Similarity Multidimensional Indexing[♣]

© Elena Mikhaylova, Boris Novikov, Anton Volokhov

Saint Petersburg state university

egmichailova@mail.ru, borisnov@acm.org, a.v.volokhov@gmail.com

Abstract

The multidimensional k-NN (k nearest neighbors) query problem arises in a large variety of database applications, including information retrieval, natural language processing, and data mining. To solve it efficiently, database needs an indexing structure supporting this kind of search. However, exact solution is hardly feasible in multidimensional space. In this paper we describe and analyze an indexing technique for approximate solution of k-NN problem. Construction of the indexing tree is based on clustering. Construction of hash indexing is based on s-stable distributions. Indices are implemented on top of high-performance industrial DBMS.

Keywords: k-NN search, multidimensional indexing, LSH

1. Introduction

Efficiency of search is critical for modern information retrieval. Commonly, search queries retrieve relatively small portion of information. In this case, indexing search is much more efficient, than the full scan. Any given process or stored object can be characterized by a set of features that are usually called attributes. The purpose of any index is quick access to the object by the values of some of its attributes. In other words, the indices provide effective implementation of associative search.

If attribute values belong to a linearly ordered set, searching can be implemented with one-dimensional indices. These indexing structures (such as B-trees and hash-files) are well studied in 70-80 years. But many real applications have a goal to find by values of several attributes or by attributes that can't be naturally linearly ordered. In this case we need to implement searching in multidimensional space. Of the numerous multidimensional indexing structures proposed in the 80-ies., only R-trees [1] passed the test of time. The R-tree indexes are widely used in various applications and implemented in industrial databases.

Multidimensional searching is primarily associated with processing spatial and spatio-temporal data. Another class of applications, processing multidimensional data includes systems based on various flavor of a text vector model for methods of data mining, pattern recognition, data compression etc. Data obtained from the application of this class are typically characterized by high dimensionality.

Typically, the processing of multidimensional data requires searching on the basis of proximity or similarity, rather than exact attributes equality. The most common search query is to find K nearest neighbors for a given dataset.

In this work we present and compare two approaches for similarity multidimensional search. One idea is based on tree-clustering structure, the second one uses LSH method. The former is better in precision and the latter supersedes in the speed of computation. The remaining part of the paper is organized as follows. Section 2 contains the overview of related work. Section 3 informally outlines the techniques used in our approach. Our algorithms and data structures are presented in section 4, followed by analysis of experiments or results in section 5.

2. Related work

Years of multidimensional searching evolution led to development of various indexing algorithms. Best known of them are R-trees [8], X-tree [2], AV-files and many others [3]. In the past decade, considerable attention is attracted by the scheme of space-sensitive hashing (LSH - locality sensitive hashing). M-trees [4] are intended for indexing the points of a metric space.

R-tree indices [8] work well for low-dimensional spatial data. But R-tree family degrades rapidly due to the overlap in index pages in high dimensions. The problem of multidimensional search is extremely difficult with large-dimensional vectors. It is well known that the construction of universal indexing structures for large-scale data is impossible. This fact, known as the "curse of dimensionality," determined by the topology of the multidimensional space. It does not depend on the index constructing method. The practical consequence is that performance of any index structure, starting at some dimension, becomes worse than a full scan. This problem can be somewhat mitigated by considering the indexing structures and algorithms that give approximate results. Typically, the vectors represent only the proximity of the source application objects (eg vectors for information retrieval can not accurately express the meaning of the

Proceedings of the 13th All-Russian scientific conference "Digital Libraries: Advanced Methods and Technologies, Digital Collections" - RCDL'2011, Voronezh, Russia, 2011

documents). According to that, it is reasonably to use approximate methods for finding. For example, LSH [6] gives only a probabilistic guarantee of correct answer for k-NN query [9, 5].

Locality-sensitive hashing works with vector space. This approach was introduced [15] and well developed [16-17] in last decade.

