University of Alberta

Exploring the Color Histogram's Dataspace for Content Based Image Retrieval

by

Alex Coman

Technical Report TR 03-01
January 2003

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Exploring the Color Histogram's Dataspace for Content Based Image Retrieval

Alexandru Coman

Abstract

With the growth of available information in digital format, indexing has drawn much attention as a viable solution to reduce retrieval time when searching large databases. There are many indexing techniques available nowadays, but, as they were developed with general goals in mind, they do not perform at their best in many cases.

In this report we explore the use of prior knowledge about the data to be indexed (e.g. its spatial distribution) in order to enhance the performance of the indexing structure. Based on a concrete example – the constraint existing in color histograms – we explore how such property induces other data constraints. We design a specialized index structure to take advantage of the induced data constraints. We also propose a new lower bound distance between data points and Minimum Bounding Rectangles to be used when indexing histograms with existing indexing structures.

The proposed indexing technique is shown to be efficient and stable with respect to variations in number of data objects, page size, feature vector dimensionality and variation in number of nearest neighbors to be searched for.

1 Introduction

1.1 Motivation

In the last few years great effort has been put into analyzing and indexing the large number of images that have become available with the boom of the Internet. Through digital photography and image scanners, many personal photo albums have been transformed into private digital library collections. Even before the expansion of digital technologies in private usage, large collections of images were available for professionals in fields such as design, photography, astronomy, publishing and others. Efficient browsing and retrieval are important issues for all these people, for private or professional use.

In contrast to standard database systems, similarity search is the main functionality required by most multimedia applications. While textual information may be directly stored and queried, multimedia data is handled through feature transformations, where multimedia object's properties are usually transformed into a high-dimensional point, also called feature vector. The similarity of two multimedia objects is then decided based on the similarity of their feature vectors. While the image processing researcher is challenged by the feature extraction task, what makes this field interesting for the information retrieval researcher is, among other factors, the complexity and dimensionality of the feature vectors. Common information retrieval techniques, effective with other types of data, cannot be used in this case, and neither are trivial adaptations of those techniques. One challenging task for image information retrieval has proved to be handling the large dimensionality of the feature vectors.

In the case of image objects, these vectors are designed to capture the image content and therefore the retrieval process is called Content Based Image Retrieval (CBIR). Typically, a CBIR system extracts some visual features from a given query image which are then compared with the features of images stored in the database. The similarity is based on the abstracted image content rather then on the

images themselves. Due to the exponential growth in the number of digital images available, human-assisted annotation is not feasible. A feature that is easy and fast to extract and also captures reasonably well the image content is color. It is common to use a Global Color Histogram (GCH) to represent the distribution of colors within an image [8]. It has the desired characteristic of low complexity for extraction and invariance to scaling and rotation. In Section3.1 we will introduce in more details color models and image colors representations, as our work presented in this report uses Global Color Histogram as the feature extracted from images. Some other features that have been used for image representation and retrieval purposes are texture [33, 50], shape [28, 35], spatial relationships [9, 24] and others. It is also common to use a combination of features to improve the quality of retrieval.

There are many CBIR systems available, some commercial, other for research purposes. Several of them are: IBM's QBIC [38, 43], Virage's VIR Image Engine retrieval system [3, 54], Excalibur's RetrievalWare [17, 44], MIT's Photobook [41, 42], Columbia University VisualSEEk [51, 55], University of Alberta's BSIm [14, 36] and IBM/NASA's Satellite Image Retrieval System [34].

With the growth of available information in digital format, indexing has drawn much attention as a viable solution to reduce retrieval time when searching large databases. While a sequential scan is efficient for CBIR from small image collections, efficient high-dimensional indexing techniques need to be explored for efficient CBIR from large collections. One of the main issues the indexing techniques for Image Retrieval have to deal with is the large dimensionality of the feature vectors. The history of multidimensional indexing techniques can be traced back to middle 1970's when the quad-tree and k-d tree were first introduced [47] for 2-dimensional data. Improved structures such as the R-tree [25] and variants were introduced in the following years, but not even the best dynamic variant, the R*-tree [4], performs well for more than 20 dimensions. Section3.2 presents a review of the most important research work in the indexing of high dimensional data, from the R-tree to state-of-the-art techniques developed recently. The proposed indexing techniques are very general with respect to data to be indexed and do not always scale well to the high dimensionality of the feature vectors used in Image Retrieval. This is due to a number of effects collectively referred to as "the curse of dimensionality". For Global Color Histograms, however, some of these effects do not occur due to the inherent constraint on the values of a histogram; the sum over all values of the normalized histogram feature vector must be equal to 1. This constraint leads to other constraints on the data's dimensional distribution, maximum distance between two objects and others. The careful observation and understanding of these constraints allow us to design a more specialized index structure for normalized histograms.

