# High-dimensional indexing for multimedia features

Ira Assent* Stephan Günnemann+ Hardy Kremer+ Thomas Seidl+
*Department of Computer Science
Aalborg University, Denmark
ira@cs.aau.dk
+Data Mining and Data Exploration Group
RWTH Aachen University, Germany
{guennemann,kremer,seidl}@cs.rwth-aachen.de

**Abstract:** Efficient content-based similarity search in large multimedia databases requires efficient query processing algorithms for many practical applications. Especially in high-dimensional spaces, the huge number of features is a challenge to existing indexing structures. Due to increasing overlap with growing dimensionality, they eventually fail to deliver runtime improvements.

In this work, we propose an overlap-free approach to indexing to overcome these problems and allow efficient query processing even on high-dimensional feature vectors. Our method is inspired by separator splits e.g. in B-trees for one-dimensional data or for sequence data. We transform feature vectors such that overlap-free splits are ensured. Our algorithm then queries the database with substantially reduced number of disk accesses, while ensuring correctness and completeness of the result.

Our experiments on several real world multimedia databases demonstrate that we build compact and overlap-free directory information in our index that avoids large percentages of disk accesses, thus outperforming existing multidimensional indexing structures.

## 1 Introduction

There is tremendous growth in multimedia databases in application domains such as medicine, engineering, biology and entertainment. New technologies such as computer tomography or digital photography produce increasingly large volumes of data. Typical access to multimedia data is required in form of content-based similarity search, e.g. to support medical diagnosis by automatically retrieving similar magnetic resonance images. Similarity models are usually defined via distance functions evaluated with respect to content related features like color distribution or shape histograms.

Due to the massive amount of multimedia data, efficient algorithms for content-based similarity search are of foremost importance. Efficient retrieval is especially challenging for high resolution of features, i.e. high-dimensional feature vectors. In high-dimensional feature spaces, not only the one-to-one comparison of multimedia features is more costly. Most importantly yet, multidimensional indexing structures that are deployed to improve runtimes, degenerate. Degeneration is due to the fact that with increasing feature dimensionality, descriptor regions (e.g. minimum bounding rectangles in R-trees) overlap to a

growing extent. Consequently, most queries will require access to a large part of the directory and random access at the leaf level. Thus, eventually index-based search algorithms may perform even worse than a sequential scan of the entire database, meaning that they fail at their very goal of efficiency improvement.

In this work, we propose an indexing approach that avoids overlap even for high-dimensional feature histograms, thus allowing efficient runtimes for content-based similarity search. Our idea is inspired by the observation that for one-dimensional or sequential data types, overlap-free indexing approaches exist. Using the inherent order of dimensions, separators are directory entries that split the data into smaller and larger values without any overlap. We propose to derive a meaningful ordering of dimensions and discretization of values to generate an overlap-free separator split, while maintaining minimum bounding rectangles to enhance the pruning power.

In our method, query processing is efficiently possibly, as the number of paths that have to be followed for any given query is greatly decreased, and consequently, the number of disk accesses is substantially reduced. We provide a multistep filter-and-refine algorithm to ensure that discretization neither produces false dismissals nor introduces false hits.

We demonstrate in thorough experiments on several real world databases that our separator-based approach builds very compact index directories, and that the number of disk accesses is substantially reduced in comparison with competing approaches.

Summarizing, advantages of our method include:

- Compact and overlap-free indexing for multimedia databases
- Scalability to high-dimensional and massive data
- Efficient, complete, and correct query processing algorithm
- Substantially reduced number of disk accesses

This paper is structured as follows: we review related work on indexing of multimedia databases in Section 2 and study the challenges involved in overlap-free indexing for content-based similarity search in Section 3. Section 4 details our method, with Section 4.1 describing the indexing concept and its properties, whereas Section 4.2 specifies the index structure followed by the query processing algorithm in Section 4.3. Our experiments in Section 5 demonstrate great reduction in disk accesses for query processing, before we conclude our work in Section 6.

## 2 Related work

In the literature, different indexing structures for efficient query processing have been proposed [Sam06]. The general idea is to maintain compact directory entries that hierarchically narrow down the search to relevant parts of the database, thus decreasing the number of disk accesses and consequently the overall search time as well.

The original B-tree was developed for indexing of one-dimensional data or keys [BM70, BM72, BU77]. Directory entries are separators between smaller and larger values with respect to the order in this dimension. Moreover, as nodes reflect disk page size, short separators usually provide compact directory entries with large fanout, and thus small trees. During search, point queries will only require access to one path along the tree, and typically range queries read only small parts of the database as well. In String B-trees or the TS-tree, this idea was also used for sequential data where separators between sequences can be computed as well [FG99, AKAS08].

In multidimensional or spatial data, the R-tree family extends the idea of page-based balanced trees [Gut84, BKSS90, MNPT06]. Directory entries are minimum bounding rectangles of the respective subtrees. For low to medium dimensional data, these indexing structures provide substantial efficiency gains. With increasing dimensionality, however, overlap of minimum bounding rectangles eventually leads to degeneration. Typical queries overlap with larger parts of the directory and hardly any data can be pruned from the search. Consequently, indexing fails to deliver efficiency gains in high dimensional data, even compared with brute force sequential scan of the entire database.

This observation has led to index structures that work towards higher dimensionality. The X-tree extends the R-tree by introducing supernodes if a reasonable minimum bounding rectangle cannot be derived, and maintains a split dimension history to optimize the split choice in R*-trees [BKK96]. This approach alleviates problems with scalability, but cannot solve the problem entirely, as in high dimensional data overlap dominates the index eventually.

Other approaches have focused on improving the performance of the sequential scan directly. In the VA-file index, quantized representatives of the database are searched sequentially, and only potentially relevant representatives are refined by direct access to the respective multimedia objects [WSB98]. The IQ-tree and A-tree combine the idea of quantization with that of multidimensional index structures [BBJ+00, SYUK00]. The IQ-tree approach uses different quantization at different levels of the tree and requires computation of the trade-off between finer resolution in quantization or splitting of nodes according to a cost model. The A-tree uses relative quantization with respect to the parent node and thus does not require costly recomputation. We compare our method against the A-tree in the experiments.

Another family of index structures is based on vantage points or other directory information based on relative distances to other objects in the database [Zez06]. Examples include MVP-tree, M-tree and the Slim-tree [BO97, CPZ97, TTSF99]. They are especially useful for metric databases where dimensional information is not necessarily available. In these cases, only other objects or their mutual distances can be used to create directories. They also suffer from poor performance when overlap is large [TTSF99].

