X-tree

Daniel Keim[a], Benjamin Bustos[b], Stefan Berchtold[c], and Hans-Peter Kriegel[d]

[a] Department of Computer and Information Science, University of Konstanz
[b] Department of Computer Science, University of Chile
[c] stb ag, Germany
[d] Institute for Computer Science, Ludwig-Maximilians-University Munich

SYNONYMS

Extended node tree

DEFINITION

The *X-tree* (eXtended node tree) [1] is a spatial access method [2] that supports efficient query processing for high-dimensional data. It supports not only point data but also extended spatial data. The X-tree provides *overlap-free split* whenever it is possible without allowing the tree to degenerate; otherwise, the X-tree uses extended variable size directory nodes, so-called supernodes. The X-tree may be seen as a hybrid of a linear array-like and a hierarchical R-tree-like directory.

HISTORICAL BACKGROUND

The R-tree [3] and the R*-tree [4], spatial access methods with a hierarchically structured directory that use minimum bounding rectangles (MBRs) as page regions, have primarily been designed for the management of spatially extended, two-dimensional objects, but have also been used for high-dimensional point data. Empirical studies, however, show a deteriorated performance of these spatial access methods for high-dimensional data. The major problem of these index structures in high-dimensional spaces is the overlap between MBRs. In contrast to low-dimensional spaces, there is only a few degrees of freedom for splits in the directory. In fact, in most situations there is only a single good (overlap-free) split axis. An index structure that does not use this split axis will produce highly overlapping MBRs in the directory and thus show a deteriorated performance. Unfortunately, this specific split axis might lead to unbalanced partitions. In this case, a split should be avoided to prevent underfilled nodes.

It is well established that in low-dimensional spaces the most efficient organization of the directory is a hierarchical organization. The reason is that the selectivity in the directory is very high which means that, for example for point queries, the number of required page accesses directly corresponds to the height of the tree. This, however, is only true if there is no overlap between directory rectangles which is very likely for low-dimensional data. It is also reasonable, that for very high dimensionality a linear organization of the directory is more efficient. The reason is that due to the high overlap, most of the directory if not the whole directory has to be searched anyway. If the whole directory has to be searched, a linearly organized directory needs less space and may be read much faster from disk than a block-wise reading of the directory. For medium dimensionality, an efficient organization of the directory would probably be partially hierarchical and partially linear. The problem is to dynamically organize the tree such that portions of the data which would produce high overlap are organized linearly and those which can be organized hierarchically without too much overlap are dynamically organized in a hierarchical form.

The X-tree is directly designed for the managment of high-dimensional objects and based on the analysis of

problems arising in high-dimensional data spaces. It extends the R*-tree by two concepts: overlap-free split according to a split history, and supernodes with an enlarged page capacity. The algorithms used in the X-tree are designed to automatically organize the directory as hierarchical as possible, resulting in a very efficient hybrid organization of the directory.

SCIENTIFIC FUNDAMENTALS

It has been observed experimentally that in high-dimensional spaces portion of the data space covered by more than one MBR in an R*-tree quickly approaches the whole data space. This is due to the criteria used by the R*-tree to split nodes, which also aim at minimizing the volume of the resulting MBRs. The high amount of overlap between MBRs means that, for any similarity query, at least two subtrees must be accessed in almost every directory node, thus reducing the efficiency of the index structure.

To avoid this problem, the X-tree maintains the history of data page splits of a node in a binary tree. The root of the *split history tree* contains the dimension where an overlap-free split is guaranteed (that is a dimension according to which all MBRs in the node have been split previously). When a directory node overflows, this dimension is used to perform the split. However, the overlap-free split may be unbalanced, i.e., one of the nodes may be almost full and the other one may be underfilled, thus decreasing the storage utilization in the directory. The X-tree does not split in this case but creates a *supernode* instead. A supernode is basically an enlarged directory node, which can store more entries than normal nodes. In this way, the unbalanced split is avoided and a good storage utilization is mantained, at the cost of diminishing some of the discriminative power of the index.

