The X-tree:
An Index Structure for High-Dimensional Data

Stefan Berchtold
Daniel A. Keim
Hans-Peter Kriegel

Institute for Computer Science, University of Munich, Oettingenstr. 67, D-80538 Munich, Germany
{berchtol, keim, kriegel}@informatik.uni-muenchen.de

Abstract

In this paper, we propose a new method for indexing large amounts of point and spatial data in high-dimensional space. An analysis shows that index structures such as the R*-tree are not adequate for indexing high-dimensional data sets. The major problem of R-tree-based index structures is the overlap of the bounding boxes in the directory, which increases with growing dimension. To avoid this problem, we introduce a new organization of the directory which uses a split algorithm minimizing overlap and additionally utilizes the concept of supernodes. The basic idea of overlap-minimizing split and supernodes is to keep the directory as hierarchical as possible, and at the same time to avoid splits in the directory that would result in high overlap. Our experiments show that for high-dimensional data, the X-tree outperforms the well-known R*-tree and the TV-tree by up to two orders of magnitude.

1. Introduction

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 [Fal 94]. Similar approaches are taken in many other areas including CAD [MG 93], molecular biology (for the docking of molecules) [SBK 92], string matching and sequence alignment [AGMM 90], etc. Examples of feature vectors are color histograms [SH 94], shape descriptors [Jag 91, MG 95], Fourier vectors [WW 80], text descriptors [Kuk 92], etc. In some applications, the mapping process does not yield point objects, but extended spatial objects in high-dimensional space [MN 95]. 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. For querying these databases, it is essential to use appropriate indexing techniques which provide an efficient access to high-dimensional data. The goal of this paper is to demonstrate the limits of currently available index structures, and present a new index structure which considerably improves the performance in indexing high-dimensional data.

Our approach is motivated by an examination of R-tree-based index structures. One major reason for using R-tree-based index structures is that we have to index not only point data but also extended spatial data, and R-tree-based index structures are well suited for both types of data. In contrast to most other index structures (such as kdB-trees [Rob 81], grid files [NHS 84], and their variants [see e.g. SK 90]), R-tree-based index structures do not need point transformations to store spatial data and therefore provide a better spatial clustering.

Some previous work on indexing high-dimensional data has been done, mainly focussing on two different approaches. The first approach is based on the observation that real data in high-dimensional space are highly correlated and clustered, and therefore the data occupy only some subspace of the high-dimensional space. Algorithms such as Fastmap [FL 95], multidimensional scaling [KW 78], principal component analysis [DE 82], and factor analysis [Har 67] take advantage of this fact and transform data objects into some lower dimensional space which can be efficiently indexed using traditional multidimensional index structures. A similar approach is proposed in the SS-tree [WJ 96] which is an R-tree-based index structure. The SS-tree uses ellipsoid bounding regions in a lower dimensional space applying a different transformation in each of the directory nodes. The second approach is based on the observation that in most high-dimensional data sets, a small number

Proceedings ot the 22nd VLDB Conference
Mumbai (Bombay), India, 1996

of the dimensions bears most of the information. The TV-tree [LJF 94], for example, organizes the directory in a way that only the information needed to distinguish between data objects is stored in the directory. This leads to a higher fanout and a smaller directory, resulting in a better query performance.

For high-dimensional data sets, reducing the dimensionality is an obvious and important possibility for diminishing the dimensionality problem and should be performed whenever possible. In many cases, the data sets resulting from reducing the dimensionality will still have a quite large dimensionality. The remaining dimensions are all relatively important which means that any efficient indexing method must guarantee a good selectivity on all those dimensions. Unfortunately, as we will see in section 2, currently available index structures for spatial data such as the R*-tree[1] do not adequately support an effective indexing of more than five dimensions. Our experiments show that the performance of the R*-tree is rapidly deteriorating when going to higher dimensions. To understand the reason for the performance problems, we carry out a detailed evaluation of the overlap of the bounding boxes in the directory of the R*-tree. Our experiments show that the overlap of the bounding boxes in the directory is rapidly increasing to about 90% when increasing the dimensionality to 5. In subsection 3.3, we provide a detailed explanation of the increasing overlap and show that the high overlap is not an R-tree specific problem, but a general problem in indexing high-dimensional data.

Based on our observations, we then develop an improved index structure for high-dimensional data, the X-tree (cf. section 3). The main idea of the X-tree is to avoid overlap of bounding boxes in the directory by using a new organization of the directory which is optimized for high-dimensional space. The X-tree avoids splits which would result in a high degree of overlap in the directory. Instead of allowing splits that introduce high overlaps, directory nodes are extended over the usual block size, resulting in so-called supernodes. The supernodes may become large and the linear scan of the large supernodes might seem to be a problem. The alternative, however, would be to introduce high overlap in the directory which leads to a fast degeneration of the filtering selectivity and also makes a sequential search of all subnodes necessary with the additional penalty of many random page accesses instead of a much faster sequential read. The concept of supernodes has some similarity to the idea of oversize shelves [GN 91]. In contrast to supernodes, oversize shelves are data nodes which are attached to internal nodes in order to avoid excessive clipping of large objects. Additionally, oversize shelves are organized as chains of disk pages which cannot be read sequentially.

We implemented the X-tree index structure and performed a detailed performance evaluation using very large amounts (up to 100 MBytes) of randomly generated as well as real data (point data and extended spatial data). Our experiments show that on high-dimensional data, the X-tree outperforms the TV-tree and the R*-tree by orders of magnitude (cf. section 4). For dimensionality larger than 2, the X-tree is up to 450 times faster than the R*-tree and between 4 and 12 times faster than the TV-tree. The X-tree also provides much faster insertion times (about 8 times faster than the R*-tree and about 30 times faster than the TV-tree).

1. According to [BKSS 90], the R*-tree provides a consistently better performance than the R-tree [Gut 84] and R^+-tree [SRF 87] over a wide range of data sets and query types. In the rest of this paper, we therefore restrict ourselves to the R*-tree.