In paper [15] the idea of using randomized hyperlanes for "sketching" initial dataset was presented.

Then, in paper [16] was introduced improved method, which uses pivots from s-stable distribution to hash vectors from the dataset. Idea to preprocess dataset with dimensionality reduction methods [18] allows implementing new algorithm that appears to be near optimal LSH method [17]. However, it works only with Gaussian norm in Euclidean space.

In many cases, it is computationally difficult not only to find and to prepare vectors (e.g., feature extraction of images), but even to calculate the function of similarity (or distance between vectors). Search could be greatly accelerated if it is possible to store a matrix of pairwise distances between objects. Unfortunately, this solution is not scalable because the size of the index in this case, will square dependence on the number of objects.

In [10] k-NN search problem is dealt with the double filtering effect of clustering and indexing. The clustering algorithm ensures that the largest cluster fits into main memory and that only clusters closest to a query point need to be searched and hence loaded into main memory. In each cluster data is distributed with ordered-partition tree (OP-tree) main memory resident index, which is efficient for processing k-NN queries.

High-dimensional clustering is used by some content-based image retrieval systems to partition the data into groups (clusters), which are then indexed to accelerate processing of queries. Recently, the Cluster Pruning approach was proposed in [12] as a simple way to produce such clusters. The evaluation of the algorithm was performed within an image indexing context. The paper [12] discusses the parameters that affect the efficient to the algorithm, and proposes changes to the basic algorithm to improve performance.

[11] presents an adaptive Multi-level Mahalanobis-based Dimensionality Reduction (MMDR) technique for high-dimensional indexing. The MMDR technique discovers elliptical clusters for more effective dimensionality reduction by using only the low-dimensional subspaces, data points in the different axis systems are indexed using a single B+-tree. The technique is highly scalable in terms of data size and dimensionality. It is also dynamic and adaptive to insertions. An extensive performance study was conducted using both real and synthetic datasets, and the results show that our technique not only achieves higher precision, but also enables queries to be processed efficiently.

In paper [14] a new clustering method is proposed: to group together only points that are close to each other. The remaining points are stored separately, not clustered. Such a structure is obtained unbalanced. In order to create the indexing structure, in [13], the distances between the selected set of pivots and the data objects is computed, sorted and nearest distances are stored in separated tables. For search at first query is compared with pivots.

3. The approach and rationale

In this paper, we describe two methods that were used for approximate multidimensional search. The first one is actually a variation of a matrix tree, based on spanning tree clustering algorithm. The second is an implementation of the local-sensitive hashing (LSH), specifically tuned for high computational speed.

The goal of this work is to construct the sub-optimal index structure for approximate searching of multidimensional objects based on features of similarity (or distance function). This structure will provide a reasonable response time for practical ranges of different parameters. In order to evaluate the characteristics and performance of this structure, we have implemented the model over the data provided industrial relational DBMS. Perhaps this decision affects the efficiency but also ensures quick implementation, suitable for use in the prototype, for example, in an experimental system for content based image retrieval. That is, the developed system allows finding images on a number of characteristics similar to the one desired.

If we have a reasonable matrix of pairwise distances, the problem of finding K nearest neighbors can be solved by one-dimensional index range scan. This operation is efficiently implemented in any relational database. In order to limit the size of an index, we will not store all pairwise distances, but only the distance to the K nearest points. Usually, search engines need to solve the problem of K-nn only for small values of K.

3.1 Locality-sensitive idea

Hash functions family is called locality sensitive with parameters (R, cR, P_1, P_2), if following:

$$\begin{aligned} \|V_1 - V_2\| < R \to Pr_H\big(h(p) = h(q)\big) > P_1 \\ \|V_1 - V_2\| > cR \to Pr_H\big(h(p) = h(q)\big) < P_2 \end{aligned} \quad \textbf{(1)}$$

Intuitively it means exactly that points with low $\|v_1 - v_2\|$ (near in initial space) will collide into the same value with higher probability, then points with higher one.

