqrt{\sum_{i=n-m}^{n-1} \left(Q[rank[i]] - O[rank[i]]\right)^2} \leq pd$$

The above formula indicates the relationship among n, m and pd. Given m and pd, the n will be the *largest* number (to minimize $Prob(n,m)$) which satisfies the following *partial distance constraint*:

$$\sqrt{\sum_{i=n-m}^{n-1} \left(Q[rank[i]] - O[rank[i]]\right)^2} \geq pd$$

We refer to this formula as the *partial distance constraint*. Theorems 4.2, 4.3, and the *partial distance constraint* pave the way for the search by partial distance algorithm - SPA.

4.3. The KNN Search Algorithm

Having established the relationship among m, n, and the partial distance, we are now ready to present the KNN search algorithm.

KNN Algorithm in LDC

```
Input: Q - Query point
Output: knn[] - k-nearest neighbors to Q
Variables:
    O[N]: Array of N cluster centers
    rank[][]: Rank array for each cluster center
    QDC[][] : Query DC for each cluster center
    Δr : Step for adjusting search radius r
    ΔΘ : Step for adjusting search space
1.  for i=0 to N-1
2.      rank[i] ← rank_dimension(Q,O[i]);
3.      QDC[i] ← LDC(Q,O[i]);
4.  set r, Θ, Φ;
5.  do //start searching
6.      candidates[] ←
                SPA(Q,QDC[],rank[],r,O[],Θ,pre_r,pre_pd[]);
7.      if candidates.size > Φ
8.          Θ = Θ - ΔΘ;
9.          continue;
10.     if candidates.size < K
11.         Θ = Θ + ΔΘ;
12.         continue;
13.     knn[] ← compute_distance(candidates[],Q);
14.     r = r + Δr;
15. until the K^th-NN is found
```

Figure 5. Main KNN Search Algorithm in LDC

Figure 5 shows the main routine for the KNN search in the LDC tree, given the query point Q. At the start of the search, the dimension ranking array is first filled based on relative position of the query point to the cluster centers in O (line 2). Relative to each cluster center, a DC for the query point can be computed based on the LDC algorithm in Figure 2 (line 3).

The search process is iterative in nature. It consists of incrementally adjusting the search radius (by Δr), until all the KNNs are discovered (line 5-15). The incremental search process is controlled by three parameters, r the

search radius, Θ, the desired *maximal fraction of space to search* and Φ, the desired *maximal number of candidates*. The initial values of these parameters are set in line 4.

Conventionally, we determine the initial value of r by the difference between the index key of the query point and index key of its nearest point in the same leaf node, where the query point possibly resides. The parameter, Θ, indicates the desired maximal fraction of space to be searched within the whole data space. It will be adjusted to control the number of candidates being considered as searching process goes on. Its initial value can be estimated based on sampling or from past historical records. While such an approach will work for uniformly distributed data, things are less straightforward if the data distribution varies in different part of the data space. In such situation, the number of candidates cannot be accurately estimated based on Θ. Hence having the same Θ value may correspond to largely different number of candidates for different queries. To ensure retrieval efficiency, small numbers of candidates are always desired. Φ, the desired maximum number of candidates, is used to adjust Θ to avoid overflow by candidates. We defer the discussion on Δr and Φ to Section 4.5.

The SPA method (line 6), utilizes partial distances to rapidly prune the candidates to a very small set by searching on-the-fly, employing $Prob(n, m)$. In particular, the threshold Θ and the search radius r are used to compute the suitable pair of (n,m) such that the number of candidates being considered is less than the desired fraction Θ of the points in the database. The parameter *pre_r* is used to detect if the search radius has been increased from previous SPA call. And the parameter *pre_pd*[] remembers the previous pruning distances to avoid the same computation as previous SPA call did. Both are initialized to 0 before the search starts.

In the KNN algorithm, the parameter, Θ can be adjusted (line 7-12) if the number of candidates returned by SPA is larger than Φ. By reducing Θ, fewer candidates will be produced in the next round of search (line 8). On the other hand, if the number of candidates is smaller than K, Θ will be increased to include more candidates (line 11). $\Delta\Theta$ can be set based on the volume of candidates returned from SPA. Higher volume corresponds to a larger $\Delta\Theta$, and vice versa.

If the K^{th}-NN's distance is equal to or smaller than the $min(pre\_pd[i])$, all KNNs are found (line 15). Notice that the minimal value among all *pre_pd*[i], instead of r, is the final pruning distance.