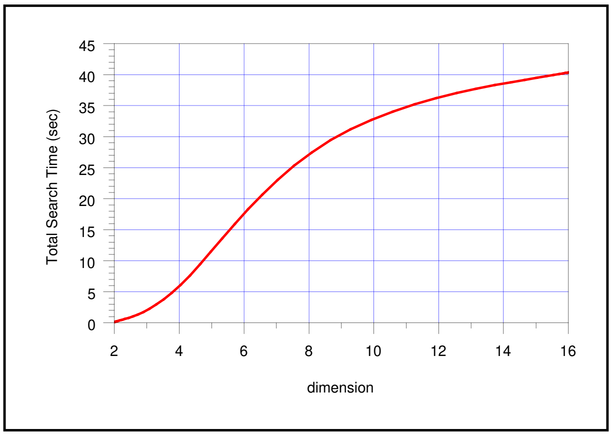


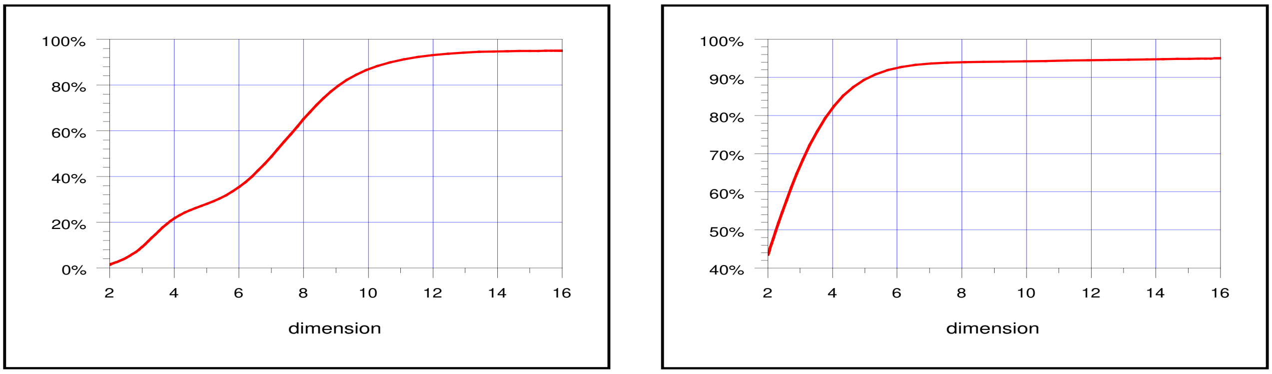
Figure 1: Performance of the R-tree Depending on the Dimension (Real Data)

2. Problems of (R-tree-based) Index Structures in High-Dimensional Space

In our performance evaluation of the R*-tree, we found that the performance deteriorates rapidly when going to higher dimensions (cf. Figure 1). Effects such as a lower fanout in higher dimensions do not explain this fact. In trying to understand the effects that lead to the performance problems, we performed a detailed evaluation of important characteristics of the R*-tree and found that the overlap in the directory is increasing very rapidly with growing dimensionality of the data. Overlap in the directory directly corresponds to the query performance since even for simple point queries multiple paths have to be followed. Overlap in the directory is a relatively imprecise term and there is no generally accepted definition especially for the high-dimensional case. In the following, we therefore provide definitions of overlap.

2.1 Definition of Overlap

Intuitively, overlap is the percentage of the volume that is covered by more than one directory hyperrectangle. This intuitive definition of overlap is directly correlated to the query performance since in processing queries, overlap of directory nodes results in the necessity to follow multiple paths, even for point queries.



a. Overlap (Uniformly Distributed Data)   b. Weighted Overlap (Real Data)

Figure 2: Overlap of R*-tree Directory Nodes depending on the Dimensionality

Definition 1a (Overlap)
The overlap of an R-tree node is the percentage of space covered by more than one hyperrectangle. If the R-tree node contains n hyperrectangles $\{R_1, \ldots R_n\}$, the overlap may formally be defined as

$$Overlap = \frac{\left\| \bigcup_{i, j \in \{1 \ldots n\}, i \neq j} (R_i \cap R_j) \right\|}{\left\| \bigcup_{i \in \{1 \ldots n\}} R_i \right\|} .^{1}$$

The amount of overlap measured in definition 1a is related to the expected query performance only if the query objects (points, hyperrectangles) are distributed uniformly. A more accurate definition of overlap needs to take the actual distribution of queries into account. Since it is impossible to determine the distribution of queries in advance, in the following we will use the distribution of the data as an estimation for the query distribution. This seems to be reasonable for high-dimensional data since data and queries are often clustered in some areas, whereas other areas are virtually empty. Overlap in highly populated areas is much more critical than overlap in areas with a low population. In our second definition of overlap, the overlapping areas are therefore weighted with the number of data objects that are located in the area.

Definition 1b (Weighted Overlap)
The weighted overlap of an R-tree node is the percentage of data objects that fall in the overlapping portion of the space. More formally,

$$WeightedOverlap = \frac{\left| \left\{ p \mid p \in \bigcup_{i, j \in \{1 \ldots n\}, i \neq j} (R_i \cap R_j) \right\} \right|}{\left| \left\{ p \mid p \in \bigcup_{i \in \{1 \ldots n\}} R_i \right\} \right|} .^{2}$$

1. $\|A\|$ denotes the volume covered by A.
2. $|A|$ denotes the number of data elements contained in A

In definition 1a, overlap occurring at any point of space equally contributes to the overall overlap even if only few data objects fall within the overlapping area. If the query points are expected to be uniformly distributed over the data space, definition 1a is an appropriate measure which determines the expected query performance. If the distribution of queries corresponds to the distribution of the data and is non-uniform, definition 1b corresponds to the expected query performance and is therefore more appropriate. Depending on the query distribution, we have to choose the appropriate definition.

So far, we have only considered overlap to be any portion of space that is covered by more than *one* hyperrectangle. In practice however, it is very important how many hyperrectangles overlap at a certain portion of the space. The so-called multi-overlap of an R-tree node is defined as the sum of overlapping volumes multiplied by the number of overlapping hyperrectangles relative to the overall volume of the considered space.

In Figure 3, we show a two-dimensional example of the overlap according to definition 1a and the corresponding multi-overlap. The weighted overlap and weighted multi-overlap (not shown in the figure) would correspond to the areas weighted by the number of data objects that fall within the areas.

