ACHIEVING OPTIMAL AVERAGE DATA NODE STORAGE UTILIZATION IN K-DIMENSIONAL POINT DATA INDEXES

EVANGELOS OUTSIOS

Department of Applied Informatics, University of Macedonia
156 Egnatia Str., Thessaloniki, 54006, Greece

GEORGIOS EVANGELIDIS

Department of Applied Informatics, University of Macedonia
156 Egnatia Str., Thessaloniki, 54006, Greece

Indexing of k-dimensional point data is becoming again a hot research topic because of the need to efficiently index and retrieve high dimensional vectors (points) in data mining applications. The most common query on such vectors is kNN searching, which is a variation of range searching. Most multidimensional indexes for point data follow the paradigm of the ubiquitous B+tree and store data entries at the leaf level of the index (data nodes). Since this level naturally occupies the majority of nodes in a multidimensional index tree, it is crucial that an index structure achieves the best possible average storage utilization regardless of data distribution and order of data insertion. An additional conflicting goal is the minimization of the index term that is posted at the levels above when data nodes are split. In this paper we revisit data node splitting techniques for point access methods like the KDB-tree, hB-tree, and, in general, any index that stores point data at its leaf level nodes and splits them so that no overlapping subspaces are created at the leaf level. We experiment with various splitting techniques that produce the minimum index term for posting but differ in the shape of the resulting nodes and the average storage utilization. We also test our splitting techniques using uniform and skewed data distributions. The comparison is on the average data node storage utilization and the efficiency of range query searches.

1. Introduction

Lately, there is an increased interest in access methods for multidimensional points or vectors (point access methods - PAMs). This is explained by the fact that data mining applications need to manipulate and analyze vast quantities of multi-dimensional vectors, especially when dealing with time-series data. The well-known problem of the "curse of dimensionality", states that above 8 to 16 dimensions and depending on the dataset at hand, exhaustive search of the dataset is faster than using a PAM for the purpose of range or k-NN queries [1].

Moreover, vectors in data mining applications can have hundreds of dimensions that can be correlated and, thus, it is necessary to reduce their dimensionality in order to reduce the volume of data without losing much

information. Researchers in the data mining field use various dimensionality reduction techniques and usually reduce the dimensionality of vectors down to 6 to 20 dimensions. Thus, PAMs that index the multidimensional space without creating overlapping subspaces and work well in low to medium dimensions are an attractive choice for indexing such large point datasets.

In this paper, we assume that our data records are multi-dimensional points and that the index of choice is a PAM, e.g., the hB-tree [3,4] or the KDB-tree [5], etc., that stores points in the leaf nodes and splits them in non-overlapping regions. We choose to split data nodes using hyper-planes, i.e., a single attribute, since this approach requires the smallest index term to describe the split. Only the splitting attribute and its value are required to be posted at the index level above (the parent of the overfull node) to describe the split. We experiment with various splitting techniques and report on their performance in terms of the average data node storage utilization and the efficiency of range query searches.

In Section 2, we discuss the problem of data node splitting and how it can affect the performance of an index structure. In Section 3, we present three splitting techniques, and, in Section 4, we describe the experiments we conducted on the performance of the splitting techniques. We conclude the paper in Section 5.

2. Data node splitting

Data nodes contain the records (multidimensional points) of the database. In a way analogous to the B+tree [2], 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 is overfull.

In Figure 1, we demonstrate a simple example where two splits took place and we got three data nodes occupying three disk pages (we assume 7 points per data node). We started with an initial data node d1, which became overfull and was split to two data nodes d1 and d2. Next, d2 became overfull and was split to d2 and d3. However, we can observe that the total number of points residing in the three data nodes is 10 and can easily fit to two data nodes only. We could have avoided the second split and end up with only two data nodes to store our points, had we planned the first split more carefully.

In Figure 2, we see that because we performed the first split in a different manner, the second split did not happen and the same number of points as in the case of Figure 1 was accommodated in two data nodes.