In this work, we index multimedia data as in the TS-tree for sequential data. Our goal is to compute separators between nodes to guarantee overlap-freeness, yet maintain minimum bounding rectangles as in R*-trees.

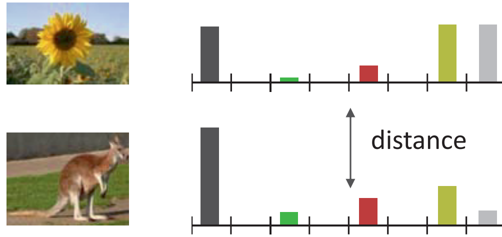



Figure 1: Feature vector based similarity

## 3 Multimedia indexing

Similarity models define the relevant characteristics of multimedia objects, for example color distribution in images, or pixel frequencies in shapes. Such features can be compared via distance functions which assign a degree of dissimilarity to any pair of histograms. An example is given in Figure 1. Two images and their respective color histograms, i.e. the relative frequencies of color pixels in each image, are depicted. To compare the images, distance functions compute a dissimilarity value on their histogram features.

We formally introduce the concept of feature vectors that can represent a variety of multimedia characteristics.

**Definition 1** ***Feature vector.***

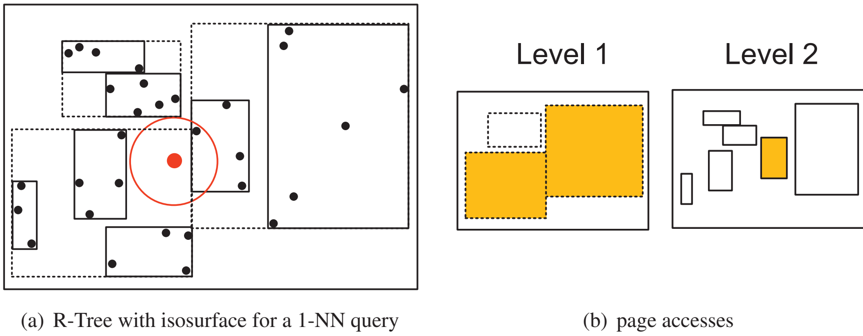
*For any multimedia object $o$ in database $DB$, a $d$-dimensional feature extraction $E$ is a mapping $E : DB \rightarrow R^d$ to a $d$-dimensional real valued feature vector $f = (f_1, \dots, f_d)$ with bin entries corresponding to each of the dimensions $i \in \{1, \dots, d\}$.*

An example that we will use throughout this work without loss of generality is (normalized) image color histograms. Then mapping $E$ is instantiated as the relative number of pixels corresponding to a certain color range in feature space:

$$f_i = \frac{|\{p_{xy} | p_{xy} \in b_i\}|}{|\{p_{xy}\}|}$$

where $p_{xy}$ denotes a pixel in position $x, y$, and $b_i$ denotes the corresponding bin per dimension $i$.

In multidimensional index structures, the main idea is to use the information contained in the individual dimensions. Hierarchical index structures create a small-size directory of the entire database based on those dimensions. This directory is used to direct the similarity search process to determine the result set. To do so, multidimensional features are grouped together according to their respective values. They are summarized by bounding geometries kept at the next higher levels in the directory. During retrieval, the query is compared to these bounding geometries to direct the search towards the result. In Figure 2(a), two-dimensional points are grouped in a hierarchy of minimum bounding rectangles. These rectangles are stored in a tree (Figure 3). Starting at the root, the query is




(a) R-Tree with isosurface for a 1-NN query (b) page accesses

Figure 2: Minimum bounding rectangles geometrically

compared to the rectangles, thus choosing only paths down to the data that are relevant for the query. Comparison to rectangles using the Euclidean distance is straightforward. If the query $q = (q_1, \ldots, q_n)$ is smaller than the rectangle $\mathbf{MBR} = ((l_1, u_1), \ldots, (l_n, u_n))$ in that dimension, compute the distance to the lower boundary of the rectangle, if it is larger to the upper. Otherwise, the distance is zero:

$$dist_{MBR}(q, \mathbf{MBR}) = \sum_{i=1}^{n} \begin{cases} (q_i - l_i)^2 & q_i < l_i \\ (q_i - u_i)^2 & u_i < q_i \\ 0 & else \end{cases}$$

Figure 2(b) shows the page accesses for a Nearest Neighbor query for the query point in (a). Clearly, only a subset of the pages has to be accessed, resulting in a significant speed up.

As R*-trees grow bottom-up by splitting of minimum bounding rectangles if the corresponding node overflows, in high-dimensional spaces, many dimensions are not split at all or only once. This means that many of the MBRs extend across the entire unsplit dimension. Consequently, overlap is huge, and queries intersect with most of the minimum bounding rectangles. Figure 4 shows this problem in a low dimensional setting for illustrative purposes: if many MBRs overlap, the query $q$ at the center of the circular query range has to access most nodes in the tree, requiring access to the entire directory plus random read of the database. This worst case scenario is not typical for low dimensional spaces, but very common in high dimensional spaces.

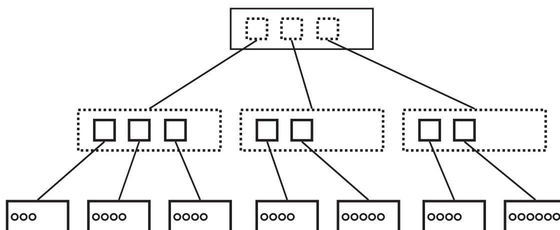
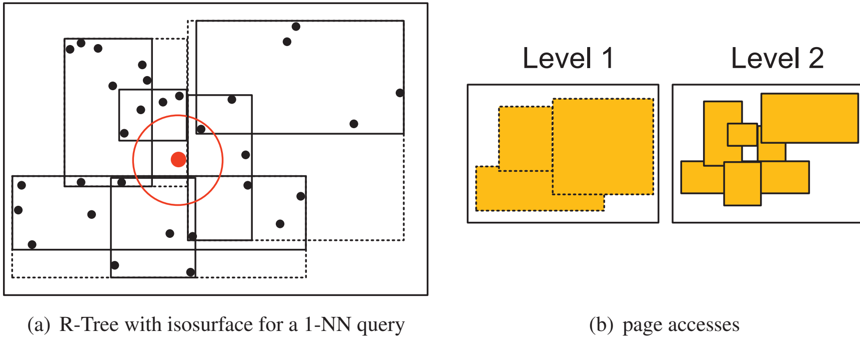


Figure 3: Hierarchy of minimum bounding rectangles




(a) R-Tree with isosurface for a 1-NN query

(b) page accesses

Figure 4: Overlap leads to index degeneration

To improve efficiency of multidimensional indexing, avoiding overlap is thus crucial.

