IJIIM
International Journal on Integrated Information Management
Vol.01 (2012) DOI: 10.15556/IJIIM.01.01.002

Efficient indexing methods in the data mining context

Nikolaos Kouiroukidis[a], Georgios Evangelidis[a]

[a] *Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece*

Abstract: The proliferation of applications manipulating enormous sizes of multidimensional vector data, that require indexing in order to support operations like kNN searching in an efficient manner, has revived the interest in multidimensional indexing. It is well established that as the dimensionality of data increases the performance of queries such as range queries and nearest neighbor (1NN or kNN) queries decreases leading to a problem described as "the curse of dimensionality". In this paper we point out problems that arise when indexing multidimensional data in fixed dimensions, such as 8 or 16, because, usually, when dealing with data of higher dimensionality it is common to first apply dimensionality reduction techniques such as principal component analysis. Although there is a plethora of research papers proposing multidimensional indexing structures, most of them report empirical results with relatively small and low dimensionality datasets. We attempt a fair comparison of many state of the art indexing structures designed exclusively to index multi-dimensional points like the Hybrid tree, the iDistance and the P+tree. We include in our comparison the R*tree, a state of the art index designed both for multidimensional points and regions. It is an improvement of the well-known R-tree, and also has been revised and improved further recently. We compare the behavior of the indexes on kNN queries (for k=1 and k=10) with varying dataset sizes and dimensionality. Our goal is to determine the structure(s) that could be used efficiently in the area of data mining. We obtained the source code of the implementations of the related structures from the authors and the R-tree portal.

Keywords: Multidimensional Indexing, Data Mining

1. Introduction

In the last 25 years many indexing techniques have been proposed for the efficient storage and retrieval of multidimensional data. For the one-dimensional case, the ubiquitous B+tree [7] has been incorporated in all commercial and open source database management systems. Many more sophisticated data structures have been proposed to handle the problem of manipulating in an efficient manner enormous sizes of multidimensional data. Most of them try to solve problems concerning range queries and k nearest neighbor (kNN) queries. Difficulties arise in higher dimensions where the problem of the so called "dimensionality curse" has the effect that the higher the dimension in question the more these index structures behave like or even worse than the sequential scan in solving problems like similarity search queries.

These indexing methods usually take advantage of many factors like the manner that space is occupied by the data in question or some characteristics of the way that data space is decomposed that can lead to translating the multidimensional problem into a single-dimensional one that can be efficiently handled by a B+tree. First, we survey some of the most notable indexing structures that have been proposed in the literature, especially the R*tree [1] (successor of R-tree), the Hybrid tree [3], the P+tree [6] and the iDistance [7], and then we try to study and investigate through experimentation various factors that influence these indexes when used to solve kNN queries. These factors are the data dimensionality and the size of the indexed data that usually arise in real world datasets.

The paper is organized as follows. In Section 2, we introduce the index structures we will examine, and in Section 3 we discuss the experiments that have been conducted using these methods. In Section 4, we present our experiments, and, finally, we conclude the paper in Section 5.

2. Review of Popular Multidimensional Indexes

2.1. Hybrid Tree

This indexing structure is characterized by the space partitioning strategy that is employed when a node splits and it supports distance-based queries (both range and nearest neighbor). It is more similar to Space Partitioning (SP-) Based data structures than Data Partitioning (DP-) Based data structures. As the authors state, the overlap is allowed only when trying to achieve an overlap-free split would cause downward cascading splits and hence a possible violation of utilization constraints. The space partitioning within each index node is accomplished with a k-d tree. Each internal node of the regular kd-tree represents a split by storing the split dimension and the split position. The novelty is that there are two split positions in the kd-tree internal node of the Hybrid tree. The first split position represents the right (higher side) boundary of the left (lower side) partition while the second split position represents the left boundary of the right partition.

When splitting a data node the Hybrid tree chooses as the split dimension the one that minimizes the increase in EDA (expected number of disk accesses per query), thereby optimizing the expected search performance for future queries and the split position is determined as close to the middle as possible along the specified split dimension. This tends to produce more cubic BRs (Bounding Rectangles) and hence ones with smaller surface areas. The smaller the surface area, the lower the probability that a range query overlaps with that BR, the lower the number of expected number of disk accesses.

Unlike data node splitting where the choice is independent of the query size, the choice of the split dimension for index nodes depends on the probability distribution of the query size. For splitting an index node the Hybrid tree chooses the dimension that minimizes the increase in EDA averaged over all queries. Given the split dimension, the split positions are chosen such that the overlap is minimized without violating the utilization requirement.