Thus, data node splitting, although it seems to be an easy and trivial process, is a very important issue and requires careful planning. By avoiding unnecessary splits, we:

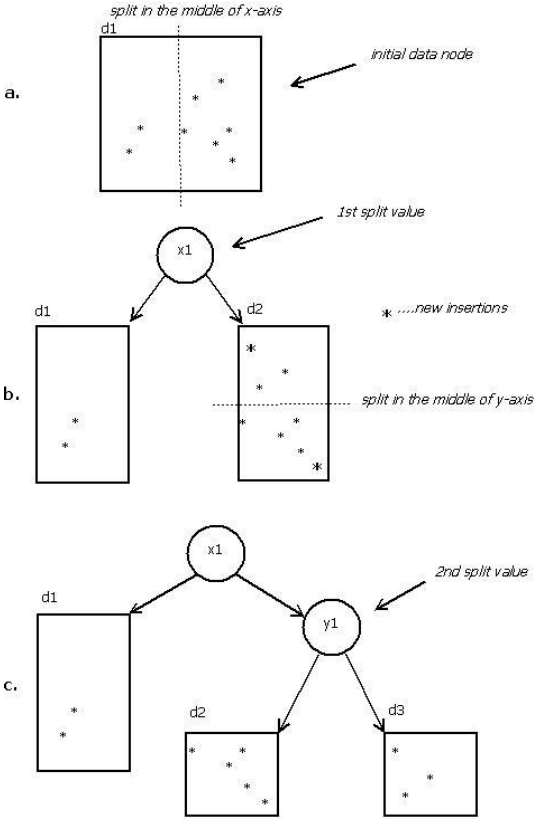

Figure 1: Ineffective data node splitting

- reduce the total number of data nodes and hence the disk pages needed,
- increase data node storage utilization,
- reduce the height of the tree, since less splits means less information posted above,
- make exact match query faster, since the tree height is smaller,
- make range queries more effective, since the points searched are accommodated to fewer nodes and less disk pages are accessed.

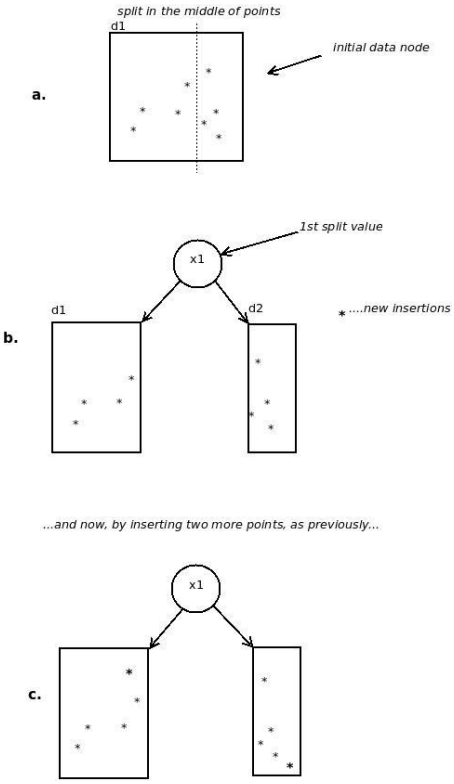

Figure 2: Effective data node splitting

2.1. *Splitting criteria*

Let's discuss the criteria by which we decide to split a data node. To better follow the discussion and visualize the problem we examine the 2-dimensional case.

When we have an overfull data node the aim is to carry out the split in a way that minimizes the cost of future queries. One approach is to split the space of the data node in half along the longest edge and to ignore the distribution of the points in the node [1]. The two resulting data nodes will refer to the same amount of space and will have regular shapes, i.e., edges of similar lengths. This means that they will have the same probability of receiving new insertions of points or of being visited by subsequent range queries. The problem is that by following this approach we may end up having nodes with low or zero storage utilization.

A variation of the above approach is to split the space in half using the attribute that achieves the best point split, i.e., the one closer to a 1/2-1/2 split. Again, it may be the case that neither x nor y lead to a good point split.

An alternative approach is to concentrate on the distribution of points. With this approach we always try to achieve an even point split without guaranteeing an even space split. For example, in Figure 3 we could choose the median of attribute x to evenly split the points of the data node.