# 4 Overlap-free indexing

We avoid overlap using a method successfully used in B-trees and TS-trees for one-dimensional and sequential data. The basic observation is that separator split leads to overlap-free descriptors. Consequently, the indexing concept relies on transformation and ordering of the feature vectors for this purpose. We show how this concept is used algorithmically in a lossless multistep filter-and-refine approach.

## 4.1 Indexing concept

Naively using a B-tree to index multimedia feature vectors would mean that only the first few dimensions are used for indexing [BM70, BM72, BU77]. Consequently, the first dimensions would dominate the query processing, and, even worse, only information on those first dimensions could be used before reaching leaf level. Thus, if there is no descriptor information on the remaining dimensions, we would see far too many disk accesses during query processing.

We therefore propose to proceed as in TS-trees for time series and include meta data on all dimensions, as well as principal components analysis to ensure that the first dimensions are the most relevant for indexing [AKAS08, Jol86]. For multimedia feature vectors, the additional meta data consists of minimum bounding rectangles, just as in R-trees or R*-trees.

**Definition 2** ***Separator.***

*A separator $S$ is the shortest string that is lexicographically larger than the left subtree $t_l$ and lexicographically smaller (or equal) than the right subtree $t_r$ : $t_l < S \leq t_r$.*

**Example. Separators according to default order.**

Assume a small database of only four feature vectors $f_1 = (0.1, 0.1, 0.1, 0.7)$, $f_2 = (0.1, 0.1, 0.2, 0.6)$, $f_3 = (0.1, 0.1, 0.3, 0.5), f_4 = (0.1, 0.1, 0.4, 0.4)$. Using the order of dimensions as given in this example, a suitable separator to distinguish between the two smaller objects and the two larger objects would be $s = (0.1, 0.1, 0.3)$ with $s > f_1, f_2$ and $s \leq f_3, f_4$. A different method, e.g. based on the variances of the dimensions, could identify dimensions three and four as those with the most information, and would thus re-order the dimensions accordingly: $f'_1 = (0.1, 0.7, 0.1, 0.1), f'_2 = (0.2, 0.6, 0.1, 0.1), f'_3 = (0.3, 0.5, 0.1, 0.1), f'_4 = (0.4, 0.4, 0.1, 0.1)$. Then, a separator to distinguish between the transformed feature vectors in the new space would be $s' = (0.3)$. This leads to separators that provide directory information on the most relevant dimensions in the database, thus providing good pruning power. Assuming that most queries follow the distribution in the database, queries might show values of $q = (0.1, 0.1, *, *)$, i.e. have a distance of zero to $s$ in the first two dimensions, leading to no pruning possibilities. Separators in the transformed space are moreover much shorter, and thus require less storage space which in turn leads to greater fanout.

Alternatively, the variance based reordering can be replaced by PCA (principal component analysis [Jol86]). Thus the statistical covariance in the dimensions is also considered by the transformation. In our algorithm and in the experimental section we use the PCA approach, while for illustrative purposes the variance based method is used.

Building separators on suitably ordered feature vectors still leaves one problem: on continuous valued vectors, separators would typically only be of length one, that is, be built just for the very first dimension. This is due to the fact that for real numbers, it is not very common to see the exact same values in many different objects. Consequently, splitting between different values does not require incorporating any further dimension and hence the objects are grouped only by the similarity in the first dimension. Therefore, discretization is necessary to ensure that separators include more dimensions.

**Example. Separators on continuous values.**

Given a database of five feature vectors $f_1 = (0.1, 0.1, 0.1, 0.7)$, $f_2 = (0.11, 0.29, 0.4, 0.2)$, $f_3 = (0.12, 0.38, 0.3, 0.3)$, $f_4 = (0.13, 0.07, 0.7, 0.1)$, $f_5 = (0.14, 0.1, 0.1, 0.7)$. One can see, that the two features $f_1$ and $f_5$ are very similar. But using the order of dimensions as given in this example, a suitable separator to distinguish between the two smaller objects and the three larger objects would be $s = (0.12)$ with $s > f_1, f_2$ and $s \leq f_3, f_4, f_5$. Thus, even though the objects $f_1$ and $f_5$ are very similar they are not grouped together, because they differ marginally in the first dimension. At the same time the separator length is only one, even though the first dimension provides low information. This fact cannot be uncovered by the separators, as real numbers that clearly distinguish between the values in the first dimension exist. By discretization of the objects, this effect is avoided. If only a certain number of ranges are used, separators will grow as more objects with the same discretized representation are indexed. For example, assuming a discretization with respect to the ranges $a = [0, 0.25)$, $b = [0.25, 0.5)$, $c = [0.5, 0.75)$, and $d = [0.75, 1]$, we get $\hat{f}_1 = (a, a, a, c), \hat{f}_2 = (a, b, b, a), \hat{f}_3 = (a, b, b, b), \hat{f}_4 = (a, a, c, a), \hat{f}_5 = (a, a, a, c)$. A

discretized separator would thus be $\hat{s} = (a, a, c)$ with $\hat{s} > \hat{f}_1, \hat{f}_5$ and $\hat{s} \leq \hat{f}_2, \hat{f}_3, \hat{f}_4$.

As we can see, discretization thus leads to automatic growth of separators, along with a mapping of similar feature vectors to the same discretized representation. Thus, discretization leads to a loss of information in feature vectors. To ensure correctness of the result, discretized representatives are used only in the index directory, whereas leafs store pointers to the original data. Upon reaching the leaf level, any potentially relevant feature vector in the original continuous-valued representation is used for comparison. Thus, during query processing, the query is compared against discretized ranges. Using a conservative distance computation, the actual distance is potentially underestimated. Refinement with the original features ensures that any false alarms are cleared out.