In addition, the Hybrid tree indexes empty space using the Encoded Live Space (ELS) Optimization using a few bits. The Hybrid tree is completely dynamic with insertions, deletions and updates that occur interspersed with search queries without requiring any reorganization.

2.2. P+Tree

This index structure first divides the data space into clusters that are essentially hyper-rectangles. Then it transforms each subspace into a hypercube and applies the pyramid technique [2] in each subspace. The number of subspaces is always an integral power of 2. The order of Division is defined as the times we divide the space and it is an important parameter of the P+tree. To facilitate the building process of the P+tree and query processing, there is an auxiliary structure called the space-tree, which is built during the space division process. The space-tree is similar to the k-d tree, but it stores the transformation information instead of data points in the leaf nodes. For the construction phase, a P+-tree is basically a B+-tree where the data records with their keys are stored in the leaf nodes.

In the P+-tree, the authors apply the pyramid technique in each subspace. Under the pyramid technique, pyramid values of points in pyramid i cover the interval $[i, i + 0.5]$. There are 2d pyramids, from pyramid 0 to pyramid $2d - 1$, so pyramid values of all points are within the interval $[0, 2d)$. To discriminate points from different subspaces, SNo * 2d is added, where SNo is the number of subspaces. Window queries Q are processed in two logical phases. First, clusters that intersect Q are determined so that the rest of the clusters can be pruned, and second, the transformed query T(Q) is applied on the transformed subspaces. For kNN queries, a hypercube-shaped window query centered at x is initiated with an initial side length, which is typically small. Then the side length increases gradually until we are sure that the kNNs are found.

2.3. iDistance

The design of iDistance was motivated by the following observations. First, the (dis)similarity between data points can be derived with reference to a chosen reference or representative point. Second, data points can be ordered based on their distances to a reference point. Third, distance is essentially a single dimensional value. This maps high-dimensional data in single dimensional space, thereby enabling reuse of existing single dimensional indexes such as the B+tree. Moreover, false drops can be efficiently filtered out without incurring expensive distance computation. The transformation of high-dimensional points to single dimensional points is done using a three-step algorithm. In the first step, the high-dimensional data space is split into a set of partitions. In the second step, a reference point is found for each partition. Suppose that we have m partitions, $P_0 , P_1 , . . . , P_{m-1}$ and their corresponding reference points, $O_0 , O_1 , . . . , O_{m-1}$. Finally, in the third step, all data points are represented in a single dimensional space as follows. A data point $p : (x_0 , x_1 , . . . , x_{d-1})$, $0 \le x_j \le 1$, $0 \le j < d$, has an index key, y, based on the distance from the nearest reference point O_i as follows:

$$y = i \times c + dist(p, O_i)$$

where c is a constant used to stretch the data ranges. Essentially, c serves to partition the single dimension space into regions so that all points in partition Pi are mapped to the range $[i \times c, (i + 1) \times c)$.

For a kNN query centered at q, a range query with radius r is issued. The iDistance kNN search algorithm searches the index from the query point outwards, and for each partition that intersects with the query sphere, a range query is issued. If the algorithm finds k elements that are closer than r from q at the end of the search, the algorithm terminates. Otherwise, it extends the search radius by Δr and the process is repeated till the stopping condition is satisfied.

2.4. R-tree*

The R*-tree is a data partitioning structure that indexes MBRs (minimum bounding rectangles). The minimization of both coverage and overlap of the MBRs influences the performance of R* tree. When overlap occurs on data query or insertion, more than one branch of the tree needs to be expanded and traversed (due to storage redundancy). When the coverage is minimized this has the effect of improving pruning performance, so whole pages are excluded from search more often.

The R*-tree attempts to reduce both, with a combination of a revised node split algorithm and the concept of forced reinsertion when nodes overflow. This is based on the observation that R-tree structures are highly sensitive to the order in which their entries are inserted, so an insertion-built (rather than bulk-loaded) structure is likely to be sub-optimal. So the deletion and reinsertion of some entries allows them to "find" a place in the tree that may be more appropriate than their original location. When a node overflows, a portion of its entries are removed from the node and reinserted into the tree. This produces better- clustered groups of entries in nodes, with the effect that node coverage is reduced. Furthermore, actual node splits are often postponed, causing average node occupancy to become higher. Re-insertion can be seen as a method of incremental tree optimization triggered on node overflow.