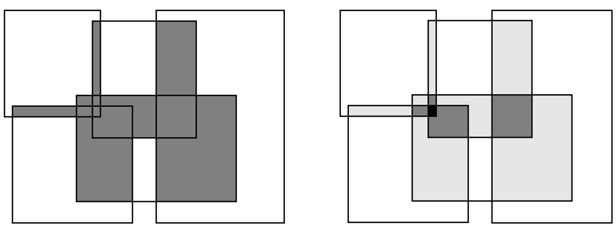


Figure 3: Overlap and Multi-Overlap of 2-dimensional data

2.2 Experimental Evaluation of Overlap in R*-tree Directories

In this subsection, we empirically evaluate the development of the overlap in the R*-tree depending on the dimensionality. For the experiments, we use the implementation of the R*-tree according to [BKSS 90]. The data used for the experiments are constant size databases of uniformly distributed and real data. The real data are Fourier vectors which are used in searching for similarly shaped polygons. The overlap curves presented in Figure 2 show the average overlap of directory nodes according to definition 1. In averaging the node overlaps, we used all directory levels except the root level since the root page may only contain a few hyperrectangles, which causes a high variance of the overlap in the root node.

In Figure 2a, we present the overlap curves of R*-trees generated from 6 MBytes of uniformly distributed point data. As expected, for a uniform distribution overlap and weighted overlap (definition 1a and 1b) provide the same results. For dimensionality larger than two, the overlap (cf. Figure 2a) increases rapidly to approach 100% for dimensionality larger than ten. This means that even for point queries on ten or higher dimensional data in almost every directory node at least two subnodes have to be accessed. For real data (cf. Figure 2b), the increase of the overlap is even more remarkable. The weighted overlap increases to about 80% for dimensionality 4 and approaches 100% for dimensionality larger than 6.

3. The *X-tree*

The X-tree (eXtended node tree) is a new index structure supporting efficient query processing of high-dimensional data. The goal is to support not only point data but also extended spatial data and therefore, the X-tree uses the concept of overlapping regions. From the insight obtained in the previous section, it is clear that we have to avoid overlap in the directory in order to improve the indexing of high-dimensional data. The X-tree therefore avoids overlap whenever it is possible without allowing the tree to degenerate; otherwise, the X-tree uses extended variable size directory nodes, so-called supernodes. In addition to providing a directory organization which is suitable for high-dimensional data, the X-tree uses the available main memory more efficiently (in comparison to using a cache).

The X-tree may be seen as a hybrid of a linear array-like and a hierarchical R-tree-like directory. It is well established that in low dimensions 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, e.g. for point queries, the number of required page accesses directly corresponds to the height of the tree. This, however,

MBR_0	$SplitHistory_0$	Ptr_0	– – –	MBR_{n-1}	$SplitHistory_{n-1}$	Ptr_{n-1}

Figure 4: Structure of a Directory Node

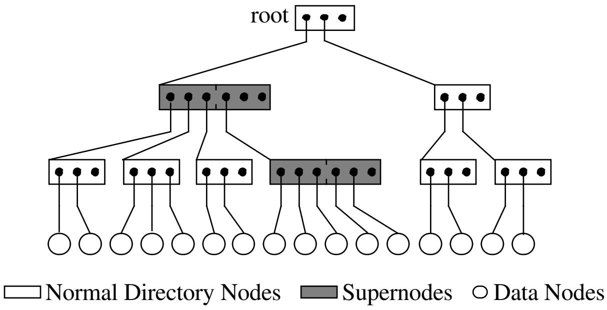


Figure 5: Structure of the X-tree

is only true if there is no overlap between directory rectangles which is the case for a low dimensionality. 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[1] 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 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.

3.1 Structure of the *X-tree*

The overall structure of the X-tree is presented in Figure 5. The data nodes of the X-tree contain rectilinear minimum bounding rectangles (MBRs) together with pointers to the actual data objects, and the directory nodes contain MBRs together with pointers to sub-MBRs (cf. Figure 5). 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 son nodes during the search process. This, however, is more inefficient 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 indicated in Figure 5. Due to the fact that the overlap is in-

1. In comparison to a hierarchically organized directory, a linearly organized directory only consists of the concatenation of the nodes on the lowest level of the corresponding hierarchical directory and is therefore much smaller.

creasing with the dimension, the internal structure of the X-tree is also changing with increasing dimension. In Figure 5, 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 which corresponds to the number of page accesses necessary for point queries is decreasing with increasing dimension.

Supernodes are created during insertion only if there is no other possibility to avoid overlap. In many cases, the creation or extension of supernodes may be avoided by choosing an overlap-minimal split axis (cf. subsection 3.3). For a fast determination of the overlap-minimal split, additional information is necessary which is stored in each of the directory nodes (cf. Figure 4). If enough main memory is available, supernodes are kept in main memory. Otherwise, the nodes which have to be replaced are determined by a priority function which depends on level, type (normal node or supernode), and size of the nodes. According to our experience, the priority function $c_t \cdot type + c_l \cdot level + c_s \cdot size$ with $c_t \gg c_l \gg c_s$ is a good choice for practical purposes. Note that the storage utilization of supernodes is higher than the storage utilization of normal directory nodes. For normal directory nodes, the expected storage utilization for uniformly distributed data is about 66%. For supernodes of size $m \cdot BlockSize$, the expected storage utilization can be determined as the average of the following two extreme cases: Assuming a certain amount of data occupies $X \cdot m$ blocks for a maximally filled node. Then the same amount of data requires $X \cdot \frac{m^2}{m-1}$ blocks when using a minimally filled node. On the average, a supernode storing the same amount of data requires $\left(X \cdot m + X \cdot \frac{m^2}{m-1}\right)/2 = X\left(\frac{m(2m-1)}{2m-2}\right)$ blocks. From that, we obtain a storage utilization of $m/\left(\frac{m(2m-1)}{2m-2}\right) = \frac{2 \cdot m - 2}{2 \cdot m - 1}$ which for large m is considerably higher than 66%. For m=5, for example, the storage utilization is about 88%.