Now we proceed to the core of the search algorithm - the Search on-the-fly by PArtial-distance (SPA) algorithm. SPA consists of three steps: allocate the DCs, generate suitable values of m and n, and prune data by DC comparison. Figure 6 describes the algorithm.

Given a search radius r and a query Q, SPA first checks if the searching space intersects with a cluster range. For each cluster center i, O[i].min and O[i].max indicate the min and max values in the cluster range. If they intersect, SPA allocates two key values nearest to the values of $max(i * c - dist - r, i * c + O[i].min)$ and $min(i * c + dist + r, i * c + O[i].max)$ respectively in the cluster space (line 6-7), followed by allocating their corresponding DCs (line 8-9). Since all DCs at the DC level of the LDC tree are linked to each other, DCs within a range can be sequentially scanned and retrieved for DC comparison thus avoiding random I/O access.

SPA Algorithm

```
Input: Q, QDC[], rank[], r, O[], Θ, pre_r, pre_pd[];
Output: candidates[];
1. for i=0 to N-1
2.    dist = compute_distance(O[i], Q);
3.    if O[i].min > dist+r and O[i].max+r < dist
4.        continue;
5.    // allocate the starting and ending DCs
6.    l_leaf ← alloc_leaf(btree,max(i*c-dist-r,i*c+O[i].min));
7.    r_leaf ← alloc_leaf(btree,min( i*c+dist+r,i*c+O[i].max));
8.    l_DC ← alloc_DC(l_leaf);
9.    r_DC ← alloc_DC(r_leaf);
10.   // generate m and n given a pd as pruning distance
11.   pd = r;
12.   do
13.       generate_pruning_length(n,m,pd,Q,rank[i],O[i]);
14.       pd = pd - Δr;
15.   while Prob(n,m) ≥ Θ & (r=pre_r || pd > pre_pd[i])
16.   pre_r = r; pre_pd[i] = pd + Δr;
17.   // pruning on-the-fly
18.   for each DC from l_DC to r_DC
19.       num_diff_bit=0;
20.       for j=0 to n-1
21.           if DC[rank[i][j]] & 1 ≠ QDC[i][rank[i][j]] & 1
22.               num_diff_bit++;
23.       if num_diff_bit < m
24.           add DC to candidates[];
25. return candidates[];
```

Figure 6. SPA Algorithm

Next, SPA aims to identify the pair (n, m) such that $Prob(n, m)$ satisfies the desired fraction of space to search - Θ (line 10-16). Given a *pd* value which is initialized to be r, the method *generate_pruning_length* tests values of m from 1 to D and generate the corresponding n by enforcing the *partial distance constraint*. The (n, m) pair which minimized $Prob(n, m)$ can be computed this way. If the minimum $Prob(n, m)$ is still too large, *pd* will adjusted to a smaller value so that a new (n, m) pair (and corresponding a new minimum $Prob(n, m)$) is generated.

After finalizing the pair (n, m), further processing takes place by application of Theorem 4.2 (line 17-24). Any DC with m or more different bits from the query DC is safely pruned. At this step, the DC comparison involves only bitwise operations on n bits (instead of D) making it CPU efficient.

Consider Figure 4 again. A cluster space is divided into four partitions by the cluster center. Given a search radius r, *pd* is first initialized to r. Since $pd < d_y$ and $pd < d_x$, the best (n, m) pair is identified as (2,1). Thus, any point in the space having a DC with 1 or more mismatches can be pruned. The query DC is encoded as 11, thus the partitions identified by DCs of 10, 01, and 00 can be safely pruned. The final searching space is the light shaded area in the partition identified with a DC of 11. We shall use another example to illustrate the concept of SPA.

Example 1 An Example on SPA: *For illustration purposes, we assume the L_1 norm (instead of L_2) as the metric function. Consider a 2NN query with query point Q=(0.9,*

Q	$(0.9, 0.1, 0.55, 0.7, 0.35)$
$\|Q[i] - O[i]\|$	(0.4,0.5,0.05,0.2,0.15)
key	1.3
DC	10110
$rank$	[1,0,3,4,2]

Table 2. A query with its key, DC and rank.

