Project

Xian Yang, Ge Shi, Shuying Guan

Dec 2018

1 Description

In the area of similarity search in high-dimensional vector spaces, there exists a number of conventional methods which proves to be efficient, such as space-partitioning methods, data-partitioning index trees and bottom-up methods. However, the dimensional curse is still a unavoidable problem for those methods as the performance could significantly degrade with the increase of dimension. This paper did a thorough analysis on those methods and demonstrates that they could be easily outperformed by a sequential scan whenever the dimensional is above 10. To further support the impact of high dimension in HDVS, this paper establishes a general model for clustering and partitioning to accurately evaluate and predict the average cost of similarity search in HDVSs. It also formally shows that these methods inherit linear complexity at high dimensionality which results in degeneration to a sequential scan if the dimension goes beyond the limit.

Aiming at getting rid of the curse of dimensionality, this paper proposes an alternative technique in similarity search call vector approximation which potentially improves the efficiency of sequential scan as soon as possible. This method is initialized by dividing data spaces into rectangular cells without hierarchical organization as R-tree has. After we obtain the approximation of data points according to their belonging cells, nearest neighbor search is conducted in a scanning manner with a filtering step which drastically excludes the candidate vectors based on these approximations.

2 Vector Approximation

The vector approximation file (VA-File), is an array of compact, geometric approximations to data points [2]. It divides each dimension of data space into 2^{b_j} parts which can be represented by a b_j bits index. Given a data vector, the approximation of it is the concatenation of index in of dimension based on the geometric clue. The space uniquely defined by d dimension indices is called a block. Therefore, the approximation indicates which block the data vector lies in and we can get a rough estimation of the location of the data vector from it. This gives us a heuristic idea on retrieving the nearest neighbors of an anchor data vector with low cost of searching.

Table 1: Notation summary of VA-File

d	number of dimensions	j	range of dimensions $1, 2, ..., d$
n	number of vectors	i	range of vectors $1, 2, ..., n$
$\vec{v}_i$	$i-th$ vector	$\vec{v}_{i,j}$	$j-th$ dimension component of $\vec{v}_i$
b	number of bits per approximation	b_j	$j-th$ dimension bits number per approximation
a_i	approximation for $\vec{v}_i$	$r_{i,j}$	region into which $\vec{v}_i$ falls in dim j
$m_j[k]$	$k-th$ partition mark in dim j	p	p power in distance
L_p	weighted distance function $L_p(\vec{v}_i, \vec{v}_q)$	k_b	binary form of index k
l_i, u_i	lower and upper bounds of a block	$l_{i,j}, u_{i,j}$	contribution to l_i, u_i for dimension j

2.1 Structure

Assume n is the number of data vectors and d the number of dimensions, we use i to denote the $i-th$ data vector and j to denote the $j-th$ dimension. To build a VA-File, we need all the data vectors to share the same number of dimension. Here we assume the range of the magnitude of data vectors is between 0 and 1. For each vector, a b-bit approximation is derived. Then, to implement a VA-File, we need to (1) allocate b_j bits to each dimension which sum up to b; (2) divide each dimension into 2^{b_j} partitions and determine the division marks value $m_j[k]$, $k \in 0, ..., 2^{b_j}$; (3) assign a data vector an approximation based on its geometric location in each dimension; (4) get the lower and upper bounds l_i and u_i of the distances between an anchor vector $\vec{v}_q$ in the query and all the blocks; (5) Filter out the invalid blocks and compute the distances between the data vectors in the valid blocks with $\vec{v}_q$.

The relationship between the total number of bits b and the number of dimensions d determines b_j as follows:

$$b_j = \lfloor \frac{b}{d} \rfloor + \begin{cases} 1, \text{ j} \leq b \mod d \\ 0, \quad \text{otherwise} \end{cases}$$

Small b_j is usually enough for high dimension data. A b-bit approximation makes 2^b blocks which is over 1 billion when $b = 30$. If there are 30 dimension, 1 bit allocation for each dimension is enough to make over 1 billion blocks.