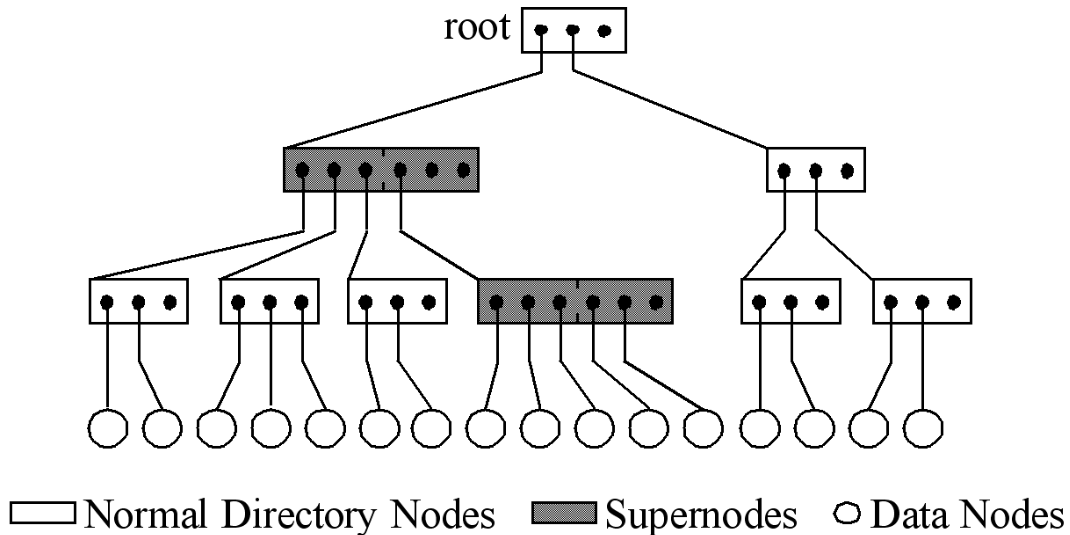

Figure 1: Structure of the X-tree

The overall structure of the X-tree is presented in Figure 1. The data nodes of the X-tree contain rectilinear MBRs together with pointers to the actual data objects, and the directory nodes contain MBRs together with pointers to sub-MBRs (see Figure 2). The X-tree consists of three different types of nodes: data nodes, normal directory nodes, and supernodes. Supernodes are large directory nodes of variable size (a multiple of the usual block size). The basic goal of supernodes is to avoid splits in the directory that would result in an inefficient directory structure. The alternative to using larger node sizes are highly overlapping directory nodes which would require to access most of the child nodes during the search process. This, however, is less efficient than linearly scanning the larger supernode.

Note that the X-tree is completely different from an R-tree with a larger block size since the X-tree only consists of larger nodes where actually necessary. As a result, the structure of the X-tree may be rather heterogeneous as

Figure 2: Structure of a directory node in the X-tree

indicated in Figure 1. Due to the fact that the overlap is increasing with the dimension, the internal structure of the X-tree is also changing with increasing dimension. In Figure 3, three examples of X-trees containing data of different dimensionality are shown. As expected, the number and size of supernodes increases with the dimension. For generating the examples, the block size has been artificially reduced to obtain a drawable fanout. Due to the increasing number and size of supernodes, the height of the X-tree is decreasing with increasing dimension.

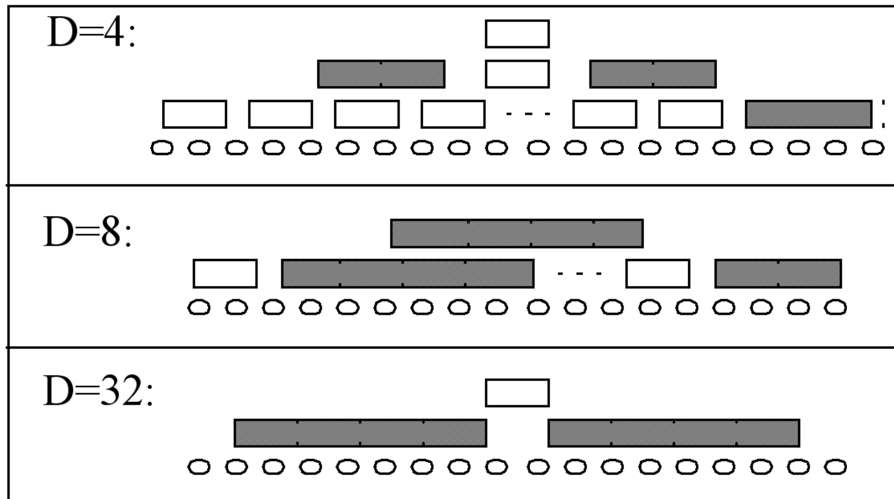

Figure 3: Various shapes of the X-tree in different dimensions

The most important algorithm of the X-tree is the insertion algorithm. It determines the structure of the X-tree which is a suitable combination of a hierarchical and a linear structure. The main objective of the algorithm is to avoid splits which would produce overlap. The algorithm first determines the MBR in which to insert the data object and recursively calls the insertion algorithm to actually insert the data object into the corresponding node. If no split occurs in the recursive insert, only the size of the corresponding MBRs has to be updated. In case of a split of the subnode, however, an additional MBR has to be added to the current node which might cause an overflow of the node. In this case, the current node calls the split algorithm which first tries to find a split of the node based on the topological and geometric properties of the MBRs. Topological and geometric properties of the MBRs are for example dead-space partitioning, extension of MBRs, etc. The heuristics of the R*-tree [4] split algorithm are an example for a topological split to be used in this step. If the topological split however results in high overlap, the split algorithm tries next to find an overlap-minimal split which can be determined based on the split history.