**Definition 3** ***Discretization.***

*Discretization of a feature vector $f$ to a partition $P = p_1, \ldots, p_m$ with $p_i = [l_{p_i}, u_{p_i})$ of adjacent ranges with $u_{p_{i-1}} = l_{p_i}$ maps each feature value $f_i$ to $p_j$ with $l_{p_j} \leq f_i < u_{p_j}$.*

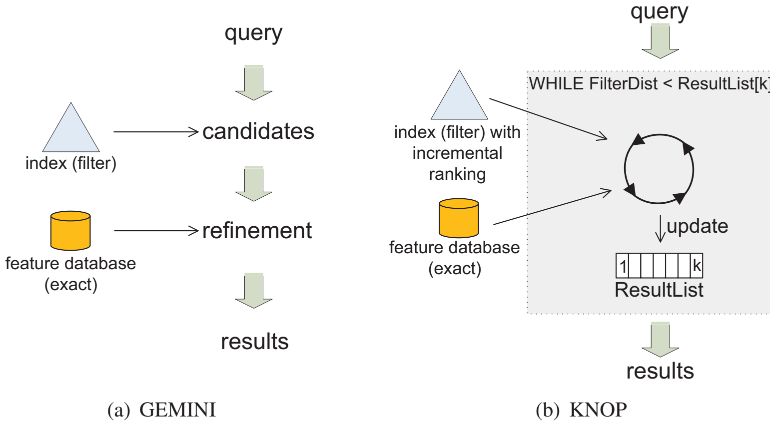
**Example. Query processing with discretized separators.**

In a database of five feature vectors $f_1 = (0.1, 0.1, 0.1, 0.7), f_2 = (0.11, 0.29, 0.4, 0.2)$, $f_3 = (0.12, 0.38, 0.3, 0.3), f_4 = (0.13, 0.07, 0.7, 0.1), f_5 = (0.14, 0.1, 0.1, 0.7)$. After variance based reordering and discretization, we get: $\hat{f}_1 = (c, a, a, a), \hat{f}_2 = (a, b, b, a)$, $\hat{f}_3 = (b, b, b, a), \hat{f}_4 = (a, c, a, a), \hat{f}_5 = (c, a, a, a)$. A suitable separator would be $s = (b)$ with $s > \hat{f}_2, \hat{f}_4$ and $s \leq \hat{f}_1, \hat{f}_3, \hat{f}_5$. Assuming a query $q = (0.1, 0.1, 0.1, 0.8)$ we want to find all features $f_i$ with a maximal (euclidean) distance of $\varepsilon = 0.1$ to $q$. Obviously the correct result set is $\{f_1\}$.
For query processing, $q$ is transformed to $\hat{q} = (d, a, a, a)$. Now we can calculate the distance between $\hat{q}$ and $s$, which results in a minimal value of $(d, a, a, a) - (b, *, *, *) = d - b = 0.75 - 0.5 = 0.25 > \varepsilon$. Therefore both features $\hat{f}_2, \hat{f}_4 < s$ can be pruned, because their distance could only increase. If we proceed with the distance computation to the remaining discretized features $\hat{f}_1, \hat{f}_3, \hat{f}_5$ the result set would be $\{\hat{f}_1, \hat{f}_5\}$.

Hence query processing on the discretized representation in the index structure alone would obviously result in incorrect results, as the exact value of the original feature vector is ignored. We therefore proceed in a multistep filter-and-refine fashion. Multistep algorithms, as e.g. in the GEMINI or KNOP frameworks speed up query processing by efficiently reducing the database to a small set of candidates via filtering [Fal96, SK98]. Only for those candidates the exact distance is used for refinement. The general idea is illustrated in Figure 5(a): the query is first evaluated with respect to the indexed representatives to obtain the set of candidates. To refine them, the full representation from the database is used.

To ensure that query processing via multistep filter-and-refine algorithms yields the desired efficiency gains, a number of quality criteria should be fulfilled. Summarized as *ICES* [AWS06], they refer to

- Index: to tap the full potential of database techniques for low runtimes, usage of indexing structures should be supported by the algorithm.




(a) GEMINI (b) KNOP

Figure 5: Multistep filter-and-refine frameworks

- Completeness: to ensure that efficiency is not at the cost of correctness of the result, the filter should not drop any true result.
- Efficiency: the filter step should be substantially more efficient than the refinement step, such that the overall algorithm is more efficient as well.
- Selectivity: the filter step should reduce the entire database to a small fraction of potential candidate objects, such that the costly refinement step has to be executed as little as possible.

Obviously, our approach offers index support. It is also complete, as we will prove below. Efficiency is guaranteed as computing the distance to the boundaries of the descriptors is efficiently possible. And finally, selectivity will be demonstrated in our experiments.

We deploy the KNOP multistep framework, as it minimizes the number of refinements in $kNN$ queries by keeping a $dist_{max}$ value of the current $k$th best nearest neighbor distance by immediate refinement of each potential candidate as it is demonstrated in Figure 5(b).

Formally, we ensure completeness of query processing via proof of the lower bounding property. Lower bounding property means that the distance computed between query and the index descriptors is never greater than the actual distance. Computing the distance between query and discretized features as the minimum of any feature vector within this range, fulfills the lower bounding property. As shown in [Fal96, SK98], lower bounding distance functions guarantee no false dismissals during multistep filter-and-refine processing according to the GEMINI or KNOP framework, respectively.

**Theorem 1** ***Lower bounding property of discretized feature vectors.***

*For any query feature vector $q$ and any feature vector $f$, the distance between $q$ and $f$ is lower bounded by the distance between $q$ and discretized feature vector $\hat{f}$:*

$$dist(q, \hat{f}) \leq dist(q, f)$$

**Proof 1**

*From the definition of the distance to upper and lower bounds, we immediately have*

$$dist(q,\hat{f}) = \sum_{i=1}^{n} \begin{cases} (q_i - l_i)^2 & q_i < l_i \\ (q_i - u_i)^2 & u_i < q_i \\ 0 & else \end{cases} \leq \sum_{i=1}^{n} \begin{cases} (q_i - f_i)^2 & q_i < l_i \\ (q_i - f_i)^2 & u_i < q_i \\ (q_i - f_i)^2 & else \end{cases} = dist(q,f)$$

□

### 4.2 The overlap-free tree

In this section we formalize the concepts presented in the previous section by introducing the overlap-free tree (OFT). With the discretized separator information at hand we can construct an overlap free index by adapting a B-Tree construction algorithm. The query processing for such an index was already shown in a previous example, where the separator information was used for pruning some feature vectors. To enhance the pruning power our index additionally stores MBRs in the index nodes. Thus pruning of subtrees is accessorily possible similar to the pruning methods in R-trees.

