Improving Query Efficiency in High Dimensional Point Indexes

Evangelos Outsios and Georgios Evangelidis

University of Macedonia, Department of Applied Informatics, 54006, Thessaloniki, Greece
{outsios, gevan} (at) uom.gr

Abstract: *In this paper, we focus on the leaf level nodes of tree-like k-dimensional indexes that store the data entries, since those nodes represent the majority of the nodes in the index. We propose a generic node splitting approach that defers splitting when possible and instead favors merging of a full node with an appropriate sibling and then re-splitting of the resulting node. Our experiments with the hB-tree, show that the proposed splitting approach achieves high average node storage utilization regardless of data distribution, data insertion patterns and dimensionality.*

Keywords: *K-dimensional point indexing, Optimizing data node storage utilization, Range query performance*

I. INTRODUCTION

Lately, with the increased interest in Data Mining, indexing of k-dimensional vectors has become essential when dealing with kNN classification. Brute force application of kNN classification on large databases involves as many computations of distances as the size of the database, since one has to find the k closest points to the query point. Data reduction and/or data dimensionality reduction techniques are used to reduce the computational cost, but they usually decrease the accuracy of the kNN classifier. Alternatively, indexing can be used to reduce the linear cost of searching to logarithmic. Unfortunately, all high-dimensional indexes suffer from the "dimensionality curse" problem. It has been shown that above 8 dimensions, most indexes perform no better than the exhaustive sequential search of the whole database when answering kNN queries (Berchtold et al., 1998).

For very large high-dimensional datasets, the most sensible approach to kNN classification is a combination of a data dimensionality reduction technique, to reduce the dimensions down to 8 to 16, and then, the use of a high dimensional point index. That is why the quest for efficient indexes in medium to low dimensions has regained the interest of the research community. Efficient indexes should not be affected by the cardinality of the dataset, the data distribution, the dimensionality, and the insertion patterns. Since the kNN classifier is a model-free classifier, new insertions in the dataset should dynamically update the model, i.e., the index, without affecting its performance.

Indexes with guarantees in node storage utilization, obviously, lead to better query performance, since fewer nodes (disk pages) are visited to answer a query. kNN queries are a specialization of range queries and require visiting of multiple leaf or data level nodes of the index, where rids of the points or the points themselves are stored.

In this paper, we deal with tree-like k-dimensional indexes that partition the space in non-overlapping subspaces, like the KDB-tree (Robinson, 1981) or the hB-tree (Lomet and Salzberg, 1990; Evangelidis et al., 1997). The hB-tree and the hB-pi* tree (Zhou and Salzberg, 2008), a variation that also indexes empty space, have been recently shown to outperform the R*-tree (Beckmann et al., 1990), the most well known spatial index. We focus on their leaf or data level index nodes since those nodes represent the majority of the index nodes. We propose a generic node splitting approach that delays data node splitting when possible and instead favors redistribution of the contents of a full node with an appropriate sibling. Our experiments with the hB-tree, show that the proposed splitting approach achieves high node storage utilization and good range query performance.

In Section II, we present related work in improving data node storage utilization and provide a short description of the KDB-tree and the hB-tree. We also present a policy for selecting the splitting attribute in high dimensional indexes. We propose a new data node splitting method in Section III, and we present experimental results in Section IV. Finally, we conclude the paper in Section V.

II. RELATED WORK

In this section, we review the approaches that have been proposed in the literature for improving storage utilization. First, we discuss the 1-dimensional case with the B+tree, and then, we briefly describe the structure of the hB-tree and the KDB-tree. Finally, we give some insight on how splitting attribute selection policies can improve storage utilization and range query performance, when splitting data nodes.

A. Data node storage utilization

For the B+tree, there are many ways to increase node storage utilization (Comer, 1979). For example, Knuth (1973), proposes to delay splitting by locally redistributing the contents of nodes until two sibling nodes become full. Then the two full nodes are split into three nodes with a node storage utilization of at least 66%, an improvement over the 50% storage utilization of the B+tree. Although the average node storage utilization remains unaffected and about 69% (≈ ln2) for uniform data distributions (Yao, 1978), this approach achieves better storage utilization for non-uniform data distributions.