3. Previous experiments

In this section we present the experiments that have been carried out during the evaluation of the above-mentioned index structures in the corresponding original papers. The experiments vary in terms of the chosen dimensionality, dataset size, page size, and type of queries tested. For example, the Hybrid tree was tested against the SR-tree [10], the hB-tree [4] and the sequential scan, the P+- tree against the pyramid technique, iMinMax [5] and the sequential scan, iDistance against the iMinMax, the A-tree [9], and the linear scan, and, finally, the R*-tree against other R-tree variations and the GRID file [11].

The summary of the experiments that were carried out by the authors is illustrated in Table 1. We notice that there is no common framework to compare all four indexes and draw conclusions regarding their performance on kNN queries. In the following section we describe our approach in comparing the four indexes.

Table 1: Comparison of index structures regarding previous experimental analysis

	Index	**indexes compared**	**Size Dimensionality**	**distribution**	**Block size**	**Range Query**	**KNN-query**
1	R* tree	quadratic split Rtree, linear split Rtree, Greene's Rtree	100000 2d	uniform, clustered, parcel, real, gaussian, mixed uniform	1024 bytes	rectangle intersection	-

2	Hybrid tree	SR tree, hB-tree, Sequential Scan	1.2 million 16d 70K-16,32,64d	fourier datasets collhist dataset	4096 bytes	Selectivity 0,07% Fourier 0,2% COLHIST	-
3	P+ tree	Pyramid, iminmax	581012 10d 68040 32d 500000 16d32d	real data set real data set clustered	4096 bytes	-	2-10-NN
4	IDistanc e	M tree,Omni Seq.Scan, Arya's bbd tree	100000 to 500000 of 8,16,30 dim	uniform & clustered	4096 bytes	-	10-50NN

4. Experimental Evaluation

We conducted a series of experiments on the four indexing structures using for each one implementation code provided by the authors. We varied certain parameters like the dimensionality, the data set size, and the parameter k in kNN queries, as well as parameters that are unique in each of the participating indexing structures like the number of clusters in iDistance or the order of Division in P+-tree.
Regarding the R*-tree, we used the C++ implementation by Hans-Peter Kriegel's group obtained from the R-tree portal site. We had to make major changes to the original code in order to compile it with the g++ 4.3.2 compiler running under Linux. Our system was a modest one, having one Intel Pentium 4 CPU running at 1.8 GHz with 2GB RAM.

Our purpose was to run experiments with very large data sets with more than 100.000.000 points each, but the creation of indexes of some of the tested indexing data structures, namely, the R*-tree and the Hybrid tree in 16-d data sets proved to be a very lengthy operation. For example, it took one day to create an R*-tree with 16.000.000 points. So, we decided to test all index structures with smaller data sets and achieve a unified view of their performance. The page size was set to all of the competing indexing structures to 4096 bytes and the k value in the kNN search algorithm was set to 1 and 10 for both of the 8 and 16 dimension cases that we tested. The number of clusters in the iDistance method was set to 64 in all the experiments, and the order of division parameter in the P+-tree implementation was set to 4 in all the experiments. We implemented a simple data generator for the experiments. Finally, because the P+-tree implements only window queries, we had to experiment for the appropriate side length of the window query so that correct average answers of 1 and 10 NN points could be computed accounting for 8 and 16 dimensions.

Table 2 shows the time required to build the indexes for varying dimensionality and dataset size. iDistance does not provide us with such information. We observe that the Hybrid tree is about an order of magnitude faster than the rest of the methods. All methods scale linearly to both the dimensionality and the dataset size.

Table 2: Index build time (real time in secs)

	DIM=8			DIM=16		
	100K	**200K**	**300K**	**100K**	**200K**	**300K**
R* tree	77	159	237	148	299	448
Hybrid tree	4	10	13	13	28	45
P+ tree	47	107	167	78	153	268

Regarding the size of the created indexes, iDistance and P+ tree are essentially B+-trees and the code we obtained does not compute the relevant statistics. The R*-tree and the Hybrid tree report similar results, with the Hybrid tree showing increased storage requirements as the dataset size increases, as shown in Table 3.

Table 3: Index size in pages

	Dim=8			Dim=16		
	100K	**200K**	**300K**	**100K**	**200K**	**300K**
R* tree	1516	3035	4552	2942	5883	8824
Hybrid tree	1111	2213	4155	2242	4457	8099

Table 4 reveals that the Hybrid tree, although it is a true multidimensional index, has the smallest average page I/O when dealing with 1NN queries, and it is insensitive to dimensionality. The P+tree has also very good performance due to the fast searching time of the B+-tree that it employs.