1.2 Our Contribution

In this report we will show how the efficiency of retrieval can be improved by taking advantage of constraints inherently existing in image feature vectors at both index construction and search time. Although we will use GCHs as our image feature of choice, similar constraints may exist in other features used in CBIR or other fields. For example, histograms can also be used to represent and compare the shapes of objects [2, 19].

Firstly, we refine the distance function that computes the minimum distance between the query point and a potential point inside a Minimum Bounding Rectangle (MBR). Since the minimum distance is used in pruning the search space, it is important to approximate it as accurately as possible. The improved distance function will help reduce the number of Minimum Bounding Rectangles (MBRs) searched during retrieval, as well as changing the MBRs search order, as we will present in Section4.5. Thus, we improve the search efficiency. This function can be used to improve pruning in combination with any existing indexing techniques that use Minimum Bounding Rectangles as bounding regions.

Secondly, based on our observations regarding the data constraints imposed by the use of Global Color Histograms as feature vectors as well as the data distribution (synthetic and real data), we

propose a new split policy. To show its efficiency we will use it during bulkload construction of the index structure. This basic index structure is an extreme case of the X-tree index for high dimensionality [7]. We will also show that a strategy that proves efficient for uniformly distributed data is not as good for real data. We show how the split policy can be adapted for real data collections in order to improve the efficiency of our index structure.

2 Report Outline

The report is organized as follows. Section 3 contains an overview of the current research in information indexing as well as several color models and color properties representations. Section 4 gives a description of our observations regarding the existing data constraints induced by using normalized histograms. We will also introduce a variant of an indexing technique that will take advantage of these constraints at index creation time. We will describe an accurate distance that helps improve pruning the search space at retrieval time. The evaluation of the improvements obtained by using the new proposed technique over state-of-the-art index methods will be presented in Section 4. Finally, Section 5 concludes the report and states some possible directions for future research.

3 Related Work

In the following sections, some information regarding color descriptors and possible similarity measures is presented, as well as an overview of the most popular and efficient indexing methods applied to color indexing over time.

3.1 Color Representation and Similarity Models

Color is an extensively used visual attribute in image retrieval that often simplifies object identification and extraction [23]. This attribute is very convenient since it provides multiple measurements at a single pixel level on the image, often enabling classification to be done without the need of complex spatial decisions [49], like objects' shape and positioning. Its extraction is very fast, making it the feature of choice in large dynamic collections, when image features have to be extracted in real time. Traditionally, color histograms have been used as color feature vectors. Some advantages are that they are robust to image rotation and scaling and image similarity can be computed using simple metric distances (e.g: Hamming or Euclidean distance). In the following paragraphs, color representation and some similarity models will be presented in overview.

3.1.1 Color Spaces

The most commonly used color representation model is RGB, which is composed of three primary colors: Red, Green and Blue. This model has both a physiological foundation and a hardware related one. Results from neurophysiology [8] show the existence of three distinct types of cones in the human retina used for capturing what humans define as color. The Red, Green and Blue colors correspond to the location of the maximum of cone responses to a monochromatic stimulus. The RGB model is also most frequently used to reproduce colors in optic devices such as TV monitors and computer screens. These three colors are called primary colors and are additive. By varying their combinations, other colors can be obtained [21]. This color model can be represented as a unit cube with black (0,0,0) and white (1,1,1) as extremes of the scales, red (1,0,0), green (0,1,0) and blue (0,0,1) as primary colors and cyan (0,1,1), magenta (1,0,1) and yellow (1,1,0) as secondary colors (Figure 1). A drawback of this

model is that it is not perceptually uniform; that is the calculated distance between two colors does not truly correspond to the perceptual difference [32].