LSH method is based on hashing dataset with functions chosen uniformly at random from locality-sensitive family. To satisfy approximate nearest neighbor query, we don't need to perform full scan, but can find neighbors only in buckets, where query point falls.

To construct LSH family, one needs to choose hash functions and determine probabilities P_1 and P_2. Indexing structure, according to these parameters can do nearest neighbors searching with following query(2) and preprocessing(3) time:

$$n^{\frac{(\log 1/P_1)}{(\log 1/P_2)}} \quad \textbf{(2)}$$

$$n^{1+\frac{(\log 1/P_1)}{(\log 1/P_2)}} \quad \textbf{(3)}$$

4. Data structures and algorithms

4.1 Algorithms description

In the experiments, we would rely on high-performance relational database system. Our storage structure was constructed on Microsoft SQL-server. The data were presented in the form of points of multidimensional space. We consider the objects as multidimensional vectors.

4.1.1. Spanning tree

The first storage structure organized as follows. For a given vector space we construct a minimum length spanning tree. We accept vectors as individual graph components and compute all pairwise distances between them.

The metric space is defined as a pair (D; d) where D denotes the domain of objects and d: DЧD → R is a total function which must have the well-known non-negativity, symmetry, identity and triangle inequality properties. The distance is calculated using the metric of L1.

We create index structure so:

A. Add to the tree minimal length arc if endpoints of this arc belong to different components. At the same time we restrict maximum number of arcs entering each vertex.
B. Repeat step A until the number of components becomes equal to one.
C. Split the tree to l components by removing l longest arcs. During splitting we monitor the number of vertices in the connected component – it must be more then min and less then max.
D. For all building component we find the central vector – centroid. These centroids are vectors in same space, but theirs quantity is much smaller then amount of source vectors, and we consider them as new level vectors.
E. To a new level we repeat step A – D. So we built the second level of our indexing structure.
F. We add levels with steps A – E until the number of allocated centroids on the top-level became sufficiently small. The latter set of centroids forms the upper level of the index tree.

Search is organized as follows:

A. Begin from upper level component.
B. Calculate distances between the current level's vectors and given search query.
C. We choose the nearest vector and the corresponding component. We proceed to the next level and repeat step B. Then, using precalculated matrix of distances, we choose the nearest centroid and go to the next level.
D. On lowest level we select nearest vectors to the query.

By adding new vector:

A. Repeat steps A-D from search, we find the lowest level component which centre is nearest to the specified vector.
B. If number of vectors exceeds the upper limit, we do not divide it into two parts, and climb one level up and perform re-clustering for all vectors caught in this component.

4.1.2. s-stable distribution LSH

This randomized algorithm is using the idea, described in [16]

Here we use scalar product with random vectors from s-stable distribution to "sketch" initial high dimensional vector.

Locality-sensitive hashing family with parameters (R , cR, P_1, P_2)

(P_1,P_2 – probabilities of "good" and "bad" collisions, R – query radii, c – approximation factor) is defined as follows:

$$h(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor . \quad \textbf{(4)}$$

a – random vector from s-stable distribution, predefined in pivots table; b – random shift, chosen uniformly at random from [0..w]; v – vector from initial dataset.

Due to s-stability, for every two vectors (p, q) distance between their projections (a.p – a.q) is distributed as $\|p-q\|_s X$ where X is a s-stable distribution. It means that if two points a near(low $\|p-q\|$) then they should collide with high probability, and if they are far they should collide with small probability.

However, if we use these functions to preprocess dataset, we will have a plenty of false-positive answers

$\partial = 1-(1-P_1^K)^L$ because of low difference between P_1 and P_2

To increase it, we construct composite hash function g_i, $i \in 1..L$ as a concatenation of hash functions $h_{i,j}$, $j \in 1..K$.

For new hash functions $P_1 = P_1^K$, $P_2 = P_2^K$.