P	$Data$	key	DC
P_1	(0.1,0.9,0.3,0.55,0.0)	1.45	01*0*10
P_2	(0.35,0.2,0.95,0.8,0.9)	1.7	00*1*11
$\mathbf{P_3}$	(0.85,0.15,0.6,0.65,0.45)	1.1	10*1*10
P_4	(0.2,0.8,0.65,0.95,0.4)	1.2	01*1*10
$\mathbf{P_5}$	(0.92,0.15,0.4,0.6,0.25)	1.32	10*0*10
P_6	(0.65,0.8,0.1,0.4,0.3)	1.05	11*0*00
P_7	(0.15,0.9,0.3,0.1,0.7)	1.45	01*0*01
P_8	(0.4,0.1,0.25,0.7,0.75)	1.3	00*0*11
P_9	(1.0,0,0.99,0.05,0.95)	2.49	11*0*11

Table 3. A cluster of data points with keys and DCs.

0.1, 0.55, 0.7, 0.35) and a cluster center O=(0.5, 0.6, 0.5, 0.5, 0.5). The respective DC, the indexing key and rank array for Q are depicted in Table 2, and a cluster of data points is shown in Table 3. It can be derived manually that the 2NNs are P_3 and P_5 with distance 0.4 and 0.42 respectively.

Our main KNN algorithm initializes the search radius $r = 1.0$, $\Theta = 0.4$ and $\Phi = 3$. We set Δr to 0.5. SPA first allocates the left node by a point search on the B^+-tree on key value = max(dist(Q,O)-r,O.min)=1.05. Similarly, the right node is allocated by point search on key value = min(dist(Q,O)+r,O.max)=2.3. Obviously, all the DCs, except P_9's, will be scanned for DC comparison.

Next, SPA generates a pair (n, m) which minimizes $Prob(n, m)$, given the value of pd. When pd=1.0 and under the partial distance constraint of

$$\sum_{i=n-m}^{n-1} (|Q[rank[i]] - O[rank[i]]|) \geq pd$$

, the pair (5,4) will minimize $Prob(n, m)$. However, since $Prob(n, m)$ is still higher than Θ, the value of pd will be reduced by Δr to have a value of 0.5. With this new value of pd, SPA will minimize $Prob(n, m)$ to a value of $\frac{5}{2^4}$=0.31 by setting (n, m) to (4,2).

In the final step, since (n, m) is set to (4,2), the candidates with 2 or more different bits in the 4 most distinguishable dimensions (from the rank array, we can see that they are dimension 1,0,3 and 4) will be pruned. In this example, only P_3 and P_5 will remain. Since the number of candidates is less than Φ and equal to K, only data points P_3 and P_5 are accessed.

After computing the distance on the full dimension, the distance to the second nearest neighbor is 0.42, which is less than pd=0.5 (notice that current r is 1.0), and the search stops.

4.4. Optimizing the Generation of (n, m)

There are three main cost terms comprising the overall cost of SPA. First, it is the point search in the LDC tree. Since a point search in a B^+-tree is extremely fast, this part can almost be neglected. Second, it is the cost of generating the pair (n, m). And third, it is the cost of comparing DCs using bitwise operations.

It is evident that for large values of n and m, computing $Prob(n, m)$, becomes prohibitively expensive. The following theorem alleviates this complication:

Theorem 4.4 *As the value of $\frac{n}{m}$ increases, $Prob(n, m)$ decreases if n>2m.*

Theorem 4.4 states that as $\frac{n}{m} \uparrow$ (increases), $Prob(n, m) \downarrow$ (decreases). Hence, the problem of identifying the (n, m) pair that minimizes $Prob(n, m)$ becomes equivalent to the problem of identifying the (n, m) pair with the maximal $\frac{n}{m}$ value. This observation leads to a much more efficient procedure for setting the parameters (n,m). The meaning of the Θ threshold has to be refined however, as it becomes a *threshold for the value* $\frac{n}{m}$. By doing so, the computational cost reduces to computing n given m and the partial distance constraint. Furthermore, the range of m values to be tested (identifying n given a pd value, such that $\frac{n}{m} > \Theta$) can be less than $\frac{D}{\Theta}$ and greater than the value of m, such that (assuming the L_1 distance is used) $\sum_{i=0}^{m-1} (Q[rank[i]] - O[rank[i]]) > pd$.