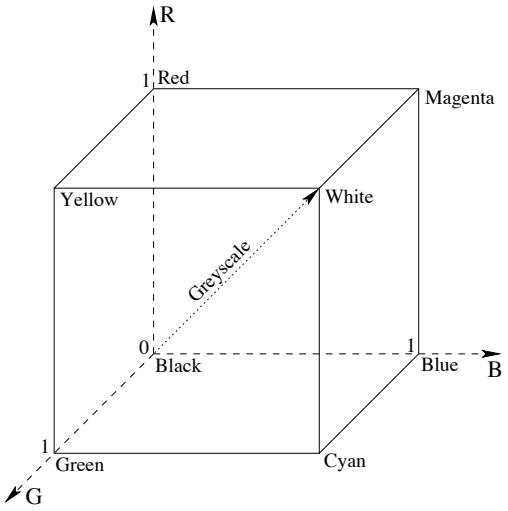


Figure 1: The RGB color space

Another color model based on the same unit cube as RGB is CMY. It is based on the secondary colors of the RGB space and it is mainly used for color printing [8]. The Cyan, Magenta and Yellow are the complements of Red, Green and Blue. As compared with the RGB model, the CMY has the limitation that none of these three colors is a pure color, and they are always adulterated by a certain proportion of each other. It is impossible to create a pure black color using this model. In order to overcome this problem, the CMY color model is extended to another model, referred as CMYK, which uses black (K) as the fourth color [1].

A color model derived on how colors appear to a human observer is the HSB (Hue, Saturation, Brightness) model. Hue describes the wavelength of the color percept. Saturation indicates the amount of white light present in a color. Brightness represents the intensity of a color. The HSB model is based on experiments analyzing human reactions to opponent primaries[1] [8] and on the observation that opponent hues (like Yellow and Blue) cancel each other when superimposed. This model can be represented as a cylinder, but it is usually represented by a double hexagonal cone. Brightness (B) represents the vertical axis, saturation (S) is represented on a side of the hexagonal cone and hue (H) is an angle around the vertical axis. Another color coding scheme belonging to the same class as HSB is HSV (Hue, Saturation, Value). The representation of the HSV space (Figure 2) is derived from the RGB space cube, with the main diagonal of the RGB model, as the vertical axis in HSV [31]. As saturation varies form 0.0 to 1.0, the colors vary from unsaturated (gray) to saturated (no white component). Hue ranges from 0 to 360 degrees, with variation beginning with red, going through yellow, green, cyan, blue and magenta and back to red. These color spaces are intuitively corresponding to the RGB model from which they can be derived through linear or non-linear transformations [8].

Another uniform color space based on human perception is L*u*v*. It is device-independent, based on the opponent color theory of human vision and highly approximates color difference as perceived by humans [8]. It was recommended by CIE (Commision Internationale e l'Eclairage) for quantifying differences in monitor displays. Lightness L* is perceptually based on brightness and u* and v* are

[1]lights of a single wavelength

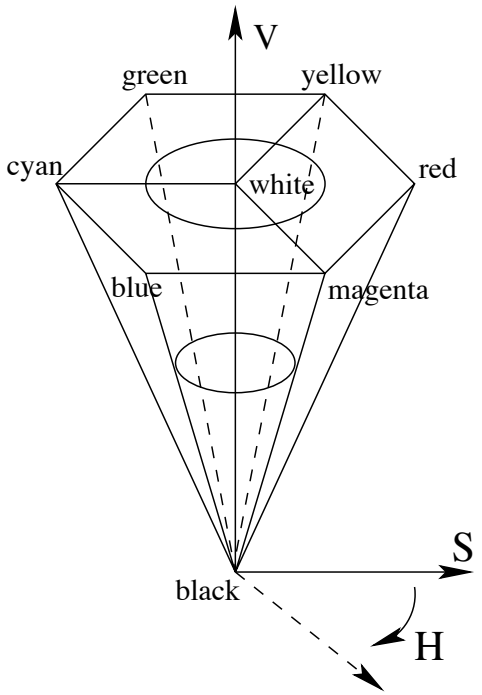


Figure 2: The HSV color space

chromatic coordinates. In the L*u*v* color model, Red is the best represented color, Green is moderately represented, while Blue is poorly represented.

The selection of a color space is very important for deriving useful image related color information. Several researchers have evaluated various color models for the purpose of image retrieval under varying sets of image conditions [22]. It has been shown that the RGB color model closely corresponds with the physical sensors of the human eye, although the human perception is more accurately reflected using the HSV color space. There is no clear proof in the specialized literature of one model being better than the other in all aspects. In the Color Based Information Retrieval research, the RGB color model is most frequently used, as it is easy to understand and extract. We use the RGB model in our research, but the proposed techniques would equally apply well when using any other color model.