To increase total probability of retrieving answer, we hash dataset with L different hash functions g_i. Equation (5) describes probability д of getting a correct answer.

$$\partial = 1-(1-P_1^K)^L \ . \qquad \textbf{(5)}$$

To make an indexing structure, we need to precompute parameters L, K and w.

w can be estimated using formula (6)

$$w_{min} = argmin_w \frac{\log 1/P_1}{\log 1/P_2} \ . \qquad \textbf{(6)}$$

Parameters K and L are choosing experimentally by constructing different data structures, satisfying equation (7)

$$L = \left\lfloor \frac{\log 1/\partial}{-\log\left(1-P_1^k\right)} \right\rfloor \ . \qquad \textbf{(7)}$$

4.2 Indexing on spanning tree

4.2.1. Data structures

For indexing structure we used 4 tables. First table stores coordinates of vectors. In order to the table structure not to depend on the dimension of space, each vector is represented in the table as a set of rows - one row for each coordinate of the vector. Each row has the following structure:

Vector number	Int
Coordinate number	Int
Value	Float

The second table stores calculated distances between all pairs of vectors. Algorithm performance doesn't depend on distance measure. In our experiments distance was calculated using the metric of L1, but for the method it does not matter. Our function is defined as follows: $r(x, y)=\sum|x_i-y_i|$. With these distances construct a table of distances in each row that stores the distance between two vectors.

Table with attributes:

Vector number 1	Int
Vector number 2	Int
Distance	Float
Component number	Int

We build a spanning tree of minimum length(depth?) and divide it into components. For arcs not belonging to spanning tree component number is equal to zero, otherwise, this field indicates the number of components, which the arc belongs to.

This table will be very large, but it is necessary only at the base index constructing stage. Later during search and insertion, we need only some of pairwise distances.

Vector table is as follows:

Vector number	Int
Component number	Int
Distance	Float
Low-level vectors number	Int

For top-level vectors component number equals zero, else this parameter indicates the number of components, which contains this vector.

We use the distance between vectors and component center by searching. For top-level vectors this distance is zero. For every vector-centroid we store low-level vector number - is necessary for know how many vectors contains this component.

The table structure component:

Component number	Int
Level	Int
Vectors amount	Int
Centroid number	Int

Vectors amount indicates how many vectors are at the lowest level of this component.

4.1.2 Index structure construction

To construct a spanning tree minimum length we use the following algorithm:

1. We consider multi-dimensional vectors as vertices of the tree and the distance between the vectors of the arc. At the lower level we assume each node a separate connected component. We set all component numbers to zero
2. Compute all pairwise distances and sort them in ascending order.
3. Select next minimum length arc with zero component number. Consider vectors - the ends of the selected arc. If these vectors belong to one component, then we skip this arc and repeat step 3 with the next arc.
4. Check the number of arcs included in these peaks and owned by any component. The classical tree construction algorithm was modified to avoid height degree of nodes. To ensure this rule the number of edges adjacent to a node is restricted with a parameter. (no more N_{in}). If at least one end of the arc constraint is violated, then we skip this arc (with marking – set component number negative) and goto step 2.

5. Add to the spanning tree arc between two vertices.
6. Connect these components are connected into one.
7. Continue to step 3 - 6 up until all the vertices not included in one connected component.
8. Divide constructed spanning tree into several connected components. Each component must have from N_{min} to N_{max} vertices. Number of components depends on the number of vectors stored in the table, and is determined with N_{min} and N_{max}. When we split a spanning tree of connected components of the arcs belonging to the spanning tree, we select the arc with maximum length. Exclude this arc from the spanning tree, forming two connected components. Check that the number of vectors in the resulting components more N_{min}. If not, it returns back and mark the removed arc. Iterate arcs from the spanning tree, while not succeed to break a connected component into two parts. Then divide each new piece, while the size of the component will not appear between N_{min} to N_{max}.
9. In each component we find the centroid. To find value of the i-th component centroid we average values of the i-th component's vectors in a given connected component. For each component, we store the number of vectors in it.
10. Consider centroids as multidimensional vectors on the next level. If vectors amount is more than given N_{max} we repeat steps 1-8 of our algorithm.