In addition, even for uniform data distributions, index performance is affected by the way data points are inserted in the index. For random (uniform) insertion patterns there is no difference on the way nodes are split. As long as nodes are split in a 1:1 ratio, average storage utilization is close to 69%. But under different patterns of insertion, for example, block insertions, where incoming points are inserted to a particular node until that node splits, average storage utilization can degrade considerably.

The picture is quite different when indexing in high dimensions. Almost all of the proposed k-dimensional indexes do not provide such guarantees. Only the hB-tree, that splits its nodes in a 1:2 ratio in the worst case, achieves a comparable to the B+tree average node storage utilization (about 67%) and worst node storage utilization of 33%. But under certain patterns of record insertions, even the hB-tree can have average node utilization close to 50%. In the k-dimensional paradigm, it is not always possible to merge and re-split nodes as in the 1-dimensional case of the B+tree, because the notions of the "next" and "previous" sibling nodes cannot be defined. Redistribution of entries among nodes is much more complicated, and, depending on the index at hand, involves complicated updates on the corresponding index terms of the participating nodes.

B. KDB-tree and hB-tree

In both the KDB-tree and the hB-tree, data nodes, i.e., leaf level index nodes, contain the k-dimensional points or data terms for those points (in the case of secondary indexes). In a way analogous to the B+tree, when a data node becomes overfull because of insertions of new points, it has to be split. After the split we end up with two data nodes, the initial one occupying the same disk page and a new one occupying a new disk page. This process is repeated continuously, every time a data node becomes overfull.

The KDB-tree splits data nodes always using a single attribute, thus all data nodes are hyper-rectangles (or bricks). Also, internal nodes, i.e., index nodes above the leaf level, are split either at the root of their internal kd-tree, thus by a hyperplane, or by using some other splitting attribute to achieve balanced splits at the cost of downward propagation of splits. The KDB-tree does not have any guarantees on node storage utilization.

The hB-tree is an improvement of the KDB-tree, since it guarantees an average node storage utilization of 67% by splitting nodes at a 1:2 ratio in the worst case (compare this with B+tree's 1:1 ratio). This can be achieved both in the internal nodes that contain index terms in the form of kd-trees and in the data nodes that contain data entries. In Lomet and Salzberg (1990), it is shown that a 1:2 split ratio is always possible. In internal nodes, an appropriate kdsubtree is extracted from the overfull node. In data nodes, it may be necessary to use more than one attributes to achieve such a split. The overfull node and the newly extracted node can be hyper-rectangles from which smaller hyper

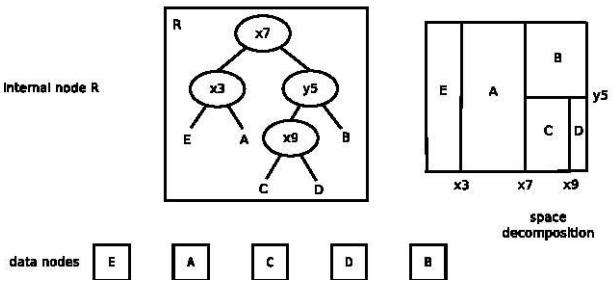

Figure 1: An example hB-tree

rectangles have been extracted, thus the name holey-Brick-tree (hB-tree). -

In Figure 1, an hB-tree with two levels is shown. It contains 5 data nodes and an internal node R, that is the root of the hB-tree. R contains the index terms for its 5 children in the form of a kd-tree. Let's assume that the last split that happened was the one that extracted node E from node A. Before the split, kdtree node x7 in R had a left pointer to data node A. The index term consisting of kd-tree node x3 (namely, the attribute and attribute value that were used to split A and extract E) was merged in the kd-tree of R to describe the new space decomposition.

C. Splitting attribute selection policy

When splitting overfull data nodes the goal is (a) to minimize the cost of future range queries, and, (b) to maximize average node storage utilization. The second goal, although it creates smaller trees, it may conflict with the first goal. This is because, good node storage utilization can lead to poor k-dimensional space partitioning.