3.1.2 Color Properties Representation

Color histograms are the most common way of describing low-level color properties of images [8]. A color histogram is represented by a set of bins, where each bin represents one color. It is obtained by counting the number of pixels that fall into each bin based on their color. Since the typical computer represents color images with up to 16,777,216 colors (RGB model, 256 Red levels x 256 Green levels x 256 Blue levels), this process generally involves substantial quantization of the color space. Color histograms could be generated either as three independent color distributions (one for each of the RGB primary colors) or, more frequently, as a joint distribution of all three primary colors. Usually, the obtained color histogram is normalized with total number of image pixels. In normalized histograms, each bin represents the percentage of pixels of the bin corresponding color found in the image.

When only one histogram is generated for the entire image, it is called Global Color Histogram

(GCH). Assuming a D-color space, a GCH is then an D-dimensional feature vector (h_1, h_2, ..., h_D), where h_i represents the normalized percentage of pixels of color c_i found in the target image. Formally, each component c_i can be defined as a unique combination of Red-Green-Blue values. The GCH by itself does not capture any spatial information about the image, so that very different layouts may have similar representations. For example, a large red object on blue background could have the same GCH representation as many pixel-size blue objects on red background. When only GCH is used as feature vector describing an image, the retrieval of similar images is based on similarity of their GCHs. Often, the similarity metric used between two image GCHs is the Euclidean distance, that is defined as:

$$d(Q,I) = \sqrt{\sum_{i=1}^{D}(h_i^Q - h_i^I)^2} \tag{1}$$

where Q represents the query image, I is one of the images in the dataset and h_i^Q and h_i^I represent the histogram value for the same coordinate (color) i in Q, respectively I. A smaller distance reflects a closer similarity match. In fact, the color histogram feature vectors are usually mapped onto points in an D-dimensional metric space, and similar images would therefore appear close to each other, with a smaller Euclidean distance between them.

Another popular distance function used to measure similarity between color histograms is the Hamming distance, that is defined as:

$$d(Q,I) = \sum_{i=1}^{D}|h_i^Q - h_i^I| \tag{2}$$

Hafner et al. propose in [26] a quadratic form distance function that they suggest it captures better the similarity between color histograms. The new distance function is defined as:

$$d(Q,I) = \sqrt{(Q-I)^t A (Q-I)} \tag{3}$$

where $A = [a_{ij}]$ is a weight matrix and a_{ij} represents the extent to which histogram's bins (colors) i and j are perceptually similar to each other. When compared with the Hamming or Euclidian distances, the quadratic distance allows similarity matching between different colors.

In order to capture some spatial information through color histograms, local color histograms (LCH) [52] can be used. An image is divided into regions and a color histogram is computed for each region. Thus, an image divided into 16 regions will be represented by 16 LCHs, one for each region.

A different type of histogram that incorporates some spatial information about an image is proposed in [40]. Each pixel is classified in a given color bin as either coherent or incoherent. A color's coherence is defined as the degree to which pixels of that color are members of a large similarly colored region. A Color Coherence Vector (CCV) stores the number of coherent (α_i) versus incoherent (β_i) pixels for each color i. $\alpha_i + \beta_i$ represents the total number of pixels of color i in the image, as represented in GCHs. In CCV, the feature vector of an image is described as I= $\langle(\alpha_1,\beta_1),...,(\alpha_D,\beta_D)\rangle$, with D being the size of the color space. Using this notation, the Hamming distance between Q and I using Global Color Histograms representation is:

$$d_{Hist}(Q,I) = \sum_{i=1}^{D} \mid (\alpha_i^Q + \beta_i^Q) - (\alpha_i^I + \beta_i^I) \mid \tag{4}$$

When using CCV representation, the distance is defined as:

$$d_{CCV}(Q,I) = \sum_{i=1}^{D}(\mid \alpha_i^Q - \alpha_i^I \mid + \mid \beta_i^Q - \beta_i^I \mid) \tag{5}$$

It is shown that $d_{Hist}(Q, I) \leq d_{CCV}(Q, I)$ [40] and therefore one can have a better distinction among similar images for the case when the distance between GCHs would produce many similarity ties. Experimental results show that Color Coherence Vectors can give superior results when compared to Global Color Histograms.