The number of bits in each dimension is used to determine partition points, and hence regions within each dimension. In particular, b_j bits separate the range of one dimension into 2^{b_j} regions within dimension j from 0 to 2^{b_j}, requiring $2^{b_j} + 1$ partition points, including 0 and 1 specifically in our setting. The running example in Figure 1 illustrates this. After partitions, we locate each data vector $\vec{v}_i$ by checking it's in which two partition marks. Assume $m_j[k] < \vec{v}_{i,j} < m_j[k+1]$, where $k \in 0, ..., 2^{b_j}$, we can set the approximation $a_{i,j}$ of the $i-th$ data in dimension j as binary expression of k, denoted as k_b. The concatenation of k_b denotes the name of the block where the data vector locates. In practice we use a dictionary structure to save the approximation blocks and the corresponding points where the key is the binary approximation and the value is a list of the data vector that share the same approximation.

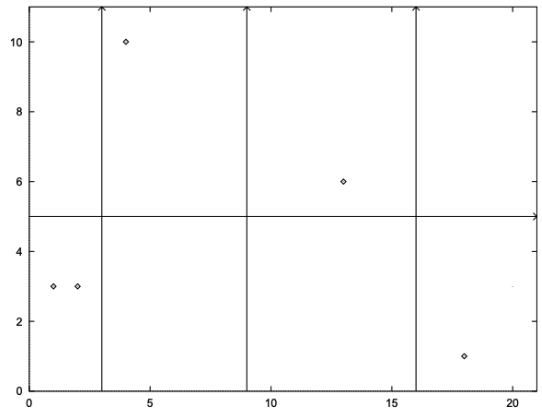


Figure 1: A two-dimensional example (d = 2, b = 3)

2.2 Loading Data

There are many ways to partition each dimension such as even partition and equal-fully partition. Even partition, for example partition the range $[0, 1]$ into two even parts $[0, 0.5]$ and $(0.5, 1]$ without considering the distribution of data, gives a bad selectivity property (the average number of data vectors in each non-empty

block) since a lot of the blocks are empty while others are nearly full. Thus, we adopt equal-fully partition which needs to know the latent distribution of data vectors.

The distribution of data can be got through two ways: (1) sort the data based the values of each dimension to get the accurate distribution; (2) make a histogram of the values of data in each dimension to get an estimation of the distribution. For n vectors in d dimensions, function (1) can be done in O(dnlog(n)) and function (2) can be done in O(dn). We can get the partition marks through finding the correspondent values of even partition of cumulative distribution function (CDF) marks. Suppose we partition the $j-th$ dimension into 2^{b_j} regions, then we can get $2^{b_j}+1$ marks.

An approximation a_i for each vector v_i is generated as follows. Let the 2^{b_j} regions in dimension j be numbered $0, ..., 2^{b_j}+1$. Let $r_{i,j}$ be the number of the region into which $v_{i,j}$ falls. We define a point to fall into a region only if it is greater than or equal to the lower bound for the region, and strictly less than the upper bound for the region. Therefore, $v_{i,j}$ falls into the region numbered $r_{i,j}$ if and only if:

$$m_j[r_{i,j}] \leq v_{i,j} < p_j[r_{i,j}+1]$$

Then, we can load the data vectors and get the approximation of them one by one and saved in the suggested dictionary of list structure.

2.3 Bounds

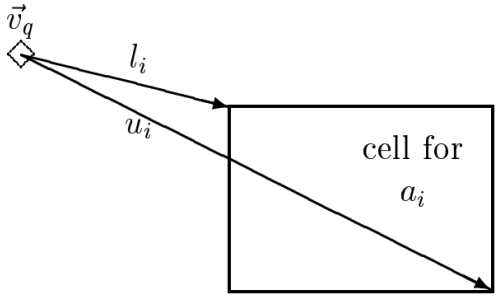


Figure 2: The lower and upper bounds of the distance from a block to the anchor vector

A easy way to do nearest search using VA-File is filtering out the blocks that have zero chance to contain the nearest neighbor by computing the upper and lower bounds of each block corresponding the anchor vector in the query. Assume a query $\vec{v}_q$ and a distance function L_p, for some power parameter p. An approximation a_i determines a lower bound l_i, and an upper bound u_i such that:

$$l_i \leq L_p(\vec{v}_q, \vec{v}_i) \leq u_i$$

The distance between two vectors is defined as:

$$L_p(\vec{v}_q; \vec{v}_i) = (\sum_{j=1}^{d}(w_j \cdot |a_j - b_j|)^p)^{\frac{1}{p}}$$

We usually define $w_j = 1$ and $p = 2$.