Table 4: Average page I/O for 1NN

K=1	Dim=8			Dim=16		
	100K	**200K**	**300K**	**100K**	**200K**	**300K**
R* tree	16	22	40	987	1226	1452
Hybrid tree	3	9	10	3	9	10
iDistance	176	202	231	908	1125	1262
P+ tree	3	10	22	3	36	63

On the other hand, iDistance appears to be more insensitive to the increase in dataset size and dimensionality that the R*-tree. The R*-tree has the worst performance in high dimensions, but performs well in low dimensionality. In the experiments we used comparable buffer sizes for all methods.

Table 5 shows that when dealing with 10NN queries, the P+tree has the best overall performance. For such queries, the Hybrid tree joins the R*-tree regarding the deterioration in performance in high dimensions. On the other hand, iDistance has the worst performance in low dimensionality, but is comparable to the R*-tree and Hybrid tree in higher dimensions.

Table 5: Average page I/O for 10NN

K=10	Dim=8			Dim=16		
	100K	**200K**	**300K**	**100K**	**200K**	**300K**
R* tree	41	56	67	1695	1818	2046
Hybrid tree	20	32	54	1604	1815	2012
iDistance	362	402	476	1789	1920	2146
P+ tree	10	36	31	687	1007	1399

Finally, in Table 6 and Table 7 we show the query times for the R*-tree and the Hybrid tree in the case of 1NN and 10NN queries. We do not report the times for the rest of the methods that use B+-trees because they are almost zero. The Hybrid tree has a very good behavior, comparable to the P+-tree and the iDistance, although it is a true multidimensional index. The R*-tree has the worst performance.

Table 6: 1NN queries execution times (real time in secs)

K=1	Dim=8			Dim=16		
	100K	200K	300K	100K	200K	300K
R* tree	1,5	3	4,49	1,9	3,86	5,69
Hybrid tree	0	0	0	0,08	0,11	0,08

Table 7: 10NN queries execution times (real time in secs)

K=10	Dim=8			Dim=16		
	100K	200K	300K	100K	200K	300K
R* tree	1,57	3,12	4,7	1,94	3,88	6,04
Hybrid tree	0	0	0	0,17	0,23	0,25

Acknowledgements

We thank the authors for providing us with the code of the implementations of the relevant indexing methods namely R* tree, Hybrid tree, P+ tree and iDistance.

References

[1] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The R*-tree: An efficient and robust access method for points and rectangles. In Proc. SIGMOD..

[2] Berchtold, S., Bohm, C., and Kriegel, H.-P. (1998). The pyramid- technique: Towards breaking the curse of dimensionality. In Proc. SIGMOD.

[3] Chakrabarti, K. and Mehrotra, S. (1999). The hybrid tree: An index structure for high dimensional feature spaces. In Proc. International Conference on Data Engineering, pp. 322–331.

[4] Lomet, D.B. and Salzberg, B. (1990). The hB-tree: a multiattribute indexing method with good guaranteed performance. ACM TODS, 15(4), pp. 625-658.

[5] Ooi, B.C., Tan, K.L., Yu, C., and Bressan, S. (2000). Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In Proc. ACM PODS, pp. 166–174.

[6] Zhang, R., Ooi, B.C., and Tan, K-L. (2004). Making the Pyramid Technique Robust to Query Types and Workloads. In Proc. ICDE, pp. 313-324.

[7] Yu, C., Ooi, B.C., Tan, K.L., and Jagadish, H. (2001). Indexing the distance: an efficient method to knn processing. In Proc. International Conference on Very Large Data Bases, pp. 421–430.

[8] Comer, D. (1979). The Ubiquitous B-Tree. ACM Computing Surveys, 11(2), pp. 121–137.

[9] Sakurai, Y., Yoshikawa, M., Uemura, S., and Kojima, H. (2000). The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. In Proc. International Conference on Very Large Data Bases, pp. 516-526.

[10] Katayama, N. and Satoh, S. (1998). SR-tree: An index structure for nearest-neighbor searching of high-dimensional point data. Systems and Computers in Japan, 29(6), pp. 59-73.

[11] Nievergelt, J., Hinterberger, and H., Sevcik, K.C. (1984). The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Trans. Database Syst., 9(1), pp. 38-71.