Formally, nodes in the index contain discretized representatives. As in TS-trees, we use finer partitioning to a degree $r$ of the leaf entries and the MBRs, and a rougher partitioning of the separators to ensure that more dimensions are used. Separators are simply added to the MBR information in R-trees or R*-trees, to rule out overlap during split, and identify relevant parts of the tree during query processing:

**Definition 4** ***Index node.***

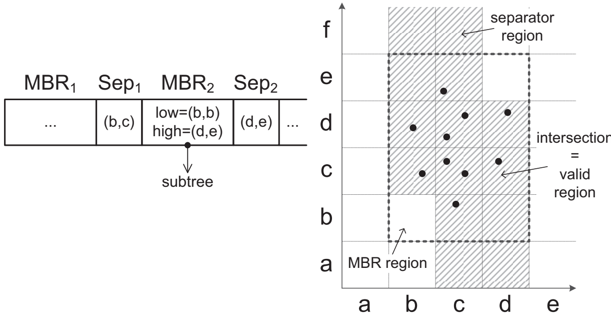
*For branching factor $m$, rough discretization parameter $r$ and fine discretization parameter $f$ a node stores:*

- *$k$ entries ($m \leq k \leq 2m$) with MBRs and pointers to subtrees and additionally $k-1$ separators*
- *separators are discretized to the $r$ partitions and prefix compressed*
- *MBRs and leaf entries are discretized to the $f$ partitions*

The root node, of course, is allowed to contain fewer entries, i.e. between $2 \leq k \leq 2m$ if it is an inner node.

An example inner node is presented in Figure 6 (left). Each feature vector $\hat{f}$ covered by an entry/subtree, is located between the two neighboring separators (e.g. $\mathbf{Sep}_1 \leq \hat{f} < \mathbf{Sep}_2$) as well as within the MBR (e.g. $\mathbf{MBR}_2$). Figure 6 (right) illustrates this valid region for $\mathbf{MBR}_2$ and the two separators. In general we use a descriptor $\mathbf{D} = (\mathbf{MBR}, \mathbf{S_l}, \mathbf{S_r})$ to identify such regions.

During query processing, we could simply compute the distance only to the MBRs as in R-trees. However, as in B-trees or TS-trees, we may use the information given by



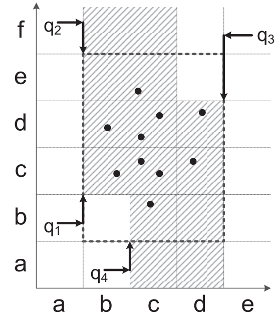




Figure 6: Example of an inner node (left) and the resulting regions in a 2d space (right)

Figure 7: mindist to valid region for different query points

the separators not only for the split, but for query processing as well. Basically, we are interested in not only computing the minimum distance to the MBR or the separator, but instead to their intersection [AKAS08].

Figure 7 shows four different query points and their minimal distances to the valid region. As one can see, there are three different cases how the distances are determined. These cases mainly differ in the assignment of the query points to the three shaded "columns" (i.e. partitions in the first dimension) of the diagram.

- case A: If a query point is assigned to the first column, one has to ensure that the (virtual) point where the minimal distance could be realized, is lexicographically larger than the left separator (see $q_1$). Additionally we can add the distance to the MBR in the remaining dimensions, if the virtual point is still outside the MBR (see $q_2$).
- case B: The same argument holds if a query point is assigned to the last column (see $q_3$). First the lexicographical-smaller-relation to the right separator must be obtained and second the distance to the MBR (for the remaining dimensions) can be added.
- case C: In the last case a query point is assigned to a column in between (see $q_4$). The lexicographical relation to both separators is directly fulfilled, and one can add the distance to the MBR.

For a query $q$ we have to minimize over all three cases to get the minimal distance to the valid region. A running example with a (discretized) query $q = (a, a)$ and the descriptor from Figure 6 is illustrated in Figure 8.

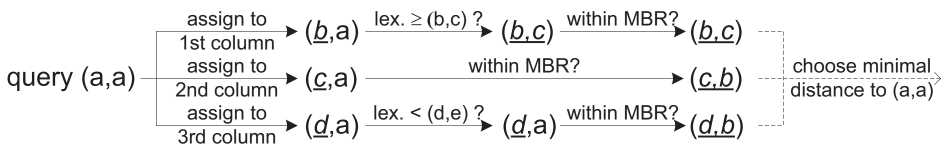



Figure 8: Schematic representation of the mindist computation

The rare situation (case D), where both separators starts with the same symbol ($\mathbf{S_l}_1 = \mathbf{S_r}_1$), i.e. only one "column" exists, can be considered by calculating the minimal distance in this first dimension and go ahead in a recursive fashion. The complete minimal distance is formalized by:

**Definition 5** ***Minimum distance to MBRs and separators.***
*The* $mindist$ *between query* $q = (q_1, \ldots, q_n)$ *and a descriptor* $\mathbf{D} = (\mathbf{MBR}, \mathbf{S_l}, \mathbf{S_r})$ *of a subtree is defined as* $mindist(q, \mathbf{D}) := mindist(q, \mathbf{D}, 1)$*:*

$$mindist(q, \mathbf{D}, i) = \begin{cases} |q_i - \mathbf{S_l}_i| + mindist(q, \mathbf{D}, i+1) & \mathbf{S_l}_i = \mathbf{S_r}_i \quad (\textit{case D}) \\ mindist'(q, \mathbf{D}, i) & else \end{cases}$$

$$mindist'(q, \mathbf{D}, i) = \min \begin{cases} |q_i - \mathbf{S_l}_i| + mindist_{\geq}(i+1) & (\textit{case A}) \\ |q_i - \mathbf{S_r}_i| + mindist_{<}(i+1) & (\textit{case B}) \\ |q_i - \mathbf{S}| + dist_{MBR}(i+1), \forall \mathbf{S} : \mathbf{S_l}_i < \mathbf{S} < \mathbf{S_r}_i & (\textit{case C}) \end{cases}$$

*where:*

$$mindist_{\geq}(i) = \begin{cases} 0 & i > n \\ dist_{MBR}(i) & q_i > \mathbf{S_l}_i \wedge i \leq n \\ \min \begin{cases} |q_i - \mathbf{S_l}_i| + mindist_{\geq}(i+1) \\ |q_i - (\mathbf{S_l}_i + 1)| + dist_{MBR}(i+1) \end{cases} & else \end{cases}$$

$$mindist_{<}(i) = \begin{cases} \infty & i > n \\ dist_{MBR}(i) & q_i < \mathbf{S_r}_i \wedge i \leq n \\ \min \begin{cases} |q_i - \mathbf{S_r}_i| + mindist_{<}(i+1) \\ |q_i - (\mathbf{S_r}_i - 1)| + dist_{MBR}(i+1) \end{cases} & else \end{cases}$$

The functions $mindist_{\geq}(i)$ and $mindist_{<}(i)$ ensure in a recursive fashion, that the lexicographical order and the containment in the MBR is maintained, which is used by the two cases (A) and (B). In addition the expression $dist_{MBR}(i)$ is an abbreviation for the minimal distance from a point $q$ to a MBR, whereas both objects are restricted to the remaining dimensions $i \ldots n$, i.e.

$$dist_{MBR}(i) = dist_{MBR}(q_{|i \ldots n}, \mathbf{MBR}_{|i \ldots n}) = \sum_{j=i}^{n} \begin{cases} (q_j - l_j)^2 & q_j < l_j \\ (q_j - u_j)^2 & u_j < q_j \\ 0 & else \end{cases}$$

### 4.3 Algorithm

In this section, we give a general overview over our algorithm and discuss its inner workings. Algorithm 1 gives a pseudo-code description.