Let us revisit Example 1. We use $\frac{n}{m}$ rather than $Prob(n, m)$ and set Θ to be 2. Given pd=1.0, we find that m should be larger than 3. Since the dimensionality is 5, m should be less than $\frac{5}{2}$. Thus no suitable m can be produced. In the next iteration, a value $pd = 0.5$ is used. In this iteration, we conclude that m should be larger than 1. Thus, only m=2 is tested and n=4 is generated. Since $\frac{4}{2}$ =2 which is equal to Θ, the pruning length and candidate pruning length are generated. From this example, we can see that $\frac{n}{m}$ and $Prob(n, m)$ have the same capability to control the number of candidates. However, by using $\frac{n}{m}$, the computational cost is very little. In the rest of paper, Θ refers to the threshold value of $\frac{n}{m}$.

4.5. A Cost Model

The I/O cost of SPA can be calculated as:

$$IO_{total} = IO_{B^+} + IO_{DC} + IO_{candidate}$$
$$\approx IO_{DC} + IO_{candidate}$$

where IO_{B^+} is the number of pages required for point search on a B^+-tree, IO_{DC} is the number of pages required for sequential scan at the DC level, and $IO_{candidate}$ is the number of pages required to access the candidate data points. IO_{B^+} is very small and can be neglected, as B^+-trees have guaranteed logarithmic access cost.

An index technique has to be more efficient than sequential scan to be effective. A sequential I/O is usually a factor of 10 times faster then random access. Hence one of the important factors affecting the I/O cost is the number of candidates for random access. Next, we derive the proper value for Φ (the maximal number of candidates allowed for random access to ensure the efficiency of the indexing structure) for LDC tree. In the LDC tree,

the size of DC level is only $\frac{1}{32}$ (assuming 32-bit precision) of data level. The KNN in the LDC tree is iterative and parts of DC level are scanned in each iteration. In the worst case, the whole DC level is scanned (only when the whole space is searched). Notice that each candidate is accessed only once. To ensure SPA is more efficient than sequential scan, the following relationship has to be true assuming that a sequential I/O is 10 times faster than random I/O:

$$\Phi * 10 + \alpha * \frac{Page_{scan}}{32} \leq Page_{scan}$$

where α represents the number of iterations for a query to finish its KNN search and $Page_{scan}$ represents the total number of pages for the whole data set. Following the above formula, we have:

$$\Phi \leq \frac{(32 - \alpha)}{320} * Page_{scan}$$

From the above formula, we can see that in the best case when only one iteration is processed, Φ can reach nearly 10% of the data size. Generally speaking, running tens of iterations for KNN is not desirable. Setting Δr based on the difference between the cluster's minimal and maximal radii can prevent the number of iterations to be large. Even when α reaches 10 or more, let's say 16, Φ can still be as large as 5% of the data size. The above formula assumes that the whole DC level is scanned in each iteration. In a real situation, only part of the DC level is scanned. Hence Φ can be set to even a larger value.

The processor cost of SPA can be calculated as:

$$CPU_{total} = CPU_{n,m} + CPU_{DC} + CPU_{candidate}$$
$$\approx CPU_{DC} + CPU_{candidate}$$

where $CPU_{n,m}$ is the computational cost to generate the (n, m) pair, CPU_{DC} is the cost to compare DCs by performing bit operations, and $CPU_{candidate}$ is the cost to compute the distance on the complete set of dimensions on the candidate data points. Using the optimization of Section 4.4, the cost of $CPU_{n,m}$ becomes negligible.

5. Experiments

We have conducted a comprehensive set of experiments, using both real and synthetic data sets, to evaluate the overall LDC methodology proposed in this paper.

5.1. Experimental Setup

Otherwise stated, all experiments were performed on a Sun UtraSparc II 450Mhz (2 CPU), with 4G memory. We used a page size of 4K (different page size may have different effect on the results. However it is not our focus here). Unless otherwise stated, all results reported are averages over 100 queries, for 10NN. We adopted the L_2 norm for indexing and searching. Our experiments were conducted using the following data sets:

1. **Real datasets**: We created one data set extracted from 73,715 WWW images randomly crawled from over 40,000 web sites. It consists of 159-dimensional color histograms extracted from these images.
2. **Synthetic datasets**:

   We have created several uniformly distributed datasets in order to be able to effectively vary the dimensionality. The first four sets of 100K points each are in 128-, 216-, 512-, and 1024-dimensional space. We used three additional datasets consisting of 50K, 100K, and 200K 1024-dimensional data points to evaluate the effect of increasing data size on SPA. Note that a 1024-dimensional data point spans the 4K page boundary and hence a random access of each point involves reading two pages. The last two sets we used, consist of 100,000, 512-dimensional data points constructed by generating each coordinate independently using a Zipfian distribution, with skew parameters 0.7 and 1.0 respectively.