4.1.3 Search

1. Begin at the top level.
2. Compute distances between stored vectors of current level (inside current component) and query vector, order the list by distance in ascending order. If current level is lowest, we found an answer.
3. Take first vector (with minimum distance), determine the component and the number of vectors contained in it. Exclude the first element of the list. If the number of vectors in the component is less than desired, then repeat step 3.
4. Determine the vectors in selected components and goto step 2.

4.1.4 Scalability

1. Search (step 1-4) for k=1.
2. Selected component contains vectors nearest to given query, and we try to insert new vector into selected component. If the number of vectors in this connected component is less than the maximum allowed, then this component is updated with a new vector. We recalculate centroid's coordinates and the number of vectors inside components on all levels.
3. If number of vectors exceeds the upper limit, we do not divide it into two parts. We climb one level up and find all vectors belonging to selected component of the current level, and perform re-clustering for all vectors of this component. We rebuild spanning tree for a new set of vectors and divide it into connected components. We are doing so at every level of the indexing structure if current component is overcrowded.

Operation of completion thus obtained is sufficient labor intensive. But we anticipate that most queries will be directed at search and not an upgrade.

4.2 LSH-method

4.1.1. Data structures

To implement locality-sensitive hashing using s-stable distributions we used following structures:
Table contained vector number and nested table with its coordinates.

Vector number	Int
Coordinates	Nested Table of Float

Table with predefined pivots from normal distribution and random shift for them.

Pivot number	Int
Coordinates	Nested Table of Float
Random Shift	Float

L tables for different hash functions with identical structure: K columns with hash values of simple hash functions and nested table for keys of vectors that have the same hash values. We can store all hash values in the same table because number of simple hash functions is defined on preprocessing stage.

Hash_value_1	Float
Hash_value_2	Float
…	…
Hash_value_K	Float
Vector keys	Nested Table of Int

And table, that store the choice of pivots for certain hash function

Number of hash function	Int
Pivot_1	Int
Pivot_2	Int
…	…
Pivot_K	Int

Here, number of pivots is also predefined during the preprocessing stage.

We also have a statistics table that is needed only for preprocessing. It helps to determine optimal parameters for indexing structure.

Value of K	Int
Average query time	Float

4.1.2 Preprocessing and index structure construction

Step 1: Preprocessing.
To find optimal values for K and L we build indexing structures for different K from 3 to 15 and appropriate L, calculated by the formula (7)

A. Build indexing structure for certain value K.
B. Hash all points $p \in P$ with L hash functions.
C. Run testing set on this indexing structure.
D. Store value of K and average query time in statistics table.
E. Delete indexing structure.
F. Repeat A-E, until K < 16.
G. Retrieve value of K from statistics table with minimal query time
H. Build indexing structure with obtained value of K and appropriate value of L

Step 2: Search.

I. Hash query point q with hash function gi
J. Retrieve all points from bucket gi(q).
K. If retrieved points are not enough to satisfy k-NN query, then repeat 1-3 with next hash function gi+1
L. Sort retrieved points and return first k elements as an answer.

5. Experiment and analysis

To compare described algorithms, dataset refereed to content-based image retrieval was considered. It consists of 25000 rows with 41 attributes, representing different image characteristics. Methods were implemented within Microsoft SQL-Server DBMS. During experiments average accuracy (percent of exact nearest-neighbors in retrieved approximate answer) and relative time consumption were calculated.

Results for spanning-tree algorithm:

1-NN query – 87%
10-NN query – 70%
50-NN query – 79%

Query response time was 10 times faster than the one in naive full scan.

Method based on LSH produced the following results:
1-NN query – 62%
10-NN query – 50%
50-NN query – 34%
However, hashing algorithm worked approximately 40 times faster than the previous one.