For good space partitioning, the obvious approach is to split the space of the data node in half along the longest edge (attribute) and to ignore the distribution of the points in the node. The resulting data nodes will index the same amount of space and will have regular shapes, i.e., edges of similar lengths. Thus, they will have the same probability of receiving new insertions of points or of being visited by subsequent range queries. The drawback of this approach is that we may end up having nodes with low or zero storage utilization. Alternatively, one may choose to achieve the best possible node storage utilization by always trying to achieve 1:1 point splits, at the cost of bad space partitioning.

In Outsios and Evangelidis (2010), we experimented with various splitting attribute selection policies for data nodes. In this paper we choose the policy that uses the best attribute for even point split and best possible space split. This works as follows. Choose the attribute that achieves the most even point split. In case of ties, choose the attribute that splits along the longest edge. By splitting along the longest edge, we favor hyper-rectangles that are as close as possible to hyper-cubes. The goal is to minimize the cost of range queries by avoiding peculiar shaped subspaces.

III. NEW SPLITTING APPROACH

We focus our attention to the leaf level of the index, since the data nodes are the majority of the nodes in the tree index.

When splitting data nodes we should aim at:

1. Splitting the data node as evenly as possible both in terms of points (to improve node storage utilization) and space (to improve range query performance). We achieve this by using the best attribute for even point split, and, at the same time, the best possible space split.

2. Posting the most compact index term possible to minimize the number of the internal index nodes. We achieve this by always performing hyperplane splits. Thus, we minimize the size of the index terms, and the resulting data nodes are always hyper-bricks.

To further improve the performance under non-uniform data insertion patterns, we propose a new way for dealing with overfull nodes. We first define the terms paired and single data nodes. To illustrate the term paired, we examine Figure 1. Data nodes E and A are paired since they are pointed by the same kd-tree node in the kd-tree of their parent R. Data nodes C and D are also paired, whereas, data node B is considered to be single.

The idea is to exploit the structure of the kd-tree in the internal index nodes right above the data nodes, in order to identify data nodes that can re-distribute their contents. Following such an approach, leads to delayed splits of overfull nodes until their paired node becomes overfull, too.

IV. EXPERIMENTAL EVALUATION

We tested our splitting approach against the standard splitting algorithm of the hB-tree. The tested variations were the following:

m1 Original hB-tree data node splitting algorithm: do not use any redistribution scheme. When a node becomes full, split it.

m2 Redistribute among paired nodes and eventually split: Paired nodes delay splitting by redistributing their contents with their paired sibling. Only when both nodes in a pair become full, the one that overflows, splits. Single nodes split when full.

Parameter	Values
attribute value range	[0, 1]
k=dimensionality	2 – 15
database size	100K points
DNS=data node sizes	10 – 100 points
INS=internal node sizes	5 – 50 kd-tree nodes
insertion patterns	uniform and block
range query space selectivity	0.01%

Table 1: Experiment parameters and values

Table 1 lists the parameters of the experiments and the values we used.

We used relatively small node sizes in order to build hB-trees with many levels and stress our algorithms. Notice that the data node size is affected by the dimensionality of the points, i.e., 10 2-dim points occupy 1/10th of the space occupied by 10 20-dim points.

Also, we assumed that there are no deletions of points. The index only grows in time. To achieve the desired range query windows, we generated hypercube queries that covered 0.01% of the k-dimensional space. Thus, for 100K uniformly distributed points, we expect the query window to contain 10 points.

In Table 2, we compare the splitting methods m1 and m2 on average data node storage utilization and range query efficiency (in terms of average number of visited pages to answer 100 random queries with 0.01% selectivity).

We use 100K uniformly distributed points with uniform insertion pattern and we vary dimensionality. Using small node sizes we build trees with 7 levels. We observe that m1 and m2 have comparable node storage utilization across dimensions, but, as expected, m2 builds slightly smaller trees, i.e., with fewer data nodes. Thus, m2 performs slightly better in terms of average data node storage utilization and average number of accessed nodes per range query.

Next, we focus on node storage utilization when using a block insertion pattern, i.e., when a random data node is chosen and all incoming points target that node until the node splits. Then, another random data node is chosen, and so on. In this experiment, we fix the dimensionality to 6 and we vary the node sizes. Table 3, demonstrates that m1 achieves a node storage utilization slightly above 50%, whereas, m2 achieves very good average data node storage utilization. Thus, the average number of accessed nodes per range query is considerably lower.