The most popular techniques in the literature, suitable for indexing spaces of high dimensionality are the VA-file[11] and the IQ-tree[1]. We also tested the effectiveness of iDistance [12]. Our experiments indicated that the IQ-tree structure does not scale gracefully to very large dimensionality. Figure 7 presents the results of an experiment showing SPA's, VA-file's, IQ-tree's and iDistance's total response time as the dimensionality of the underlying data space increases [2]. Among all methods, SPA performs best.

It is evident that the total response time for IQ-tree increases nearly exponentially with respect to the dimensionality. This phenomenon is pronounced when the dimensionality reaches hundreds or higher. The total response time for the IQ-tree is two times that of sequential scan in a 256-dimensional space. The main reason for IQ-tree's behavior, in our opinion, is the use of Minimum Bounding Rectangles (MBRs). In very high dimensionality, there are three main drawbacks associated with MBRs: First, the distance between each MBR and random query points becomes extremely small. Most MBRs intersect with the searching spheres defined by queries, which leads to accessing most of the points in the underlying space. Second, the distance computation between an MBR and a query is expensive in a data space of very high dimensionality. Third, the representation of each MBR is too large. Given a D-dimensional space, representing MBRs in this space requires 2D-dimensional vectors. For a 512-dimensional space given a page size of 4K, the size of each MBR is exactly the page size (assuming 4 byte coordinates).

For iDistance, its performance degrades with the increase in dimensionality and is slightly worse than sequential scan in a 256-dimensional space. The reason is obvious. As the dimensionality increases, the transformation from full dimensionality to one dimensional distance value causes more information loss. Thus the ability to prune the search space is degraded. Clearly, more candidates will be included for random access as the dimensionality increases.

The performance of the VA-file structure, which is better than the IQ tree and iDistance, is known to be linear to the dimensionality and better than sequential scan if the dataset is uniformly distributed. Thus, we adopt

2 This experiment was conducted under windows XP with a Pentium III processor (600Mhz) and 128RAM.

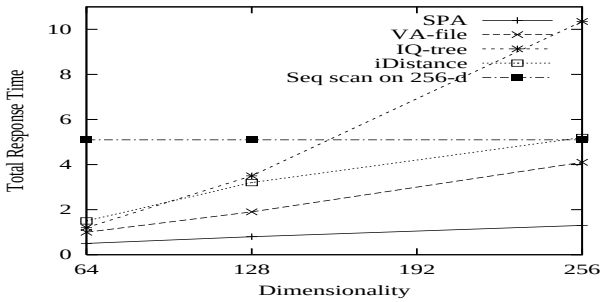


Figure 7. Effect of dimensionality on efficiency.

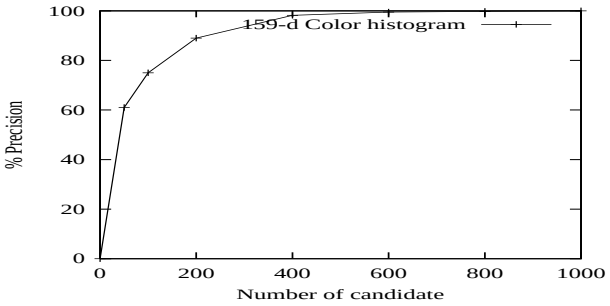


Figure 9. Effect on precision for real dataset.

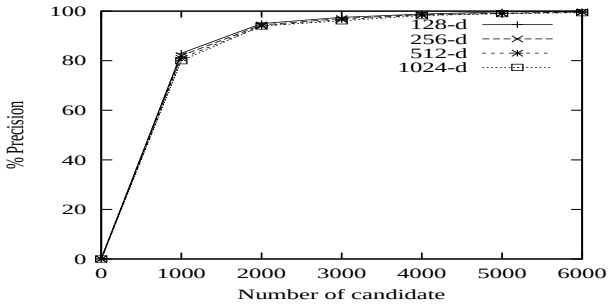


Figure 8. Effect on precision for uniform datasets.

the VA-file (uniformly representing each dimension in the VA-file by 5 bits) as a basis for our comparison.