The lower bound l_i and upper bound u_i can be derived as follows. The lower bound and upper bound of the distance from a block to vector v_q in dimension j is $l_{i,j}$ and $u_{i,j}$. As with data vectors, a query $\vec{v}_q$ consists of components $v_{q,j}$, the component of its approximation in dimension j is $a_{q,j}$. The decimal representation of $a_{i,j}$ is $r_{q,j}$. Similarly, the decimal representation of each block in dimension j is $r_{i,j}$. Then the two marks

defines a block in dimension j are $m_j[r_{i,j}]$ and $m_j[r_{i,j}+1]$. Based on this information, the bounds l_i and u_i are defined by the equations, see Figure 2:

$$l_i = (\sum_{j=1}^{d} (l_{i,j}^p)^{\frac{1}{p}}, where \begin{cases} \mathrm{v}_{q,j} - m_j[r_{i,j}+1], r_{i,j} < r_{q,i} \\ 0, \quad \mathrm{r}_{i,j} \quad = \quad r_{q,i} \\ \mathrm{m}_j[r_{i,j}] - v_{q,j}, r_{i,j} > r_{q,i} \end{cases}$$

$$u_i = (\sum_{j=1}^{d} (u_{i,j}^p)^{\frac{1}{p}}, where \begin{cases} \mathrm{v}_{q,j} - m_j[r_{i,j}], r_{i,j} < r_{q,i} \\ \mathrm{MAX}(\mathrm{v}_{q,j} - m_j[r_{i,j}], m_j[r_{i,j}+1] - v_{q,j}), r_{i,j} = r_{q,i} \\ \mathrm{m}_j[r_{i,j} + 1] - v_{q,j}, r_{i,j} > r_{q,i} \end{cases}$$

2.4 Nearest Search Algorithm

The pseudo code of nearest search algorithm is as following: The details are:

```
PROC VA-NOA(v_q: Vector);
VAR i, d, l_i, h_i: INT; Heap: HEAP; Init(Heap);

(* PHASE - ONE *)
   d := InitCandidate();
   FOR i := 1 TO v DO
      l_i, u_i := GetBounds(a_i, v_q);
      IF l_i <= d THEN
         d := Candidate(u_i, i);
         InsertHeap(l_i, i);
      END;
   END;

(* PHASE - TWO *)
   d := InitCandidate();
   l_i, i := PopHeap(Heap);
   WHILE l_i < d DO
      d := Candidate(L_p(v_i, v_q), i);
      l_i, i := PopHeap(Heap);
   END;
END VA-NOA;
```

Figure 3: The lower and upper bounds of the distance from a block to the anchor vector

(1) candidate is an array of size k, where k is the number of nearest neighbor we want.
(2) In phase-one, the candidate array is initialized as the number of dimension d. (3) The function Candidate(u_i, i) is inserting the upper bound and approximation of a block using binary search from small to big. (4) d updated as the biggest value in the candidate array which is the $k-th$ smallest upper bounds among all the blocks. (5) The Heap is a smaller-pop first heap. (6) In phase-two, the candidate array is initialized with the d get from phase-one. (7) Check if the heap is empty before pop.

3 R-Tree

3.1 Bulk Loading with Sort-Tile-Recursive

R-tree has been widely discussed and used in multi-dimentional database systems. This index technique improves query performance by storing a collection of rectangles and pointers that reflects the hierarchical data structure. However, general inserted based R-tree building method is highly susceptible to the order of input data. Also, considering the efficiency of building an R-tree on high dimensional, we implemented Sort-Tile-Recursive for R-tree packing.