In [36], a different approach is proposed that emphasizes less dominant colors, while still taking into account major colors of an image. The approach is based on the discretization of colors into binary bins which improves the retrieval performance and significantly saves storage space. After each image is quantized into a fixed number of colors $I = (c_1, c_2, ..., c_D)$, each color element c_i is further discretized into T binary bins $B_i = b_i^1 b_i^2 ... b_i^T$, where only one bit can take value 1. If the discretization function assigns equal color value ranges to each bin, the arrangement is called Constant Bin Allocation (CBA), while if it uses variable color ranges it is called a Variable Bin Allocation (VBA) approach. The VBA method gives advantage to less dominant colors by discretizing the space with larger ranges assigned to a bin for the lower values. A signature of an image will then be represented by the following bit-string: $S = b_1^1 b_1^2 ... b_1^T b_2^1 b_2^2 ... b_2^T ... b_D^1 b_D^2 ... b_D^T$, where b_i^j represents represents the j^{th} bin of the c_i color element. Since each bin is represented by just one bit, the obtained signature is very compact. For example, by using D = 64 colors, F = 4 bytes/float value and T = 10 bins, the image signature requires 80 bytes = 64 colors x 10 bits, while GCH would require 256 bytes = 64 colors x 4 bytes/float and CCV 512 bytes = 64 colors x 2 histograms x 4 bytes/float. That is a saving of over 68% over GCH and 84% over CCV. The distance between two images Q and I is defined as:

$$d(Q, I) = \sum_{i=1}^{D} \mid (pos(B_i^Q) - pos(B_i^I) \mid \tag{6}$$

where $pos(B_i^R)$ gives the position of the set bit within the set of bits B_i of image R. Experimental results show that VBA method can equal and sometimes outperform the use of GCHs. Beside the advantage of great savings in storage space, signatures can be indexed efficiently using signature based methods such as the S-tree [37].

In [53] the authors propose a method based on color moments. For each color channel (H, S and V in their case), they extract the first three color moments: average, variance and skewness. Each image is described by 9 features (3 channels x 3 moments). They also propose a similarity distance based on these features, where each color moment has a different weight. They argue that the index is small and the retrieval process is fast and effective. A drawback of this technique is that the choice of the weights in the distance function is dependent on the data set and highly affects the retrieval results.

Jacobs at al. have suggested in [27] the use of quantized 2-dimensional wavelet decomposition by applying the Haar wavelet transform to images. Similarity is determined by checking how many significant wavelet coefficients on the query image and dataset image are close to each other. As wavelet coefficients capture information of image content independent of original image resolution, the query and dataset images may have different resolutions, without affecting the quality of retrieval.

In the VisualSEEk system [51], Smith and Chang use color sets to locate regions within a color image. Color sets are binary vectors that correspond to a selection of colors. It is assumed that image regions have only a few dominant colors. Similarity between two images is determined by verifying the presence of a color set in an image region. For single region queries, the overall distance is computed as the weighted sum of distances between color sets, spatial locations, areas and spatial extents. The best match minimizes the overall distance.

A pyramidal multiresolution representation of color regions is used in the Picasso system [8, 10]. Region segmentation is performed by iteratively aggregating uniform color patches. Image segmentations obtained with this process are organized in a pyramidal schema. Each segmentation is represented

through a graph, where each node represents color uniform regions at a certain image resolution. Global color vectors are used to quickly extract candidate images containing regions with the same colors as the query. Then, candidate images are analyzed by inspecting their pyramidal representation from top to bottom, to find the best matching region for each region in the query. A similarity score is computed based on the matching score of each region.

3.2 Indexing Techniques

In order to increase the efficiency of retrieval, indexing techniques are used. Since we are generally dealing with a large number of data objects, these objects are kept on secondary storage (disk). The smallest unit a disk is logically partitioned into is called disk page (in the following referred as page). We can only access whole multiples of pages. Since secondary storage operations are very time costly compared with operations in main memory, the main concern is to minimize the number of disk pages that have to be accessed during retrieval.

Before proceeding to the presentation of some indexing techniques, we will first introduce two common types of queries in image information retrieval. We assume a generic tree-like index structure (as in Figure 3) where the leaf nodes keep the indexed objects and each internal node contains a bounding region for all objects contained in its sub-tree.