The algorithm starts by comparing the query against the information stored in the root node of the tree, i.e. the minimum bounding rectangles as well as the separators. Based

on the distance values computed, it decends the tree along the most promising path in terms of minimum distance until the first leaf node is refined. At this point, the actual data page containing the original feature vector representations is accessed. For all of these original feature vectors stored on this particular page, the algorithm sequentially compares the distance to the query. If their distances are better than the current set of $k$ nearest neighbors (which is obviously the case initially), they replace candidates in the result set that have a larger distance. $dist_{max}$, the variable that stores the distance of the $k$th nearest neighbor in the set, is updated accordingly. Query processing continues with the next most promising index or data page until the best filter distance is no longer smaller than the $k$th nearest neighbor. From the proof of the lower bounding property, we are now sure that the candidate set is exactly the true result set and return it to the user who issued the query.

**Algorithm 1** $k$NN queries in OFT

**input:** query $q$, result set size $k$
**output:** $k$ nearest neighbors

```
queue ⇐ List of (dist, Page) in ascending order by dist
queue.insert(ROOT)    // start with root page of OFT
resultArray ⇐ (dist, Objectfeatures)[k] = [(∞, NULL), ..., (∞, NULL)]
dist_max ⇐ ∞
while queue.hasNext and queue.nextDistance ≤ dist_max do
  p ⇐ queue.pollFirst
  if p.isDataPage then
    for each o in p do    // iterate over all feature vectors in the data page
      exactDist ⇐ Dist(q, o)
      if exactDist ≤ dist_max then
        resultArray[k] ⇐ (exactDist, o)
        resultArray.sort
        dist_max ⇐ resultArray[k].dist
      end if
    end for
  else    // p is index page
    for each descriptor D in p do
      h = mindist(q, D)
      if h ≤ dist_max then
        queue.insert(h, D.childpage)
      end if
    end for
  end if
end while
return resultArray
```

The actual construction of the tree is given in Algorithm 2. We start by running a suitable feature extraction algorithm on the multimedia database that results in a database of feature vectors. For these feature vectors, we run principle components analysis to de-

rive an appropriate ordering of the dimensions. Depending on the dimensionality of the feature vectors, this step may also be used to reduce the overall dimensionality of the representation in the index. Please note that this reduction does not endager the lower bounding property, as the resulting distances may only decrease. Other dimensionality reduction techniques that fulfill this lower bounding property could be used as well. To prepare the feature vectors for separator construction, discretization to a partition of the value range is computed. The partition may be simply equi-width or pre-computed based on an assumption about the feature vector distribution, or even based on an analysis of the actual distribution. The feature vectors are simply mapped to their corresponding ranges as defined in Definition 3. After this step, the feature vectors have been pre-processed for indexing. The index is constructed in a bottom-up fashion, just like any B-tree, R-tree, R*-tree, etc. While the MBRs are constructed as in the R-tree family, the split itself is based entirely on the B-tree-style split to ensure that no overlap is incurred. Separators themselves are constructed like in B-trees as the shortest string that distinguishes between left and right subtree (cf. Definition 2).

**Algorithm 2** OFT construction

**input:** multimedia database $M$
  feature database $F \Leftarrow FeatureExtractor(M)$
  transformed database $T \Leftarrow PCA(F)$
  discretized database $D \Leftarrow Discretize(T)$
  **for each** $o$ in $D$ **do**
    Node $p \Leftarrow OFT.searchNode(o)$
    $p.insert(o)$
    $s \Leftarrow p.parentNode$
    **if** $p.isFull$ **then**
      $p_1, p_2 \Leftarrow p.SeparatorSplit$
      $s.computeMBRs(p_1, p_2)$
      $s.updateSeparators()$
      $s.propagateSplit()$
    **else**
      $s.computeMBRs(p)$
    **end if**
  **end for**

## 5 Experiments

We compare our approach with a recent index approach, the A-tree that combines MBRs with quantized virtual VBRs [SYUK00], and with the R*-tree [BKSS90]. The R*-tree uses floats (four bytes) per dimension to store the continuous values of the MBRs. For the A-tree, we use a discretization parameter of 8, which fared best in our preliminary experiments. All indexing structures are implemented using Java 1.6. We use implementation invariant performance measure like the number of pages accessed or the average capacity of the index pages as well as the number of refinements necessary. Experiments were

conducted on 2.33GHz Intel XEON machines running Windows Server 2008.

We thoroughly evaluate the performance of these approaches on several real world data sets.

- Data set 1 is a subset of the well known Corel database which consists of 59,870 photographs of different scenes.
- Data set 2 is a collection of 107,350 license free photos from the pixelio web page [Pix]. Due to constant updates by users of the web site, this database is very heterogeneous.
- Data set 3 comprises 103,642 images from several TV stations which were captured at an interval of at least 5 seconds.
- Data set 4 is the Hemera photo objects database containing 53,802 images [Hem]. Every object is represented multiple times in different positions.
- Data set 5 is the Aloi database [GBS05] which consists of 10,000 object images, each shown from 72 different directions; hence the database comprises 72,000 images.

We used color histograms in extended HLS color space as feature extraction method to obtain the feature vectors for all databases. The results of the conducted experiments are very similar for the different databases. Thus, we only show a selection of the outcomes.

### 5.1 Structural analysis

In our first experiment, we evaluate our overlap-free indexing structure in terms of its structural properties. We built R*-trees, A-trees, and OFT of different discretization parameter values (4, 16 and 64) and compared them for various dimensionalities on the Corel database. As we can see in Figure 9, the average number of entries per node and therefore also the fanout is substantially larger in our OFT approach. This is due to its discretization. The A-tree, while using quantized virtual bounding rectangles, does not show large capacity of nodes due to the additional overhead incurred from storing both the MBRs and the VBRs (quantized virtual bounding rectangles). In leaf nodes, the difference to the competing algorithms is generally smaller, as we maintain finer discretized representations at this level, which requires more storage overhead. This is done for the sake of reduction of the number of data page accesses. The better the information stored at leaf level, the fewer false hits require data page reads.