In order to do bulk loading for an R-tree, consider a d-dimensional data set of n points. We are supposed to build hyper-rectangles with d intervals of the form $[l_i, d_i]$ where $i \in [1, d]$. These intervals will be used to identify which rectangle that each point belongs to as well.

Sort-Tile-Recursive will be performed recursively on each dimension. We set the capacity of leaf pages to c, which is the maximum nodes that a leaf page could hold. Then we could calculate and get the number of leaf pages $P = \lceil \frac{n}{c} \rceil$ and number of slices on each dimension $S = \lceil P^{\frac{1}{d}} \rceil$. According to these numbers, we firstly sort points based on the first coordinate of them and partition them into S rectangles. (e.g. If the first coordinate is defined by x-axis, we will get S vertical rectangles.) After we have processed the hyper-rectangles from the sorted list of first coordinate, each slice is now further expanded recursively using the remaining $d - 1$ coordinates.

3.2 K - Nearest Neighbor Search

The search for nearest neighbor in R-tree is has been studied to improve the efficiency. [1] has proposed a new algorithm to enable searching for nearest neighbor faster. [1] has introduced two metrics for nearest neighbor searching. The first one is based on the minimum distance between current block and query point. The second metric is based on the minimum of the maximum possible distances between query point and a face of non-empty minimum bounding rectangle (MBR). The first metric minimum distance (MINDIST) and the second metric minimum maximum distance (MINMAXDIST) are the lower bound and upper bound of actual distance between actual point and query point.

For MINDIST between an MBR M and the query point q for n dimension, we employ below definition. l_i indicate the lower bound of ith dimension in MBR and u_i indicate the upper bound of ith dimension in MBR. MINDIST indicate the possible smallest distance between points in MBR and query point.

$$MINDIST(M, q) = \sum_{i=1}^{n} |q_i - r_i|^2$$

where

$$r_i = \begin{cases} \mathrm{l}_i, q_i & < \quad l_i \\ \mathrm{u}_i, q_i & > \quad u_i \\ \mathrm{qi, otherwise} \end{cases}$$

For MINMAXDIST between an MBR M and the query point q for n dimension, we employ below definition. l_i indicate the lower bound of ith dimension in MBR and u_i indicate the upper bound of ith dimension in MBR.

$$MINMAXDIST(M, q) = \min_{1 \le k \le n} (|q_k - rm_k|^2 + \sum_{\substack{i \neq k \\ 1 \le i \le n}} q_i - rM_i)$$

where

$$rm_k = \begin{cases} \mathrm{l}_k, q_i \le \frac{l_k + u_k}{2} \\ \mathrm{u}_k, otherwise \end{cases}$$

$$rM_i = \begin{cases} \mathrm{l}_i, q_i \ge \frac{l_k + u_k}{2} \\ \mathrm{u}_i, otherwise \end{cases}$$

With these two metrics, we below two rules to help us pruning when we search the k nearest neighbor from root.

- When there are already k candidate points available, which means at least k points in the candidate MBR which are going to be gone through and the MINDIST(M_i,q) is greater than the last MINMAXDIST(M_j,q), M_i could be discarded since we could ensure we get at least k points whose distance is smaller than the distance between points in M_i and query point.
- When we already get k actual points in the result and the MINDIST(M, q) is greater than the greatest distance between k acutal point and query point, M could be discarded. Because we already find k points that their distance from query point is less than the minimum distance between points is M and query point.

In implementation, we start from the root MBR. For each MBR, we store the lower bound and upper bound of the MBR in each dimension, the number of points that is inside this MBR. For MBR which are leaves, we store the actual points inside this MBR and for MBR which are not leaves, we store the MBR id that is inside current MBR. Since we use bulk loading, any MBR would only under one MBR, no overlaps.