We can improve the above approach by choosing the median of the attribute that achieves the best space split, or better relax the condition that data nodes are split at the median of the chosen attribute. For example, in Figure 4, if we move the value of the splitting attribute x to the right we can achieve an even point and space split at the same time.

In the following sections, we describe three splitting techniques and show the results from the experiments we carried out. We emphasize on the advantages or disadvantages of each technique applied to random (uniformly distributed) data and highly skewed data.

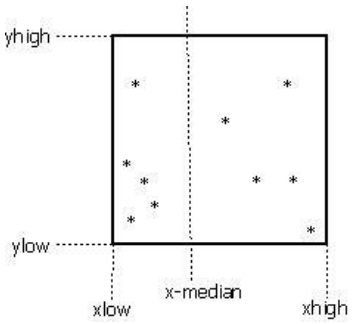

Figure 3: Splitting using the median

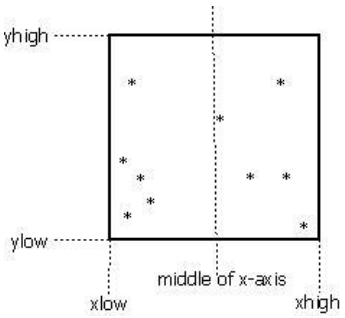

Figure 4: Splitting points and space evenly

3. Presentation of tested data node splitting techniques

3.1. *Round robin attribute for even point split*

This is the simplest technique. We perform an even point split on the first data node using the first attribute. When either of the two data nodes that resulted from the first split becomes overfull, it is split using the next attribute in turn, and so on.

For uniformly distributed in space points, this splitting process achieves good storage utilization since nodes are always split at a 1/2-1/2 ratio. It also achieves good space partitioning since data nodes are always split along the

longest edge. The performance could be very poor for skewed data. Another drawback of the technique is that every data node must store the bookkeeping information of the attribute that should be used for the following split.

Figure 5 demonstrates a scenario where this technique has very poor performance. We assume that a node can hold only 4 points, and the insertion of the fifth point causes a split. In the end, we have inserted 10 points, we have made 4 splits and we have 5 data nodes most of which have very low storage utilization. The problem here was that when in a data node most of the points had the same x attribute value we were forced to split using x instead of y that could achieve a better point split. When having such non-uniform point distributions the higher the dimensionality the worst the problem gets, i.e., we may end up with data nodes with very low storage utilization.

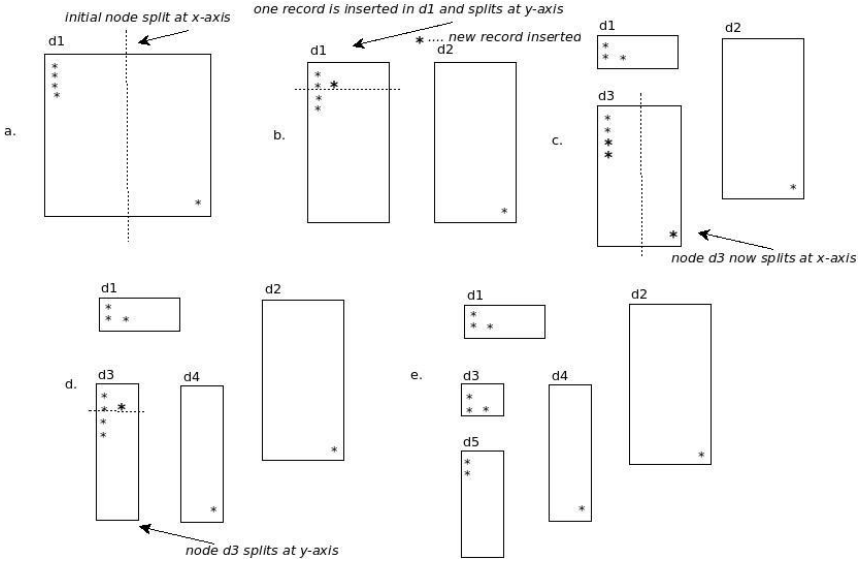

Figure 5: Round-robin splitting with poor performance

3.2. *Best attribute for even point split and best possible space split*

This technique is both point and space oriented and favors even point splits. We choose as the splitting attribute the one that achieves the best point split, i.e., closest to a 1/2-1/2 ratio. If more than one attribute qualifies, we pick the one that achieves the best space split, i.e., closest to a 1/2-1/2 ratio. Again, if more than one attribute qualifies, we choose the one that splits along the longest edge. This technique guarantees high storage utilization and good space partitioning. It

works well for random data but for non-uniform data there is no guarantee that the space partitioning will be even. In Figure 6, we see a scenario where attribute y will be chosen to perform the split of node d1.