For determining an overlap-minimal split of a directory node, one has to find a partitioning of the MBRs in the node into two subsets such that the overlap of the minimum bounding hyperrectangles of the two sets is minimal. In case of point data, it is always possible to find an overlap-free split, but in general it is not possible to guarantee that the two sets are balanced, i.e., have about the same cardinality. It is an interesting observation that an overlap-free split is only possible if there is a dimension according to which all MBRs have been split since otherwise at least one of the MBRs will span the full range of values in that dimension, resulting in some overlap.

For finding an overlap-free split, a dimension according to which all MBRs of a node S have been split previously

has to be determined. The split history provides the necessary information, in particular the dimensions according to which an MBR has been split and which new MBRs have been created by this split. Since a split creates two new MBRs from one, the split history may be represented as a binary tree, called the split tree. Each leaf node of the split tree corresponds to an MBR in S. The internal nodes of the split tree correspond to MBRs which do not exist any more since they have been split into new MBRs previously. Internal nodes of the split tree are labeled by the split axis that has been used; leaf nodes are labeled by the MBR they are related to. All MBRs related to leaves in the left subtree of an internal node have lower values in the split dimension of the node than the MBRs related to those in the right subtree.

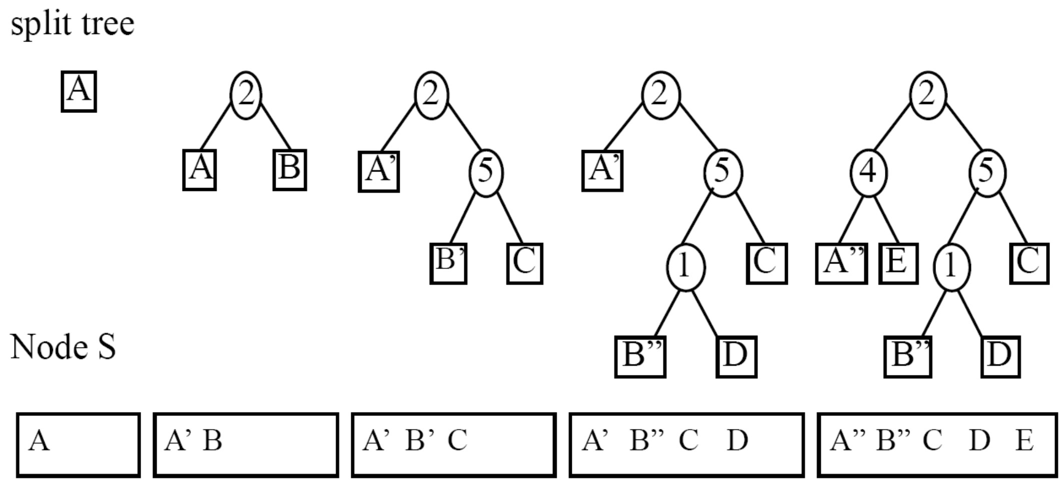

Figure 4: Example for the split history

Figure 4 shows an example for the split history of a node S and the respective split tree. The process starts with a single MBR A corresponding to a split tree which consists of only one leaf node labeled by A. For uniformly distributed data, A spans the full range of values in all dimensions. The split of A using dimension 2 as split axis produces new MBRs A and B. Note that A and B are disjoint because any point in MBR A has a lower coordinate value in dimension 2 than all points in MBR B. The split tree now has one internal node (marked with dimension 2) and two leaf nodes (A and B). Splitting MBR B using dimension 5 as split axis creates the nodes B and C. After splitting B and A again, the situation depicted in the right most tree of Figure 4 is reached, where S is completely filled with the MBRs A, B, C, D, and E.

One may find an overlap-free split if there is a dimension according to which all MBRs of S have been split. To obtain the information according to which dimensions an MBR X in S has been split, the split tree has to be traversed from the root node to the leaf that corresponds to X. For example, MBR C has been split according to dimensions 2 and 5, since the path from the root node to the leaf C is labeled with 2 and 5. Obviously, all MBRs of the split tree in Figure 4 have been split according to dimension 2, the split axis used in the root of the split tree. In general, all MBRs in any split tree have one split dimension in common, namely the split axis used in the root node of the split tree.