5.2. Effects of Φ

One interesting thing about SPA is that it can control the number of candidates to be less than Φ by adjusting Θ. Hence it is necessary to know the relationship between the retrieval precision for KNN search and the number of candidates. Next, we investigate such a relationship in SPA by testing uniform datasets with 100,000 points and the real dataset. Figure 8 shows the precision for uniform datasets as the number of candidates increases for different dimensionalities. We can see that the precision is not sensitive to the dimensionality. For all dimensionalities, the precision reaches nearly 100% after the number of candidates is above 3000, which is around 3% of the data size. Hence 3% of data size might be a good choice for the value of Φ for uniform datasets. Figure 9 shows the precision for the real dataset as the number of candidates increases. As we can see, the precision grows up to almost 100% after the number of candidates is more than 400, which is less than 0.6% of the data size. Comparing with 3% of the data size for uniform datasets, the color histogram dataset requires a much smaller number of candidates relatively to the data size. This is due to the skewness of the color histogram dataset.

5.3. Effects of Data Size

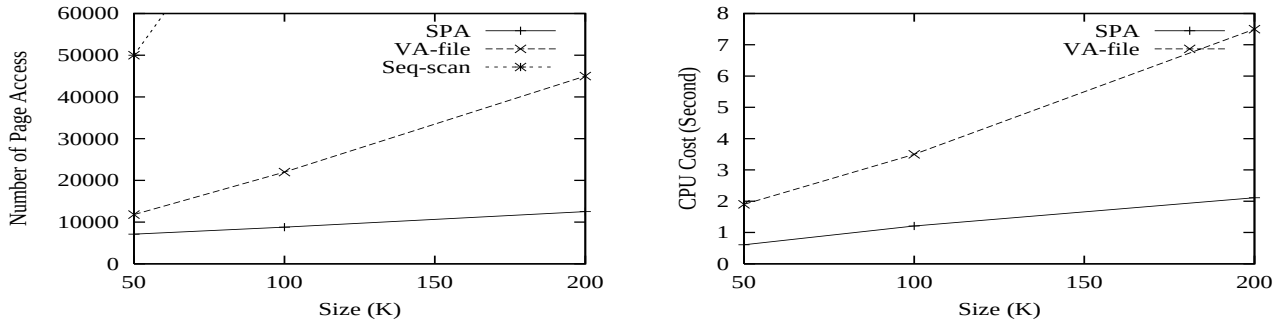
We now investigate the effect of data size on both I/O and processor cost. In this experiment, we tested a 1024-dimensional uniform dataset varying its size from 50K to 200K, and an 159-dimensional color histogram, varying its size from 10K to 70K.

Figure 10 shows the improvement achieved by SPA over the VA-file. Both the I/O and processor cost of VA-file increase faster than that of SPA, as the data size increases. It is evident that SPA offers an improvement over the IO and processor cost of VA-file, by nearly four times. Sequential scan on processor cost is not presented as its cost is *too large to fit* in the figure. This is also applied to the following figures.

The I/O cost of SPA mainly consists of two parts: sequential scan on part of the DCs and access to the candidate data points. The I/O cost of the VA-file consists of sequential scan on the VA-file and access to the candidate data points. The DC level of the LDC tree is just $\frac{1}{5}$ of the size of VA-file. Although the KNN search in the LDC tree is iterative (i.e., DC level may be scanned more than one time), it can outperform the VA-file since SPA controls the number of candidates.

The main processor cost for SPA is the DC comparison, where only bit operations are involved. However, in the VA-file, there is a distance computation for every VA. In a very high dimensional space, the full dimensional distance computation is too expensive. Figure 10b presents an experiment that verifies this statement. In this figure the processor cost for the VA-file reaches 7 seconds.