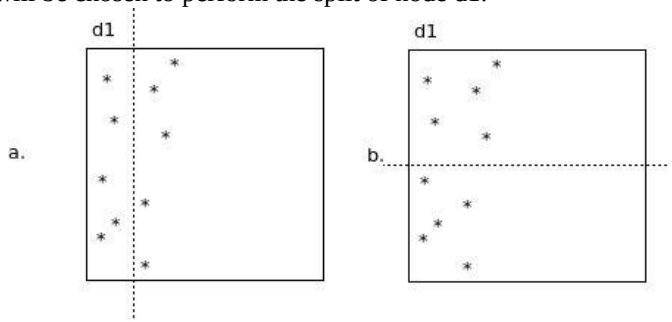

Figure 6: Data node where attribute y splits evenly both points and space

In Figure 7, we see a scenario where x should be chosen as the splitting attribute. Here, both x and y achieve the best point and space splits, but x splits along the longest edge 1.

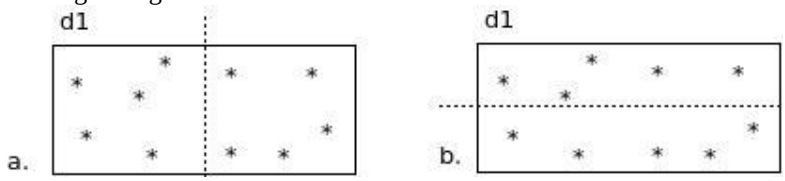

Figure 7: Both x and y achieve even point and space split, but x splits the longest edge

3.3. *Best attribute for even space split and a minimum 1/3-2/3 point split*

This method is also both space and point oriented, but favors even space splits. We choose an attribute that achieves an even space split and a point split of at least 1/3-2/3. As shown in [4], such a splitting ratio guarantees very good storage utilization. We order the attributes according to the length of the edge they split and we choose the best one. If no such attribute exists, we choose the attribute that can achieve a 1/3-2/3 point split by compromising the evenness of the space split. For example, in Figure 8 we see a scenario where y should be chosen as the splitting attribute.

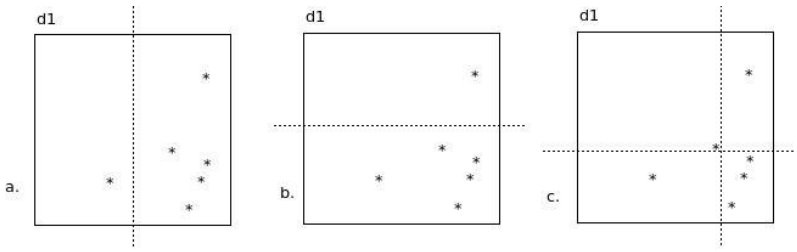

Figure 8: y compromises space split evenness to achieve a 1/3-2/3 point split

4. Experiments

We tested the three splitting techniques using uniform and highly skewed computer generated 2d points. We varied the size of the data node to hold 25, 50 and 100 points and the size of the dataset to be 10K, 100K and 1M points. We use the notation t1, t2, and t3 for the three techniques, i.e., round-robin best point split, best point split, and best space split, respectively.

In Table 1, we observe that t1 and t2 have almost identical performance regarding data node storage utilization when using uniform data. Utilization is about 70% regardless of dataset and node sizes. On the other hand, t3 gets worse as the dataset and node sizes increase. This is explained by the fact that t3 favors even space partitioning over even point partitioning. In Table 2, where the data is skewed, we observe similar behavior for t1 and t2, but now t3 has significantly improved performance – almost comparable to the one of t1 and t2 for large dataset and node sizes.