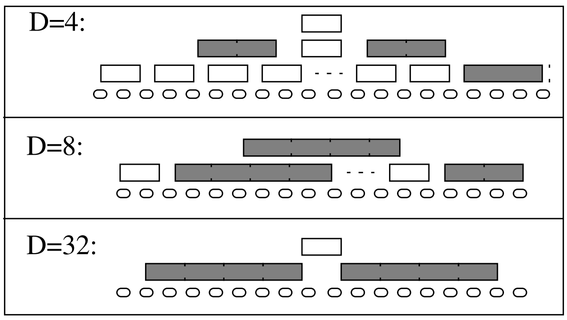


Figure 6: Various Shapes of the X-tree in different dimensions

There are two interesting special cases of the X-tree: (1) none of the directory nodes is a supernode and (2) the directory consists of only one large supernode (root). In the first case, the X-tree has a completely hierarchical organization of the directory and is therefore similar to an R-tree. This case may occur for low dimensional and non-overlapping data. In the second case, the directory of the X-tree is basically one root-supernode which contains the lowest directory level of the corresponding R-tree. The performance therefore corresponds to the performance of a linear directory scan. This case will only occur for high-dimensional or highly overlapping data where the directory would have to be completely searched anyway. The two cases also correspond to the two extremes for the height of the tree and the directory size. In case of a completely hierarchical organization, the height and size of the directory basically correspond to that of an R-tree. In the root-supernode case, the size of the directory linearly depends on the dimension

$$DirSize(D) = \frac{DatabaseSize}{BlockSize \cdot StorageUtil.} \cdot 2 \cdot BytesFloat \cdot D$$

For 1 GBytes of 16-dimensional data, a block size of 4 KBytes, a storage utilization of 66% for data nodes, and 4 bytes per float, the size of the directory is about 44 MBytes for the root-supernode in contrast to about 72 MBytes for the completely hierarchical directory.

3.2 Algorithms

The most important algorithm of the X-tree is the insertion algorithm. The insertion algorithm 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 (cf. Figure 7) 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 (cf. Figure 8) 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 [BKSS 90] 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 (cf. subsection 3.3). In subsection 3.3, we show that for point data there always exists an overlap-free split. 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

```
int X_DirectoryNode::insert(DataObject obj, X_Node **new_node)
{
    SET_OF_MBR *s1, *s2;
    X_Node *follow, *new_son;
    int return_value;

    follow = choose_subtree(obj);        // choose a son node to insert obj into
    return_value = follow->insert(obj, &new_son); // insert obj into subtree
    update_mbr(follow->calc_mbr());      // update MBR of old son node

    if (return_value == SPLIT){
        add_mbr(new_son->calc_mbr());   // insert mbr of new son node into current node

        if (num_of_mbrs() > CAPACITY){  // overflow occurs
            if (split(mbrs, s1, s2) == TRUE){
                // topological or overlap-minimal split was successfull
                set_mbrs(s1);
                *new_node = new X_DirectoryNode(s2);

                return SPLIT;
            }
            else // there is no good split
            {
                *new_node = new X_SuperNode();
                (*new_node)->set_mbrs(mbrs);

                return SUPERNODE;
            }
        }
    } else if (return_value == SUPERNODE){ // node 'follow' becomes a supernode
        remove_son(follow);
        insert_son(new_son);
    }

    return NO_SPLIT;
}
```

Figure 7: X-tree Insertion Algorithm for Directory Nodes

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. Since supernodes are created or extended in main memory, the local reorganization is only necessary when writing back the supernodes on secondary storage which does not occur frequently.

For point data, overlap in the X-tree directory may only occur if the overlap induced by the topological split is below a threshold overlap value (MAX_OVERLAP). In that case, the overlap-minimal split and the possible creation of a supernode do not make sense. The maximum overlap value which is acceptable is basically a system constant and depends on the page access time (T_{IO}), the time to transfer a block from disk into main memory (T_{Tr}), and the CPU time necessary to process a block (T_{CPU}). The maximum overlap value (*MaxO*[1]) may be determined approximately by the balance between reading a supernode of size 2*BlockSize and reading 2 blocks with a probability of MaxO and one block with a probability of (1-MaxO). This estimation is only correct for the most simple case of initially creating a supernode. It does not take the effect of further splits into account. Nevertheless, for practical purposes the following equation provides a good estimation:

$$MaxO \cdot 2 \cdot (T_{IO} + T_{Tr} + T_{CPU}) + (1 - MaxO) \cdot (T_{IO} + T_{Tr} + T_{CPU}) = T_{IO} + 2 \cdot (T_{Tr} + T_{CPU})$$
$$\Rightarrow \quad MaxO = \frac{T_{Tr} + T_{CPU}}{T_{IO} + T_{Tr} + T_{CPU}}$$

For realistic system values measured in our experiments (T_{IO} = 20 ms, T_{Tr} = 4 ms, T_{CPU} = 1 ms), the resulting *MaxO* value is 20%. Note that in the above formula, the fact that the probability of a node being in main memory is increasing due to the decreasing directory size in case of using the supernode has not yet been considered. The other constant of our algorithm (MIN_FANOUT) is the usual minimum fanout value of a node which is similar to the corresponding value used in other index structures. An appropriate value of MIN_FANOUT is between 35% and 45%.

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 under-

1. *MaxO* is the probability that we have to access both son nodes because of overlap during the search.

```
bool X_DirectoryNode::split(SET_OF_MBR *in, SET_OF_MBR *out1, SET_OF_MBR *out2)
{
   SET_OF_MBR t1, t2;
   MBR r1, r2;

   // first try topological split, resulting in two sets of MBRs t1 and t2
   topological_split(in, t1, t2);
   r1 = t1->calc_mbr(); r2 = t2->calc_mbr();

   // test for overlap
   if (overlap(r1, r2) > MAX_OVERLAP)
   {
      // topological split fails -> try overlap minimal split
      overlap_minimal_split(in, t1, t2);

      // test for unbalanced nodes
      if (t1->num_of_mbrs() < MIN_FANOUT || t2->num_of_mbrs() < MIN_FANOUT)
          // overlap-minimal split also fails (-> caller has to create supernode)
          return FALSE;
   }

   *out1 = t1; *out2 = t2;
   return TRUE;
}
```

Figure 8: X-tree Split Algorithm for Directory Nodes

flow 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, we reduce the size of the supernode by one block. The update operation can be seen as a combination of a delete and an insert operation and is therefore straightforward.

3.3 Determining the Overlap-Minimal Split

For determining an overlap-minimal split of a directory node, we have 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.

Definition 2 (Split)

The split of a node $S = \{mbr_1, \ldots, mbr_n\}$ into two subnodes $S_1 = \left\{mbr_{i_1}, \ldots, mbr_{i_{s_1}}\right\}$ and $S_2 = \left\{mbr_{i_1}, \ldots, mbr_{i_{s_2}}\right\}$ ($S_1 \neq \varnothing$ and $S_2 \neq \varnothing$) is defined as

$$Split(S) = \{(S_1, S_2) \mid S = S_1 \cup S_2 \ \wedge \ S_1 \cap S_2 = \varnothing \}.$$

The split is called

(1) overlap-minimal iff $\|MBR(S_1) \cap MBR(S_2)\|$ *is minimal*

(2) overlap-free iff $\|MBR(S_1) \cap MBR(S_2)\| = 0$

(3) balanced iff $-\varepsilon \leq |S_1| - |S_2| \leq \varepsilon$.