We have analyzed the effect of discretization on the number of page accesses in a very high dimensional setting. Figure 10 shows the results for the Corel database and the Pixelio database, respectively, on 60-dimensional color histograms and an index dimensionality of 16. Clearly, R*-trees do not scale to this index dimensionality. However, both the A-tree and our OFT approach fare much better and show reasonable numbers of page accesses for query processing. We can see that the best discretization strategy of our OFT algorithm is using between 8 and 16 equiwidth partitions per dimension. This strategy is slightly better than assigning the same number of partitions equidepth according to the normal

| Tree | Type | Index Dimensionality | | | | |
|---|---|---|---|---|---|---|
| | | 16 | 20 | 24 | 28 | 32 |
| R*-Tree | Non-Leaf | 22.78 | 18.44 | 16.40 | 14.05 | 12.16 |
| | Leaf | 46.08 | 37.74 | 31.74 | 28.01 | 24.36 |
| A-Tree | Non-Leaf | 24.59 | 19.76 | 16.51 | 13.64 | 12.50 |
| | Leaf | 110.66 | 89.09 | 77.15 | 65.50 | 57.01 |
| OFT_s4 | Non-Leaf | 56.25 | 46.08 | 40.56 | 37.00 | 31.03 |
| | Leaf | 133.04 | 108.26 | 92.24 | 80.90 | 71.44 |
| OFT_s16 | Non-Leaf | 55.50 | 47.63 | 40.26 | 38.50 | 37.38 |
| | Leaf | 134.84 | 114.25 | 99.12 | 86.39 | 76.26 |
| OFT_s64 | Non-Leaf | 53.12 | 50.20 | 49.75 | 41.17 | 40.63 |
| | Leaf | 140.87 | 119.26 | 100.28 | 85.52 | 77.55 |

Figure 9: Capacity

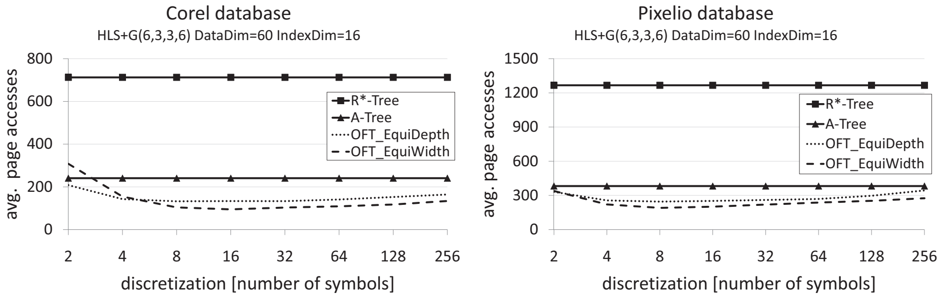



Figure 10: Effect of discretization

distribution. Apparently, the data does not follow a Gaussian pattern. Most importantly, our best technique saves more than 60% of the page accesses required by the A-tree.

## 5.2 Scalability

We further study the scalability of the OFT indexing structure compared to its competitors. Figure 11 illustrates the number of page accesses required for different index dimensionalities on the Hemera and Pixelio database. Starting from an original database of 60-dimensional feature vectors, we reduce them to different lower dimensional representations using PCA (principle components analysis, as discussed in Section 4.3). As we can see immediately from the graphs, dimensionality reduction is beneficial for all index structures as it reduces overlap in the competing approaches and reduces the storage usage in our OFT approach. Using a discretization to 16 equiwidth partitions clearly outperforms the competing techniques by at least an order of magnitude.

We further evaluate the performance on a data set of much higher feature vector dimensionality extracted on the Corel and TV databases. Figure 12 depicts the number of page accesses for different index dimensionalities starting from an original representation of

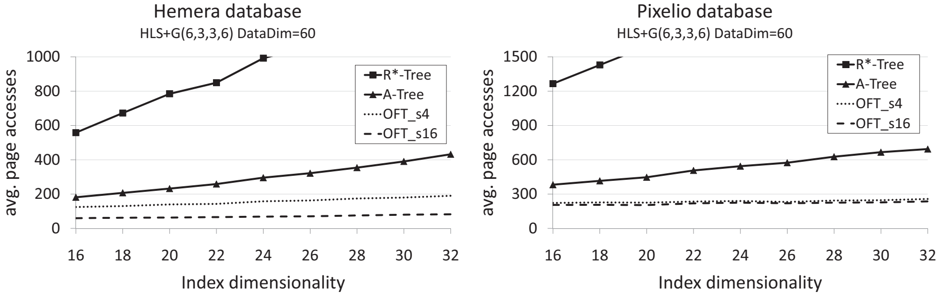



Figure 11: Variations of the index dimensionality

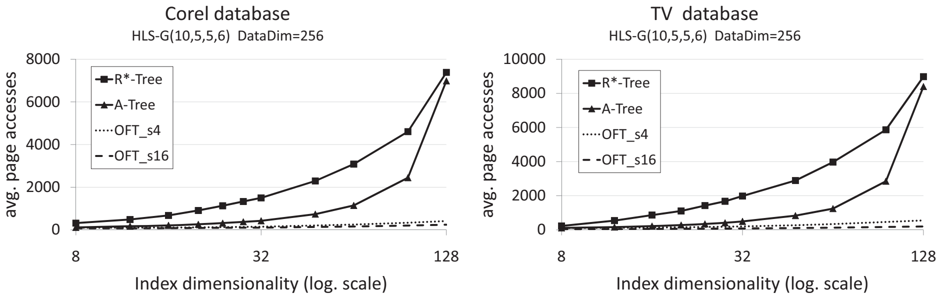



Figure 12: Variations of the index dimensionality in extreme cases

dimensionality 256. As we can see, R*-tree and A-tree suffer from increasing overlap in higher dimensionalities. Our OFT approach, however, scales extremely well. Even for this high dimensionality, the number of page accesses remains low.

To understand the effects of the difference in page accesses, we provide a more detailed experimental evaluation on the Corel, Pixelio, Hemera, Aloi and TV databases. Figure 13 therefore distinguishes between the number of index pages accessed and the number of actual data pages with the original representation. For all databases, we see that the number of index page accesses clearly dominates the number of data page accesses. OFT therefore outperforms A-trees and R*-trees mainly due to its informative index nodes that provide relevant information for pruning.