6. Conclusion

In this paper two different algorithms for multidimensional indexing were described. These algorithms were implemented and analyzed with different parameters on real dataset. Based on the results of experiments, we can draw the following conclusions: first method proposed gives a sufficiently accurate result, but compared to LSH lose heavily in time. If the k-NN search is not 2k, but 3-4k, then the precision increases, but difference between tree and hashing query response time is even more significant.

References

[1] Lars Arge, Mark de Berg, Herman J. Haverkort, Ke Yi: The Priority RTree: A Practically Efficient and WorstCase Optimal RTree
[2] Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegei: The X-tree: An Index Structure for High-Dimensional Data, In Proceedings of the 22nd International Conference on Very Large Databases (1996), pp. 28-39.
[3] D. Bremner, E. Demaine, J. Erickson, J. Iacono, S. Langerman, P. Morin, and G. Toussaint, "Output-sensitive algorithms for computing nearest-neighbor decision boundaries," Discrete and Computational Geometry, Vol. 33, No. 4, 2005, pp. 593-604.
[4] Ciaccia et al. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. VLDB-1997, 1997
[5] Thomas M. Cover and Peter E. Hart, "Nearest neighbor pattern classification," IEEE Transactions on Information Theory, (1967) Vol. 13 (1) pp. 21-27
[6] Gionis, A.; Indyk, P., Motwani, R. (1999). , "Similarity Search in High Dimensions via Hashing". Proceedings of the 25th Very Large Database (VLDB) Conference.
[7] Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984: 47-57
[8] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, Yannis Theodoridis: R-Trees: Theory and Applications, Springer, 2005. ISBN 1-85233-977-2
[9] Nigsch, F.; A. Bender, B. van Buuren, J. Tissen, E. Nigsch & J.B.O. Mitchell (2006). "Melting Point Prediction Employing k-nearest Neighbor Algorithms and Genetic Parameter Optimization".

Journal of Chemical Information and Modeling 46 (6): 2412–2422.

[10]Alexander Thomasian, Lijuan Zhang: Persistent clustered main memory index for accelerating k-NN queries on high dimensional dataset Multimedia Tools and Applications Volume 38 Issue 2, June 2008

[11]Heng Tao Shen, Xiaofang Zhou, Aoying Zhou: An adaptive and dynamic dimensionality reduction method for high-dimensional indexing. The VLDB Journal, Volume 16 Issue 2, April 2007

[12]Gylfi Thyr Gudmundsson, Bjцrn Thyr Jynsson, Laurent Amsaleg: A large-scale performance study of cluster-based high-dimensional indexing. Proceeding VLS-MCMR '10 - international workshop on Very-large-scale multimedia corpus, mining and retrieval

[13]Stanislav Barton, Valйrie Gouet-Brunet and Marta Rukoz: Large Scale Disk-Based Metric Indexing Structure for Approximate Information Retrieval by Content. *Proceeding EDBT/ICDT 2011 Joint Conference*

[14]Stephan Gьnnemann, Hardy Kremer, Dominik Lenhard, and Thomas Seidl: Subspace Clustering for Indexing High Dimensional Data: A Main Memory Index based on Local Reductions and Individual Multi-Representations. *Proceeding EDBT/ICDT 2011 Joint Conference*

[15]P. Indyk, R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. *Proceedings of the Symposium on Theory of Computing*, 1998.

[16]M. Datar, N. Immorlica, P. Indyk, and V.Mirrokni. Locality sensitive hashing scheme based on p-stable distributions. *Proceedings of the ACM Symposium on Computational Geometry,2004.*

[17]P. Indyk, A. Andoni. Near optimal hashing algorithms for approximate nearest neighbors in high dimensions. *Foundations of Computer Science,* 2006.

[18]N. Alion, B. Chazelle. Approximate nearest neighbors and Fast Johnson-Lindenstrauss Transform. Proceedings of the Symposium on Theory of Computing, 2006.

♣This work was supported by RFBR (project 10-07-00156-a).