V. CONCLUSION

We proposed a new data node splitting method for the hB-tree or the KDB-tree. Since data nodes comprise the majority of nodes in a tree index, higher data node storage utilization can improve search performance. There is a need for indexes in medium dimensionality that can efficiently answer kNN queries.

splitting method	k	INS	DNS	nodes per tree level	utilization (%)	average nodes accessed
m1	2	5	10	1,2,8,36,173,730,3211,13946	71,71	4,64
m2	2	5	10	1,2,8,37,160,720,3118,13606	73,5	4,65
m1	3	5	10	1,2,9,42,173,743,3212,13956	71,65	9,72
m2	3	5	10	1,2,9,40,166,719,3157,13591	73,58	9,39
m1	10	5	10	1,2,8,33,163,757,3284,13929	71,79	877,38
m2	10	5	10	1,2,8,33,153,701,3144,13599	73,53	860,81
m1	15	5	10	1,2,8,39,184,808,3396,14171	70,57	14005,87
m2	15	5	10	1,2,9,39,178,794,3364,14183	70,51	13859,64

Table 2: Node storage utilization and query efficiency per splitting method for uniform data and insertion pattern and varying dimensionality

splitting method	k	INS	DNS	nodes per tree level	utilization (%)	average nodes accessed
m1	6	5	10	1,4,15,57,231,974,3954,16666	54,55	1254,37
m2	6	5	10	1,2,8,35,151,640,2821,12352	73,6	1038,52
m1	6	25	50	1,15,219,3920	51,03	447,47
m2	6	25	50	1,8,156,2841	70,41	372,11
m1	6	50	100	1,2,62,1980	50,52	284,84
m2	6	50	100	1,43,1422	70,34	243,26

Table 3: Node storage utilization per splitting method for uniform data, block insertion pattern and varying node sizes

So, we examined whether our splitting method improves the performance of the above mentioned indexes.

We defined the notion of paired data nodes, and we used this notion to propose the new splitting method. Our experiments show that redistribution works really well and improves data node storage utilization and range query performance.

REFERENCES

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 Proceedings of the 1990 ACM SIGMOD international conference on Management of data, SIGMOD '90, pages 322–331, New York, NY, USA. ACM.

Berchtold, S., Bohm, C., and Kriegal, H.-P. (1998). The pyramid-technique: towards breaking the curse of dimensionality. In Proceedings of the 1998 ACM SIGMOD international conference on Management of data, SIGMOD '98, pages 142–153, New York, NY, USA. ACM.

Comer, D. (1979). Ubiquitous b-tree. ACM Comput. Surv., 11:121–137.

Evangelidis, G., Lomet, D., and Salzberg, B. (1997). The hb†-tree: a multi-attribute index supporting concurrency, recovery and node consolidation. The VLDB Journal, 6:1–25.

Knuth, D. E. (1973). The Art of Computer Programming, Vol 3, Sorting and Searching. Addison-Wesley Publ. Co., Reading, MA, USA.

Lomet, D. B. and Salzberg, B. (1990). The hb-tree: a multiattribute indexing method with good guaranteed performance. ACM Trans. Database Syst., 15:625–658.

Outsios, E. and Evangelidis, G. (2010). Achieving optimal average data node storage utilization in k-dimensional point data indexes. In Proceedings of the 5th International Scientific Conference, eRA: The Contribution of Information Technology to Science, Economy, Society and Education, Piraeus, Greece.

Robinson, J. T. (1981). The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In Proceedings of the 1981 ACM SIGMOD international conference on Management of data, SIGMOD '81, pages 10–18, New York, NY, USA. ACM.

Yao, A. C.-C. (1978). On random 23 trees. Acta Informatica, 9:159–170. 10.1007/BF00289075.

Zhou, P. and Salzberg, B. (2008). The hb-pi* tree: An optimized comprehensive access method for frequent-update multi-dimensional point data. In Proceedings of the 20th international conference on Scientific and Statistical Database Management, SSDBM '08, pages 331–347, Berlin, Heidelberg. Springer-Verlag.