Table 1: Uniform data: data node storage utilization for various dataset and page sizes

	25			**50**			**100**		
	10K	**100K**	**1M**	**10K**	**100K**	**1M**	**10K**	**100K**	**1M**
t1	69,81	70,08	70,16	70,67	69,78	69,82	70,92	68,49	69,57
t2	70,05	70,18	70,29	72,20	69,93	69,78	73,53	68,63	69,41
t3	72,86	69,35	66,45	74,63	70,00	63,76	78,13	69,93	61,63

Table 2: Skewed data: data node utilization for various dataset and page sizes

	25			**50**			**100**		
	10K	**100K**	**1M**	**10K**	**100K**	**1M**	**10K**	**100K**	**1M**
t1	69,69	70,05	70,13	71,17	69,91	69,75	68,03	69,20	69,75
t2	70,42	70,68	70,31	70,67	69,88	69,80	70,42	69,44	69,48
t3	66,78	67,81	68,11	67,34	67,02	67,89	62,50	65,49	67,59

Next we conducted some range query experiments. We chose 100 random queries with 1% space selectivity and we report the percent of the data pages visited to answer these queries. For uniform data one expects about 1% for the pages to be visited. All three techniques approach this number for large datasets (or large trees). Since t3 favors even space partitioning, it slightly outperforms the other two point partitioning techniques regardless of dataset and node sizes (see Table 3). Finally, for skewed data, t2 that favors even point partitioning and at the same time tries to achieve good space partitioning, clearly outperforms the other two techniques regardless of dataset and node sizes (see Table 4).

Table 3: Uniform data: range query performance for various dataset and page sizes

	25			**50**			**100**		
	10K	**100K**	**1M**	**10K**	**100K**	**1M**	**10K**	**100K**	**1M**
t1	1,95	1,32	1,10	2,37	1,38	1,12	2,96	1,50	1,15
t2	1,91	1,26	1,10	2,38	1,37	1,12	2,99	1,49	1,14
t3	1,83	1,22	1,07	2,21	1,34	1,10	3,00	1,48	1,13

Table 4: Skewed data: range query performance for various dataset and page sizes

	25			**50**			**100**		
	10K	**100K**	**1M**	**10K**	**100K**	**1M**	**10K**	**100K**	**1M**
t1	1,94	1,33	1,19	2,34	1,38	1,11	3,08	1,52	1,15
t2	1,63	1,18	1,07	2,00	1,25	1,07	2,62	1,36	1,11
t3	1,87	1,54	1,31	2,11	1,61	1,40	2,68	1,77	1,52

5. Conclusions

We presented three data node splitting techniques for point access methods that split space in non-overlapping regions. The techniques differ in the way they split overfull data nodes. They choose the splitting attribute and its value in order to achieve simultaneously even point and space splits. Since this is impossible to achieve unless data is uniformly distributed in space, we are interested in the performance of the various techniques when data is highly skewed. Our experiments showed that a technique that favors even point splits and tries to achieve good space splits is best when data is uniform. On the other hand, a technique that favors even space splits and tries to achieve acceptable point splittings, clearly outperforms all other techniques at the price of slightly reduced data node storage utilization.

We plan to further improve our data node splitting techniques and test their performance in higher dimensions and with non-uniform data from real world scientific applications.

References

1. Berchtold, S., Böhm, C., and Kriegel, H.P. (1998). The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proc SIGMOD*, pp. 142-153.
2. Comer, D. (1979). The Ubiquitous B-Tree. *ACM Comput. Surv. (CSUR)*, 11(2), pp. 121-137.
3. Evangelidis, G., Lomet, D.B., and Salzberg, B. (1997). The hBπ-Tree: A Multi-Attribute Index Supporting Concurrency, Recovery and Node Consolidation. *VLDB J.*, 6(1), pp. 1-25.
4. Lomet, D.B. and Salzberg, B. (1990). The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Trans. Database Syst. (TODS)*, 15(4), pp. 625-658.
5. Robinson, J.T. (1981). The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. In *Proc SIGMOD*, pp. 10-18.