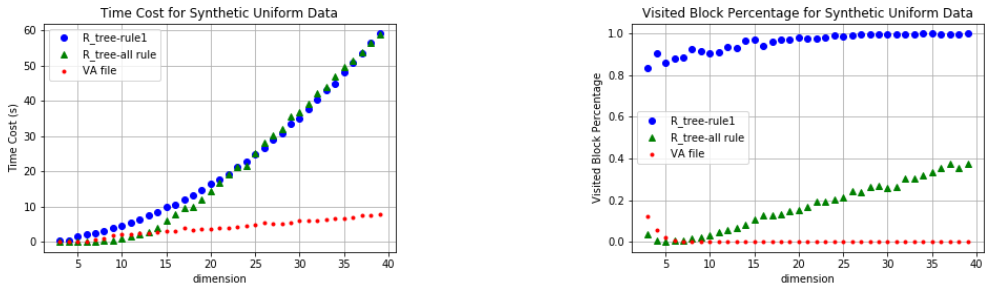
If current MBR is leaf, we calculate the distance between each points inside this MBR and query point and only keep the k smallest distance point. If current MBR is not leaf, we calculate the MINDIST and MINMAXDIST of MBRs inside current MBR, then we sort them by MINMAXDIST for each MBR. Starting from the smallest MINMAXDIST MBR, we apply the rules mentioned above to filter MBR. After filtering, we visit the MBR and filter its children with criteria in current paragraph with the sequence that we filter them.

4 Experiments & Results

For VA-File, we performed experiment with 4 splits at each dimension. For R-tree, we performed experiment with at most 8 splits at each dimension when bulk loading the data. To avoid bias, for each type of data in each number of dimension, we perform the experiment 5 times and use its average time and average visited block percentage for result. For synthetic data, we generate query point randomly from 0.0 to 1.0. For real data, we randomly select a point from the real dataset to be query point. The euclidean distance is used.

4.1 Uniform Data

We perform experiment on 50000 uniform data range from 0.0 to 1.0.



(a) Time Cost Compared

(b) Visited Block Percentage Compared

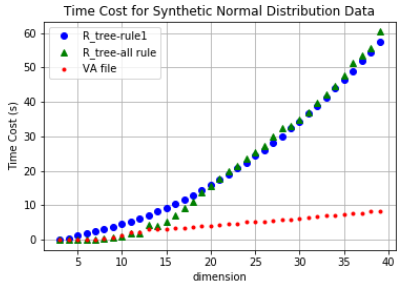
Figure 4: Experiment Result on Synthetic Uniform Data

We perform experiment to find the 10 nearest neighbor point with VA file algorithm, R-tree nearest neighbor search with only rule 1 mentioned above and R-tree nearest neighbor search with both rules mentioned above on 50000 synthetic uniform data with increasing dimension from 3 to 39. As shown above, time cost for R-tree grows exponentially fast while time cost for VA file grows slowly when dimension increase.

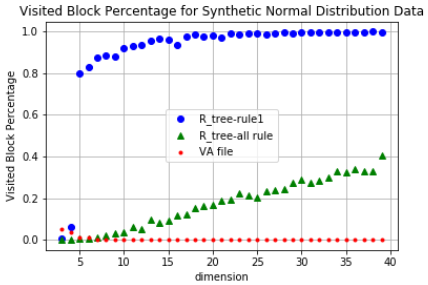
As for visited block percentage, when only the first rule applied for R-tree, the visited block percentage start high and gently grow to 1.0 as dimension increase. When all rules applied for R-tree, the visited block percentage start slow and gradually grow while percentage for VA file decrease when dimension grow.

4.2 Normal Data

We perform experiment on 50000 normal distribution data range from 0.0 to 1.0.



(a) Time Cost Compared



(b) Visited Block Percentage Compared

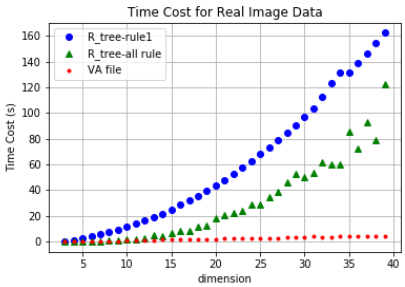
Figure 5: Experiment Result on Synthetic NOrmal Distribution Data

We perform experiment to find the 10 nearest neighbor point with VA file algorithm, R-tree nearest neighbor search with only rule 1 mentioned above and R-tree nearest neighbor search with both rules mentioned above on 50000 synthetic normal distribution data with increasing dimension from 3 to 39. As shown above, time cost for R-tree grows exponentially fast while time cost for VA file grows slowly when dimension increase.