The effect of data dimensionality is analyzed in our next experiment. Figure 14 shows the results of fixed index dimensionality under varying original feature vector dimensionalities. Once again, we observe that R*-trees do not scale with the dimensionality of the data. Compared to the A-tree, we note that our best discretization scheme requires about one order of magnitude fewer page accesses. Moreover, the performance is very stable with increasing data dimensionality.

Finally, scalability with respect to database size is demonstrated. Figure 15 illustrates for a large-scale database, consisting of all 5 data sets, and a dimensionality of 16 and 32, respectively that R*-trees and A-trees do not scale very well with increasing database

Corel database
HLS+G(6,3,3,6) IndexDim=16
OFT_s16 79,88 15,56
A-Tree 219,2 21,52
R*-Tree 660,6 51,88
0 200 400 600 800
Index Pages
Data Pages

Pixelio database
HLS+G(6,3,3,6) IndexDim=16
OFT_s16 165,68 39,72
A-Tree 336,2 46
R*-Tree 1153,36 112,72
0 200 400 600 800 1000 1200 1400
Index Pages
Data Pages

Hemera database
HLS+G(6,3,3,6) IndexDim=16
OFT_s16 52,68 6,76
A-Tree 173,8 7,92
R*-Tree 551,68 6,68
0 100 200 300 400 500 600
Index Pages
Data Pages

Aloi database
HLS+G(6,3,3,6) IndexDim=16
OFT_s16 22,12 4,28
A-Tree 61,04 1,4
R*-Tree 104,76 1,4
0 20 40 60 80 100 120
Index Pages
Data Pages

TV database
HLS+G(6,3,3,6) IndexDim=16
OFT_s16 58,04 2,12
A-Tree 247,84 2,28
R*-Tree 1014,48 2,08
0 200 400 600 800 1000 1200
Index Pages
Data Pages

Figure 13: Index and data page accesses for data dimensionality = 60

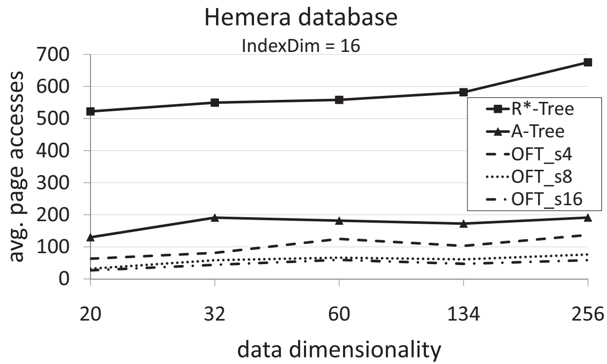
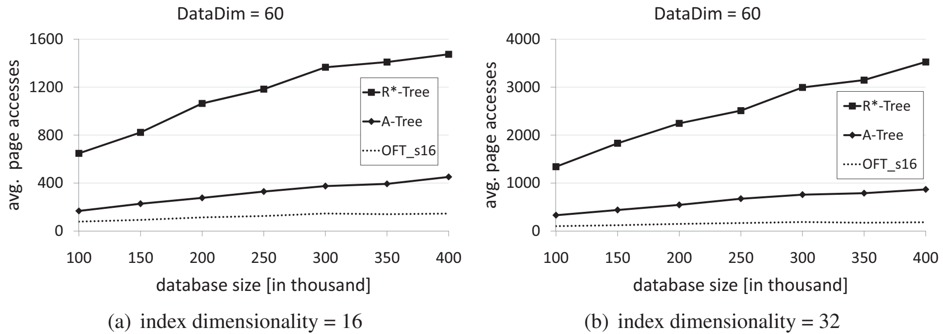



Figure 14: Influence of the data dimensionality




(a) index dimensionality = 16

(b) index dimensionality = 32

Figure 15: Scalability with respect to the dababase size

size. OFT, on the other hand side, shows remarkably good scalability with respect to the database size, clearly outperforming its competitors by at least one order of magnitude also in this respect.

# 6 Conclusion

Multimedia similarity search in large databases is a challenging task, especially for very high-dimensional feature vectors. In this work, we have proposed an approach that uses multimedia indexing structures to improve the overall query processing efficiency. A major obstacle for multimedia indexing structures is the fact that with increasing dimensionality, overlap leads to a degeneration of indexing structures. We therefore use the concept of separators to guarantee overlap-free splits. Separators require discretized values and ordering of dimensions, for which we presented a transformation that allows lossless query processing in an efficient multistep filter-and-refine algorithm. Our experiments demonstrate that our approach successfully scales multimedia similarity search to large and high-dimensional multimedia databases, clearly outperforming competing indexing structures.

# References

[AKAS08] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The TS-Tree: Efficient Time Series Search and Retrieval. In *EDBT*, 2008.

[AWS06] Ira Assent, Marc Wichterich, and Thomas Seidl. Adaptable Distance Functions for Similarity-based Multimedia Retrieval. *Datenbank-Spektrum Nr. 19*, pages 23–31, 2006.

[BBJ+00] S. Berchtold, C. Böhm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces. In *ICDE*, pages 577–588, 2000.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. In *VLDB*, pages 28–39, 1996.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[BM70] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access,*, pages 107–141, 1970.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD*, pages 357–368, 1997.

[BU77] Rudolf Bayer and Karl Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[CPZ97] Paulo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB, Athens, Greece*, pages 426–435, 1997.

[Fal96] Christos Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[FG99] Paolo Ferragina and Roberto Grossi. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of the ACM*, 46(2):236–280, 1999.

[GBS05] J. M. Geusebroek, G. J. Burghouts, and A. W. M. Smeulders. The Amsterdam Library of Object Images. *Int. J. Comput. Vision*, 61(1):103–112, 2005.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, 1984.

[Hem] Hemera Photo Objects. `http://www.hemera.com`.

[Jol86] Ian T. Joliffe. *Principal Component Analysis*. Springer, New York, 1986.

[MNPT06] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. *R-Trees: Theory and Applications*. Springer, London, 2006.

[Pix] License free photo database. `http://www.pixelio.de`.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Elsevier, 2006.

[SK98] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, pages 154–165, 1998.

[SYUK00] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, and Haruhiko Kojima. The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. In *VLDB*, pages 516–526, 2000.

[TTSF99] C. Traina, A. Traina, B. Seeger, and C. Faloutsos. *Slim-trees: High Performance Metric Trees Minimizing Overlap Between Nodes*. Springer, 1999.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB*, pages 194–205, 1998.

[Zez06] P. Zezula. *Similarity Search: The Metric Space Approach*. Springer, 2006.