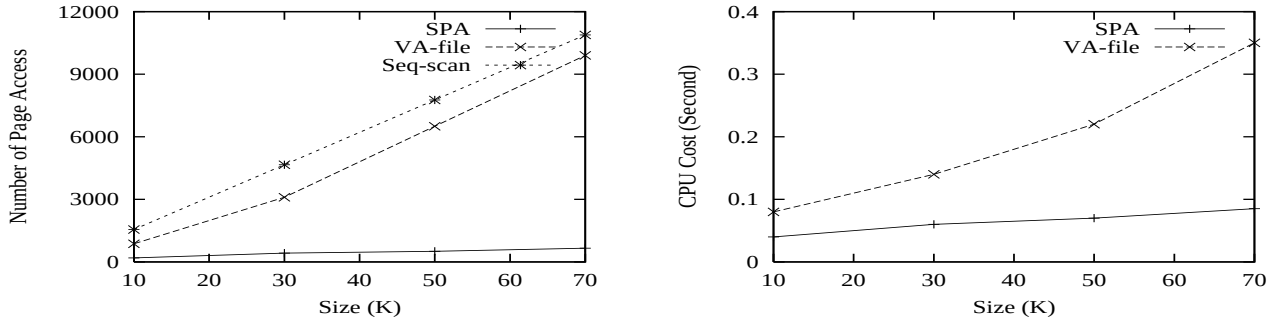
Figure 11 presents the results on the 159-dimensional color histogram dataset. In terms of the total I/O cost (Figure 11a), the VA-file performs close to a sequential scan. In contrast, SPA only requires a small number of I/Os. In this experiment, SPA improves the I/O cost over the VA-file by more than an order of magnitude. Notice that the color histogram is a skewed dataset (few dimensions have large values, many have 0 or close to 0 values). Comparing with Figure 10a which presents the results of the same experiment on a uniform dataset, we observe that SPA performs even better in skewed datasets while the performance of VA-file is much worse. Recall that SPA ranks the dimensions with respect to local reference points (cluster centers) and chooses the most distinguishable dimensions as the candidate pruning dimensions so that a larger partial distance can be used. In color histograms, the most distinguishable dimensions are obviously those dimensions with large values. Hence, comparing to uniform data, by keeping the ratio $\frac{n}{m}$ large, a smaller value of candidate pruning length is used and a smaller number of candidate data points remain for full distance comparison, In contrast to SPA, the VA-file includes most of the data points as candidates when it computes its lower and upper bounds for skewed data. Figure 11b again shows the significant advantages in terms



(a) Effect on I/O

(b) Effect on CPU

Figure 10. Effect of Data Size on Uniform Dataset.



(a) Effect on I/O

(b) Effect on CPU

Figure 11. Effect of Data Size on Color Histogram Dataset.

of processor time of SPA over the VA-file.

Our experiments indicate that SPA attains superior performance on skewed data sets, as shown by our experiments with image color histograms, in Figure 11.

5.4. Effects of Dimensionality

With this set of experiments we demonstrate the robustness of SPA as the dimensionality of the underlying data space increases. We use uniform datasets of size 100K points, of dimensionality 128, 256, 512, and 1024 respectively. Figure 12a demonstrates that SPA outperforms the VA-file more substantially as the dimensionality increases. The higher the dimensionality, the better is the performance of SPA. This is mainly because the number of candidate points tested in SPA is independent of the dimensionality and the cost of accessing DCs increases much slower than in the case of the VA-file. Figure 12b again confirms the performance advantages of SPA over a VA-file on processor time.

5.5. Effects of Skewness

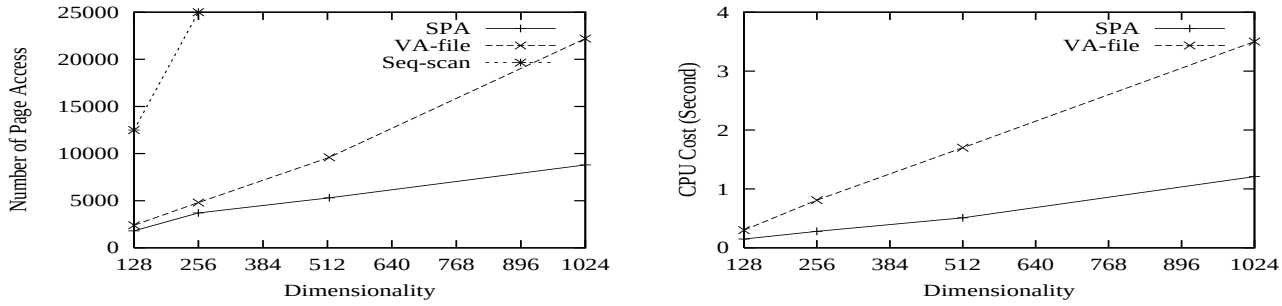
In this experiment, we will demonstrate that SPA is even more effective for skewed datasets, where the KNN search is more meaningful. In skewed datasets, the most skewed dimensions are expected to be better amenable to pruning, since SPA chooses the most distinguishable (or skewed) dimensions to maximize the partial distance. Thus, comparing with uniform datasets, SPA is expected to have better performance.