The partitioning of the MBRs resulting from the overlap-minimal split, however, may result in underfilled nodes which is unacceptable since it leads to a degeneration of the tree and also deteriorates the space utilization. If the number of MBRs in one of the partitions is below a given threshold, the split algorithm terminates without providing a split. In this case, the current node is extended to become a supernode of twice the standard block size. If the same case occurs for an already existing supernode, the supernode is extended by one additional block. Obviously, supernodes are only created or extended if there is no possibility to find a suitable hierarchical structure of the directory. If a supernode is created or extended, there may be not enough contiguous space on disk to sequentially store the supernode. In this case, the disk manager has to perform a local reorganization.

The algorithms to query the X-tree (point, range, and nearest neighbor queries) are similar to the algorithms used in the R*-tree since only minor changes are necessary in accessing supernodes. The delete and update operations are also simple modifications of the corresponding R*-tree algorithms. The only difference occurs in case of an underflow of a supernode. If the supernode consists of two blocks, it is converted to a normal directory node. Otherwise, that is if the supernode consists of more than two blocks, the size of the supernode is reduced by one block. The update operation can be seen as a combination of a delete and an insert operation and is therefore straightforward.

KEY APPLICATIONS

In many applications, indexing of high-dimensional data has become increasingly important. In multimedia databases, for example, the multimedia objects are usually mapped to feature vectors in some high-dimensional space and queries are processed against a database of those feature vectors [5]. This feature-based approach is taken in many other areas including CAD [6], 3D object databases [7], molecular biology (for the docking of molecules [8]), medicine [9], string matching and sequence alignment [10], and document retrieval [11]. Examples of feature vectors are color histograms [12], shape descriptors [13], Fourier vectors [14], text descriptors [15], etc. In some applications, the mapping process does not yield point objects, but extended spatial objects in a high-dimensional space [16]. In many of the mentioned applications, the databases are very large and consist of millions of data objects with several tens to a few hundreds of dimensions.

FUTURE DIRECTIONS

The feature-based approach has several advantages compared to other approaches for implementing similarity search. The extraction of features from the source data is usually fast and easily parametrizable, and metric functions for feature vectors, as the Minkowski distances, can also be efficiently computed. Novel approaches for computing feature vectors from a wide variety of unstructured data are proposed regularly. As in many practical applications the dimensionality of the obtained feature vectors is high, the X-tree is a valuable tool to perform efficient similarity queries in spatial databases.

CROSS REFERENCES

1. Spatial access method
2. Similarity search
3. Nearest-neighbor query

RECOMMENDED READING

[1] Berchtold, S., Keim, D., and Kriegel, H.-P. (1996). The X-tree: An index structure for high-dimensional data. In *Proc. 22th International Conference on Very Large Databases (VLDB'96)*, pages 28–39. Morgan Kaufmann.

[2] Böhm, C., Berchtold, S., and Keim, D. (2001). Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373.

[3] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proc. ACM International Conference on Management of Data (SIGMOD'84)*, pages 47–57. ACM Press.

[4] 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. ACM International Conference on Management of Data (SIGMOD'90)*, pages 322–331. ACM Press.

[5] Faloutsos, C. (1996). *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, Norwell, MA, USA.

[6] Berchtold, B., Keim, D., Kriegel, H.-P. (1997). Using extended feature objects for partial similarity retrieval. The VLDB Journal, 6(4):333–348.

[7] Bustos, B., Keim, D., Saupe, D., Schreck, T., and Vranić, D. (2005). Feature-based similarity search in 3D object databases. *ACM Computing Surveys*, 37(4):345–387.

[8] Shoichet B. K., Bodian D. L., Kuntz I. D. (1992). Molecular docking using shape descriptors. *Journal of Computational Chemistry*, 13(3):380–397.

[9] Keim, D. (1999). Efficient geometry-based similarity search of 3D spatial databases. In *Proc. ACM International Conference on Management of Data (SIGMOD'99)*, pages 419–430. ACM Press.

[10] Altschul S. F., Gish W., Miller W., Myers E. W., Lipman D. J. (1990). A basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

[11] Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Shawney H., Hafner J. (1994). Efficient Color Histogram Indexing. In *Proc. International Conference on Image Processing*, pages 66–70.

[13] Jagadish H. V. (1991). A retrieval technique for similar shapes. In *Proc. ACM International Conference on Management of Data (SIGMOD'91)*, pages 208–217. ACM Press.

[14] Wallace T., Wintz P. (1980). An efficient three-dimensional aircraft recognition algorithm using normalized fourier descriptors. *Computer Graphics and Image Processing*, 13:99–126.

[15] Kukich K. (1992). Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–440.

[16] Murase H., Nayar S. K. (1995) Three-dimensional object recognition from appearance-parametric eigenspace method. *Systems and Computers in Japan*, 26(8):45–54.