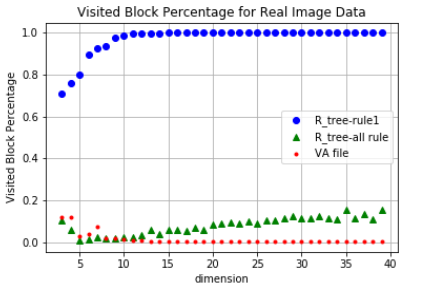
As for visited block percentage, when only the first rule applied for R-tree, the visited block percentage start high and gently grow to 1 as dimension increase. When all rules applied for R-tree, the visited block percentage start slow and gradually grow while percentage for VA file decrease when dimension grow.

4.3 Real Data

We use Labelled Faces in the Wild (LFW) dataset which is a large-scale face attributes dataset. We extract 13143 images from it, resize them into 8 x 8 images and convert them into grey scale images. These images give us 13143 64 dimension data range from 0.0 to 1.0 to perform experiment.



(a) Time Cost Compared



(b) Visited Block Percentage Compared

Figure 6: Experiment Result on Real Image Data

We perform experiment to find the 10 nearest neighbor point with VA file algorithm, R-tree nearest neighbor search with only rule 1 mentioned above and R-tree nearest neighbor search with both rules mentioned above on real image data with increasing dimension from 3 to 39. As shown above, time cost for R-tree grows exponentially fast while time cost for VA file grows slowly when dimension increase.

As for visited block percentage, when only the first rule applied for R-tree, the visited block percentage start high and gently grow to 1 as dimension increase. When all rules applied for R-tree, the visited block percentage start slow and slowly grow while percentage for VA file decrease when dimension grow.

4.4 Time Cost for R-tree

The plot of time cost of R-tree is slightly different from what we get from the original paper. In the plot of the paper, the time cost will converge relatively fast to a constant as this technique degenerate to a sequential scan. However, our plot shows a exponential increase in time cost and does not converge even we increase the test dimension to 39. We think there are several reasons resulting this plot:

- We did not use a constant number of blocks for R-tree. Instead, we only specify the number of split in STR which could constantly increase the number of blocks as dimension goes infinity. So there literally is no upper bound for number of candidate rectangles.
- The Nearest Neighbor search algorithm we implemented is totally different from the method proposed in the paper. We combined multiple rules to accelerate the process of searching possible overlapping rectangles by eliminating the majority of distant rectangles, which is achieved by the defined MinMax distance and is quite similar to VA search algorithm. According to the plot of Visited Block percentage, our R-tree has not yet decayed to the scanning method around dimension 40. So it is reasonable that the time cost does not converge. However, I do think there could be a threshold on dimension where the time cost of our R-tree would converge.

4.5 Performance on Synthetic Data and Image Data

Based on the plots of Visited Block Percentage, we observe that the two methods both perform better on real image data than on the synthetic data. The reason that have caused this could be:

- For the synthetic data, each coordinate is generated independently on other coordinates. So the entire data is sparse with high dimensional. On the contrary, real image data usually possess latent pattern within multiple dimensions. That is, each coordinates could possibly be highly correlated to other coordinates, thus making NN search much easier for both search algorithms.
- The size of the real image data is smaller than the synthetic data. On this note, with the same number of split on every dimension in R-tree, smaller data set will obtain a shallower index tree. As we do not allow the creation of empty leaf node in R-tree, the visited percentage converges even though the dimension still increases.

5 Reference

[1] Vincent, N.R.S.K.F., Nearest Neighbor Queries* Nick Roussopoulos Stephen Kelley Fréedéeric Vincent Department of Computer Science University of Maryland College Park, MD 20742.

Weber, R., Schek, H.J. and Blott, S., 1998, August. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In VLDB (Vol. 98, pp. 194-205).