We generated two 512-dimensional data sets, by populating the coordinates using a Zipfian distribution with skew parameter 0.7 and 1.0 respectively. Both consist of 100,000 data points. Figure 13 shows the effect of skew on both SPA and the VA-file. Figure 13a indicates that as the skew parameter of the Zipfian distribution increases, the I/O cost of VA-file increases rapidly. When the skew parameter becomes 1.0, the performance of VA-file is even worse than that of sequential scan. In contrast, SPA still performs much better. The performance gap between the two methods gets rapidly much wider. In real life datasets that are mostly skewed to a certain degree, we expect SPA to achieve excellent performance as demonstrated in Figure 11 using real data sets. Similar observations can be made for processor costs as shown in Figure 13b.

6. Conclusions

In this paper, we introduced a very effective data organization and representation methodology called Local Digital Code suitable for hyper-dimensional data. Such representation encompasses the application of partial distance and accommodates a novel KNN search algorithm - SPA.

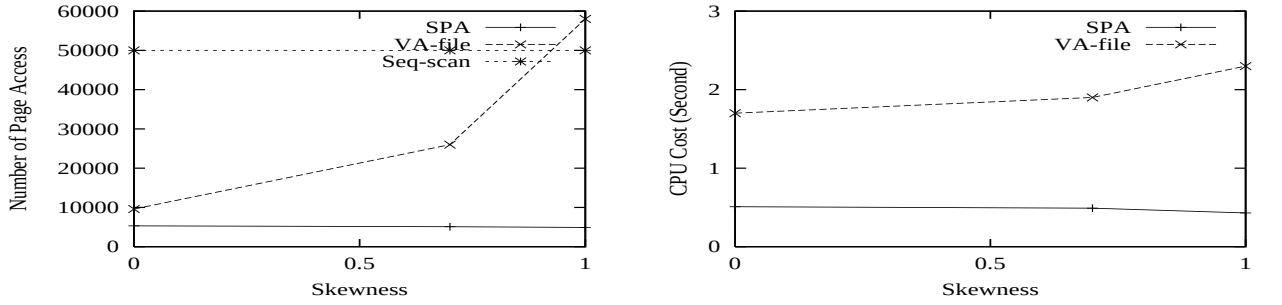
SPA uses the minimal partial distance computed from any m dimensions among n most informative dimensions between the query and static reference points (cluster



(a) Effect on I/O

(b) Effect on CPU

Figure 12. Effect of Dimensionality on Uniform Dataset.



(a) Effect on I/O

(b) Effect on CPU

Figure 13. Effect of Data Skewness.

centers), as the partial distance. Such partial distance computation avoids accessing data points so that the overall computational costs are minimized. SPA is capable of pruning points in the data space rapidly, without computing distances among them, employing DCs and simple bitwise operations. Moreover, SPA can minimize the candidate point set that requires retrieval and further processing, by employing the results of our analytical methodology. Our extensive performance study on hyper-dimensional data demonstrated that SPA outperforms known methods significantly.

As a future work, methodologies in the spirit of LDC can be possibly designed for other access method as well (besides B^+-trees); it would be worthwhile to examine the relative tradeoffs and benefits in such cases.

References

[1] S. Berchtold, C. Böhm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE*, pages 577–588, 2000.

[2] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *SIGMOD*, pages 142–153, 1998.

[3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbors meaningful? In *ICDT*, pages 217–235, 1999.

[4] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, pages 33(3):322–373, 2001.

[5] A. Borodin, R. Ostrofsky, and Y. Rabani. Subquadratic Algorithms for Approximate Clustering in High Dimensional Spaces. *Proceedings of ACM STOC*, 1999.

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

[7] A.P. de Vries, N. Mamoulis, N. Nes, and M.L. Kersten. Efficient k-NN search on vertically decomposed data. In *SIGMOD*, pages 322–333, 2002.

[8] S. Guha, R. Rastogi, and K. Shim. CURE: An Efficient Clustering Algorithm for Large Databases. *Proceedings of ACM SIGMOD*, pages 73–84, June 1998.

[9] H. Jin, B.C. Ooi, H.T. Shen, C. Yu, and A. Zhou. An Adaptive and Efficient Dimensionality Reduction Algorithm for High-Dimensional Indexing. In *ICDE*, pages 87–98, 2003.

[10] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, pages 516–526, 2000.

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

[12] C. Yu, B.C. Ooi, K.L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to KNN processing. In *VLDB*, pages 166–174, 2001.

[13] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *Proceedings of ACM SIGMOD, Montreal Canada*, pages 103–114, June 1996.