For obtaining a suitable directory structure, we are interested in overlap-minimal (overlap-free) splits which are balanced. For simplification, in the following we focus on overlap-free splits and assume to have high-dimensional uniformly distributed point data.[1] 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.

Lemma 1

For uniformly distributed point data, an overlap-free split is only possible iff there is a dimension according to which all MBRs in the node have been previously split. More formally,

$$\begin{aligned} &Split(S) \text{ is overlap-free } \Leftrightarrow \\ &\quad \exists\, d \in \{1, \ldots, D\}\ \forall mbr \in S: \\ &\qquad mbr \text{ has been split according to } d \end{aligned}$$

Proof *(by contradiction):*

" $\Rightarrow$ " : Assume that for all dimensions there is at least one MBR which has not been split in that dimension. This means for uniformly distributed data that the MBRs span the full range of values of the corresponding dimensions. Without loss of generality, we assume that the *mbr* which spans the full range of values of dimension *d* is assigned to S_1. As a consequence, $MBR(S_1)$ spans the full range for dimension *d*. Since the extension of $MBR(S_2)$ cannot be zero in dimension *d*, a split using dimension *d* as split axis cannot be overlap-free (i.e., $MBR(S_1) \cap MBR(S_2) \neq 0$). Since for all dimensions there is at least one MBR which has not been split in that dimension, we cannot find an overlap-free split.

" $\Leftarrow$ " : Assume that an overlap-free split of the node is not possible. This means that there is no dimension which can be used to partition the MBRs into two subsets S_1 and S_2. This however is in contradiction to the fact that there is a dimension *d* for which all MBRs have been split. For uniform-

1. According to our experiments, the results generalize to real data and even to spatial data (cf. section 4).

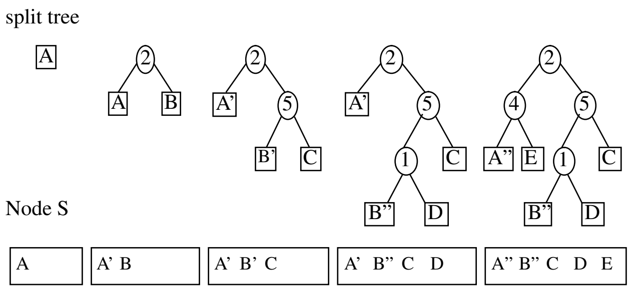


Figure 9: Example for the Split History

ly distributed point data, the split may be assumed to be in the middle of the range of dimension d and therefore, an overlap-free split is possible using dimension d.[1] ■

According to Lemma 1, for finding an overlap-free split we have to determine a dimension according to which all MBRs of S have been split previously. 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 9 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, we finally reach the situation depicted in the right most tree of Figure 9 where S is completely filled with the MBRs A", B", C, D and E.

According to Lemma 1, we 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, we only have to traverse the split tree from the root node to the leaf that corresponds to X. For example, MBR C has been split according to dimension 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 9 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.

Lemma 2 (Existence of an Overlap-free Split)
For point data, an overlap-free split always exists.

Proof *(using the split history):*

From the description of the split tree it is clear that all MBRs of a directory node S have one split dimension in common, namely the dimension used as split axis in the root node of the split tree. Let SD be this dimension. We are able to partition S such that all MBRs related to leaves in the left subtree of the root node are contained in S_1 and all other MBRs contained in S_2. Since any point belonging to S_1 has a lower value in dimension SD than all points belonging to S_2, the split is overlap-free[2]. ■

One may argue that there may exist more than one overlap-free split dimension which is part of the split history of all data pages. This is true in most cases for low dimensionality, but the probability that a second split dimension exists which is part of the split history of all MBRs is decreasing rapidly with increasing dimensionality (cf. Figure 10). If there is no dimension which is in the split history of all MBRs, the resulting overlap of the newly created directory entries is on the average about 50%. This can be explained as follows: Since at least one MBR has not been split in the split dimension d, one of the partitions (without loss of generality: S_1) spans the full range of values in that dimension. The

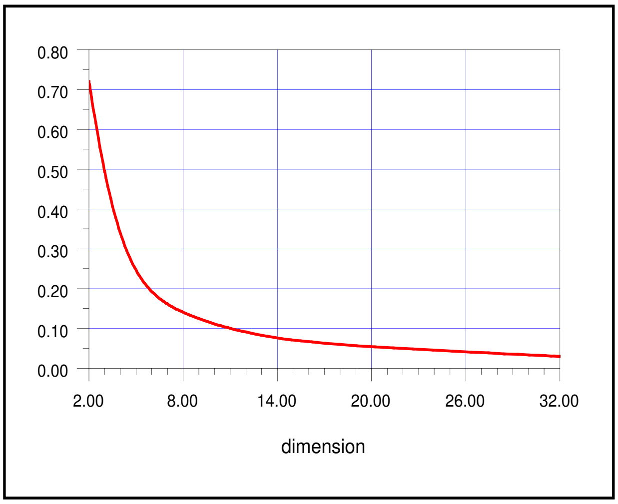
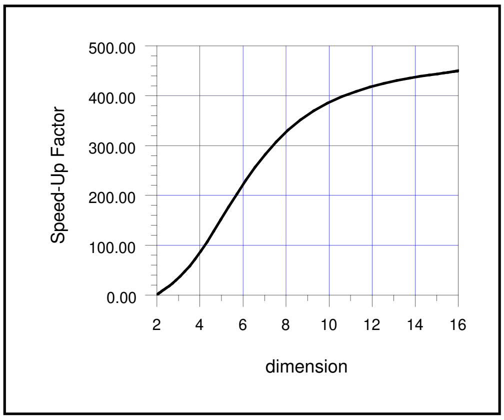
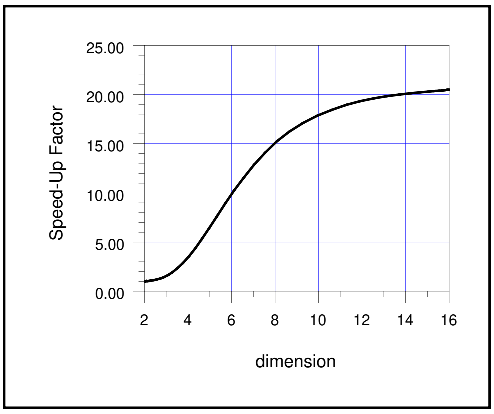


Figure 10: Probability of the Existence of a Second Overlap-free Split Dimension

1. If the splits have not been performed exactly in the middle of the data space, at least an overlap-minimal split is obtained.

2. Note that the resulting split is not necessarily balanced since sorted input data, for example, will result in an unbalanced split tree.





a. Point Query

b. 10 Nearest-Neighbor Query

Figure 11: Speed-Up of X-tree over R*-tree on Real Point Data (70 MBytes)

other partition S_2 spans at least half the range of values of the split dimension d. Since the MBRs are only partitioned with respect to dimension d, S_1 and S_2 span the full range of values of all other dimensions, resulting in a total overlap of about 50%.

The probability that a split algorithm which arbitrarily chooses the split axis coincidentally selects the right split axis for an overlap-free split is very low in high-dimensional space. As our analysis of the R*-tree shows, the behavior of the topological R*-tree split algorithm in high-dimensional space is similar to a random choice of the split axis since it optimizes different criteria. If the topological split fails, our split algorithm tries to perform an overlap-free split. This is done by determining the dimension for the overlap-free split as described above, determining the split value, and partitioning the MBRs with respect to the split value. If the resulting split is unbalanced, the insert algorithm of the X-tree initiates the creation/extension of a supernode (cf. subsection 3.2). Note that for the overlap-minimal split, information about the split history has to be stored in the directory nodes. The space needed for this purpose, however, is very small since the split history may be coded by a few bits.

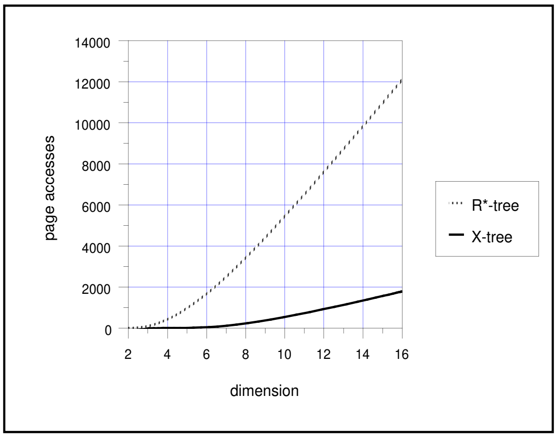
4. Performance Evaluation

To show the practical relevance of our method, we performed an extensive experimental evaluation of the X-tree and compared it to the TV-tree as well to as the R*-tree. All experimental results presented in this sections are computed on an HP735 workstation with 64 MBytes of main memory and several GBytes of secondary storage. All programs have been implemented in C++ as templates to support different types of data objects. The X-tree and R*-tree support different types of queries such as point queries and nearest neighbor queries; the implementation of the TV-tree[1] only supports point queries. We use the original implementation of the TV-tree by K. Lin, H. V. Jagadish, and C. Faloutsos [LJF 94].

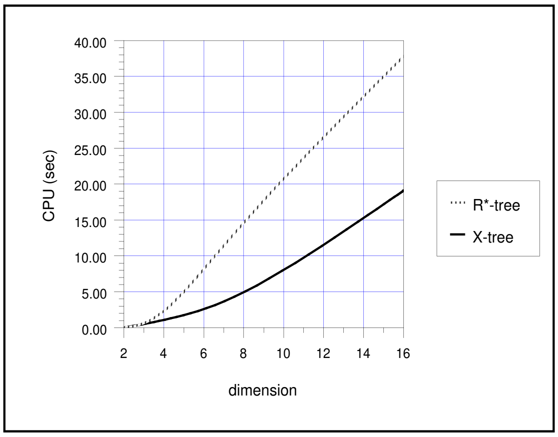
The test data used for the experiments are real point data consisting of Fourier points in high-dimensional space (D = 2, 4, 8, 16), spatial data (D = 2, 4, 8, 16) consisting of manifolds in high-dimensional space describing regions of real CAD-objects, and synthetic data consisting of uniformly distributed points in high-dimensional space (D = 2, 3, 4, 6, 8, 10, 12, 14, 16). The block size used for our experiments is 4 KByte, and all index structures were allowed to use the same amount of cache. For a realistic evaluation, we used very large amounts of data in our experiments. The total amount of disk space occupied by the created index structures of TV-trees, R*-trees, and X-trees is about 10 GByte and the CPU time for inserting the data adds up to about four weeks of CPU time. As one expects, the insertion times increase with increasing dimension. For all experiments, the insertion into the X-tree was much faster than the insertion into the TV-tree and the R*-tree (up to a factor of 10.45 faster than the R*-tree). The X-tree reached a rate of about 170 insertions per second for a 150 MBytes index containing 16-dimensional point data.

First, we evaluated the X-tree on synthetic databases with varying dimensionality. Using the same number of data items over the different dimensions implies that the size of the database is linearly increasing with the dimension. This however has an important drawback, namely that in low dimensions, we would obtain only very small databases, whereas in high dimensions the databases would become very large. It is more realistic to assume that the amount of data which is stored in the database is constant. This means, however, that the number of data items needs to be varied accordingly. For the experiment presented in Figure 13, we used 100 MByte databases containing uniformly distributed point data. The number of data items varied between 8.3 million for D=2 and 1.5 million for D=16. Figure 13, shows the speed-up of the search time for point queries of the X-tree

1. We use the original implementation of the TV-tree by K. Lin, H. V. Jagadish, and C. Faloutsos [LJF 94].



a. Page Accesses



b. CPU-Time

Figure 12: Number of Page Accesses versus CPU-Time on Real Point Data (70 MBytes)

over the R*-tree. As expected, the speed-up increases with growing dimension, reaching values of about 270 for D=16. For lower dimensions, the speed-up is still higher than one order of magnitude (e.g., for D=8 the speed-up is about 30). The high speed-up factors are caused by the fact that, due to the high overlap in high dimensions, the R*-tree needs to access most of the directory pages. The total query time turned out to be clearly dominated by the I/O-time, i.e. the number of page accesses (see also Figure 12).

Since one may argue that synthetic databases with uniformly distributed data are not realistic in high-dimensional space, we also used real data in our experiments. We had access to large Fourier databases of variable dimensionality containing about 70 Mbyte of Fourier data representing shapes of polygons. The results of our experiments (cf. Figure 11) show that the speed-up of the total search time for point queries is even higher (about 90 for D=4 and about

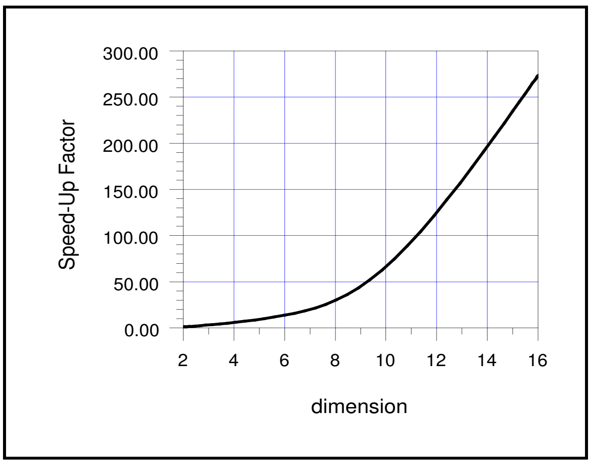


Figure 13: Speed-Up of X-tree over R*-tree on Point Queries (100 MBytes of Synthetic Point Data)

320 for D=8) than the speed-up of uniformly distributed data. This result was surprising but corresponds to the higher overlap of real data found in the overlap curves (cf. Figure 2). Additionally to point queries, in applications with high-dimensional data nearest neighbor queries are also important. We therefore also compared the performance of nearest neighbor queries searching for the 10 nearest neighbors. The nearest neighbor algorithm supported in the X-tree and R*-tree is the algorithm presented in [RKV 95]. The results of our comparisons show that the speed-up for nearest neighbor queries is still between about 10 for D=6 and about 20 for D=16. Since the nearest neighbor algorithm requires sorting the nodes according to the min-max distance, the CPU-time needed for nearest neighbor queries is much higher. In Figure 12, we therefore present the number of page accesses and the CPU-time of the X-tree and the R*-tree for nearest-neighbor queries. The figure shows that the X-tree provides a consistently better performance than the R*-tree. Note that, in counting page accesses, accesses to supernodes of size *s* are counted as *s* page accesses. In most practical cases, however, the supernodes will be cached due to the better main memory utilization of the X-tree. For practically relevant buffer sizes (1 MByte to 10 MBytes) there is no significant change of page accesses. For extreme buffer sizes of more than 10 MBytes or less than 1 MByte, the speed-up may decrease. The better CPU-times of the X-tree may be explained by the fact that due to the overlap the R*-tree has to search a large portion of the directory which in addition is larger than the X-tree directory.

Figure 14 shows the total search time of point queries depending on the size of the database (D=16). Note that in this figure we use a logarithmic scale of the y-axis, since otherwise the development of the times for the X-tree would not be visible (identical with the x-axis). Figure 14 shows that the search times of the X-tree are consistently about two orders of magnitude faster than those of the R*-tree

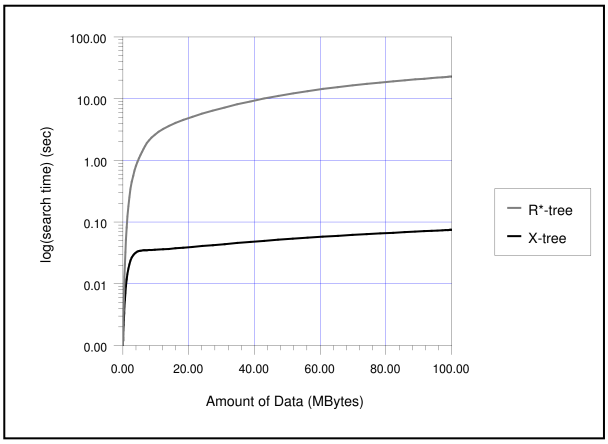


Figure 14: Total Search Time of Point Queries for Varying Database Size (Synthetic Point Data)

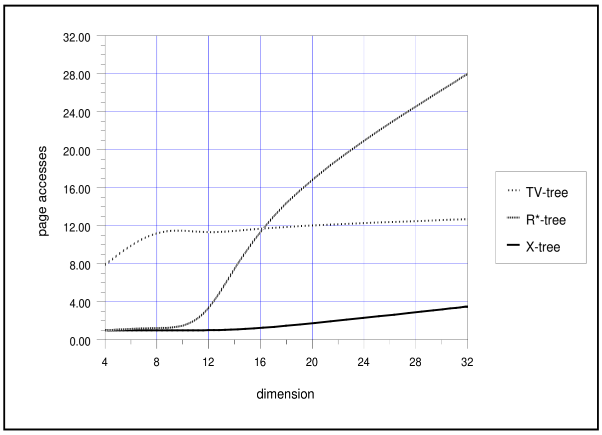


Figure 16: Comparison of X-tree, TV-tree, and R*-tree on Synthetic Data

(for D=16). The speed-up slightly increases with the database size from about 100 for 20 MBytes to about 270 for 100 MBytes. Also, as expected, the total search time of the X-tree grows logarithmically with the database size which means that the X-tree scales well to very large database sizes.

We also performed a comparison of the X-tree with the TV-tree and the R*-tree. With the implementation of the TV-tree made available to us by the authors of the TV-tree, we only managed to insert up to 25.000 data items which is slightly higher than the number of data items used in the original paper [LJF 94]. For the comparisons, we were therefore not able to use our large databases. The results of our comparisons are presented in Figure 16. The speed-up of the X-tree over the TV-tree ranges between 4 and 12, even for the rather small databases. It is interesting to note that the performance of the R*-tree is better than the performance of the TV-tree for D smaller than 16.

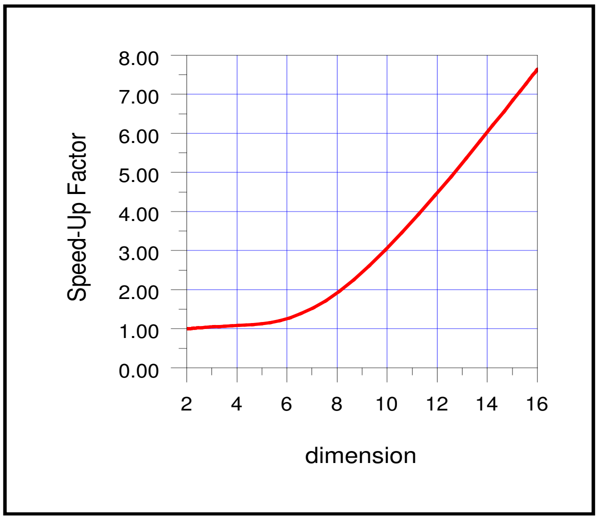


Figure 15: Speed-Up of X-tree over R*-tree on Real Extended Spatial Data

In addition to using point data, we also examined the performance of the X-tree for extended data objects in high-dimensional space. The results of our experiments are shown in Figure 15. Since the extended spatial data objects induce some overlap in the X-tree as well, the speed-up of the X-tree over the R*-tree is lower than for point data. Still, we achieve a speed-up factor of about 8 for D=16.

5. Conclusions

In this paper, we propose a new indexing method for high-dimensional data. We investigate the effects that occur in high dimensions and show that R-tree-based index structures do not behave well for indexing high-dimensional spaces. We introduce formal definitions of overlap and show the correlation between overlap in the directory and poor query performance. We then propose a new index structure, the X-tree, which uses - in addition to the concept of supernodes - a new split algorithm minimizing overlap. Supernodes are directory nodes which are extended over the usual block size in order to avoid a degeneration of the index. We carry out an extensive performance evaluation of the X-tree and compare the X-tree with the TV-tree and the R*-tree using up to 100 MBytes of point and spatial data. The experiments show that the X-tree outperforms the TV-tree and R*-tree up to orders of magnitude for point queries and nearest neighbor queries on both synthetic and real data.

Since for very high dimensionality the supernodes may become rather large, we currently work on a parallel version of the X-tree which is expected to provide a good performance even for larger data sets and the more time consuming nearest neighbor queries. We also develop a novel nearest neighbor algorithm for high-dimensional data which is adapted to the X-tree.

Acknowledgment

We are thankful to K. Lin, C. Faloutsos, and H. V. Jagadish for making the implementation of the TV-tree available to us.

References

[AFS 93] Agrawal R., Faloutsos C., Swami A.: *'Efficient Similarity Search in Sequence Databases',* Proc. 4th Int. Conf. on Foundations of Data Organization and Algorithms, Evanston, ILL, 1993, in: Lecture Notes in Computer Science, Vol. 730, Springer, 1993, pp. 69-84.

[AGMM 90] Altschul S. F., Gish W., Miller W., Myers E. W., Lipman D. J.: *'A Basic Local Alignment Search Tool',* Journal of Molecular Biology, Vol. 215, No. 3, 1990, pp. 403-410.

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[DE 82] Dunn G., Everitt B.: *'An Introduction to Mathematical Taxonomy',* Cambridge University Press, Cambridge, MA, 1982.

[Fal 94] Faloutsos C., Barber R., Flickner M., Hafner J., et al.: *'Efficient and Effective Querying by Image Content',* Journal of Intelligent Information Systems, 1994, Vol. 3, pp. 231-262.

[FL 95] Faloutsos C., Lin K.: *'Fastmap: A fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets',* Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA, 1995, pp. 163-174.

[Gut 84] Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984, pp. 47-57.

[GN 91] Günther O., Noltemeier H.: *'Spatial Database Indices For Large Extended Objects',* Proc. 7 th Int. Conf. on Data Engineering, 1991, pp. 520-527.

[Har 67] Harman H. H.: *'Modern Factor Analysis',* University of Chicago Press, 1967.

[Jag 91] Jagadish H. V.: *'A Retrieval Technique for Similar Shapes',* Proc. ACM SIGMOD Int. Conf. on Management of Data, 1991, pp. 208-217.

[Kuk 92] Kukich K.: *'Techniques for Automatically Correcting Words in Text',* ACM Computing Surveys, Vol. 24, No. 4, 1992, pp. 377-440.

[KW 78] Kruskal J. B., Wish M.: *'Multidimensional Scaling',* SAGE publications, Beverly Hills, 1978.

[LJF 94] Lin K., Jagadish H. V., Faloutsos C.: *'The TV-tree: An Index Structure for High-Dimensional Data',* VLDB Journal, Vol. 3, 1995, pp. 517-542.

[MG 93] Mehrotra R., Gary J. E.: *'Feature-Based Retrieval of Similar Shapes',* Proc. 9th Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 108-115.

[MG 95] Mehrotra R., Gary J. E.: *'Feature-Index-Based Similar Shape retrieval',* Proc. of the 3rd Working Conf. on Visual Database Systems, 1995, pp. 46-65.

[MN 95] Murase H., Nayar S. K: *'Three-Dimensional Object Recognition from Appearance-Parametric Eigenspace Method',* Systems and Computers in Japan, Vol. 26, No. 8, 1995, pp. 45-54.

[NHS 84] Nievergelt J., Hinterberger H., Sevcik K. C.: *'The Grid File: An Adaptable, Symmetric Multikey File Structure',* ACM Trans. on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.

[RKV 95] Roussopoulos N., Kelley S., Vincent F.: *'Nearest Neighbor Queries'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA, 1995, pp. 71-79.

[Rob 81] Robinson J. T.: *'The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes',* Proc. ACM SIGMOD Int. Conf. on Management of Data, 1981, pp. 10-18.

[SBK 92] Shoichet B. K., Bodian D. L., Kuntz I. D.: *'Molecular Docking Using Shape Descriptors',* Journal of Computational Chemistry, Vol. 13, No. 3, 1992, pp. 380-397.

[SH 94] Shawney H., Hafner J.: *'Efficient Color Histogram Indexing',* Proc. Int. Conf. on Image Processing, 1994, pp. 66-70.

[SK 90] Seeger B., Kriegel H.-P.: *'The Buddy Tree: An Efficient and Robust Access Method for Spatial Data Base Systems',* Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia, 1990, pp. 590-601.

[SRF 87] Sellis T., Roussopoulos N., Faloutsos C.: *'The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects',* Proc. 13th Int. Conf. on Very Large Databases, Brighton, England, 1987, pp 507-518.

[WJ 96] White, D., Jain R.: *'Similarity Indexing with the SS-tree',* Proc. 12th Int. Conf. on Data Engineering, New Orleans, LA, 1996.

[WW 80] Wallace T., Wintz P.: *'An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors',* Computer Graphics and Image Processing, Vol. 13, 1980, pp. 99-126.