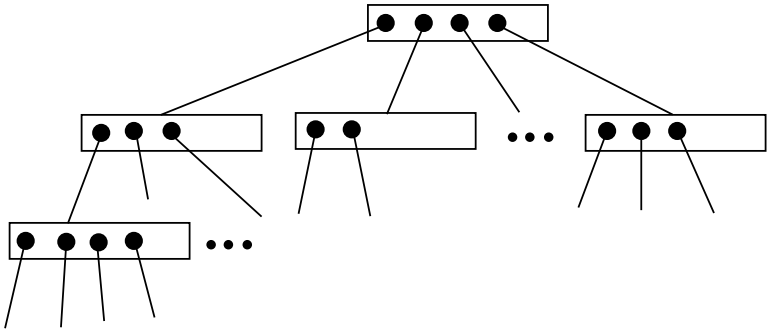
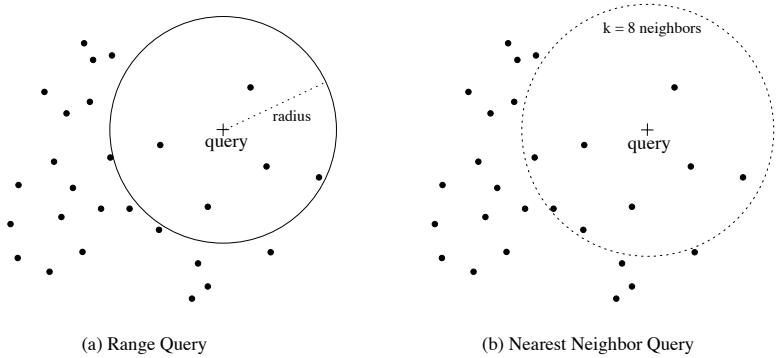


Figure 3: Generic tree index structure

Firstly, there is the *Range Query*. In a range query, a radius is given together with the query object (note that in the following the terms object and point will be used interchangeably since an object is represented by a feature vector which is a point in the feature space). The expected result is the



(a) Range Query (b) Nearest Neighbor Query

Figure 4: Query types in information retrieval

retrieval of all points closer than the given radius from the query object (Figure 4 (a)). The number of returned objects depends on the radius and, for a fixed radius, on the objects distribution in space. It could be null, if the radius is too small, or equal to the number of indexed objects, if the chosen radius is too large. An advantage of this type of query is that a maximum query to object distance is known. Therefore the search space can be pruned early without loosing any objects from the answer set. For a pseudo-code version of the range query search algorithm, see Figure 5. Note that the `Insert` procedure keeps the lists sorted, *dist* computes the distance between two data points and *mindist* computes the distance between a point and the bounding region of a node.

```
Input  : Query Object q, Radius r, ActiveNodeList anl = [root],
         RangeList rl = [NULL]
Output: RangeList rl

while anl Not Empty do
    Node N = anl.RemoveFirst;
    if N Is LeafNode then
        for each Object o in N do
            if dist(q,o) ≤ r then
                Insert o in rl;
    else
        /*Non-Leaf Node*/;
        for each ChildNode c in N do
            if mindist(q,c) ≤ r then
                Insert c in anl;
```

Figure 5: Range Query Search Algorithm

Another type of frequently used query is the *k-Nearest Neighbor Query*. In this type of query, the desired number of neighbors (k) is supplied with the query object. The expected result is a list of k-nearest neighbors of the query object (Figure 4 (b)). In this case, the distance from the query object to the furthest k^{th}-neighbor object is unknown *a priori*. Therefore, pruning the search space has to be done dynamically, based on the distance to the k^{th}-nearest neighbor encountered in intermediate steps of the search. Processing such a query requires a substantially different search algorithm than for range query [5, 46]. It is also more costly than range query search for most indexes. For a pseudo-code version of the basic *k-Nearest Neighbor* query search algorithm, see Figure 6. Note that the `Insert` procedures keeps the lists sorted, the *NNList* keeps just k elements and *knn.maxdist* represents the maximum nearest neighbor distance found.

An important issue that affects the performance of these search algorithms is the amount of overlap existing among the tree nodes. Overlap is the percentage of the space volume that is covered by more than one node sub-tree bounding region. This affects the query performance since during query processing the overlap of bounding regions results in the necessity to follow multiple paths, covering same regions of space more than once.

It is important to note that the more accurate the *mindist* distance, the better te prunning of sub-trees during the tree traversal, i.e., the faster the query processing (this is discussed in more details in Section4.5.

```
Input  : Query Object q, Neighbors k, ActiveNodeList anl = [root],
         NNList knn = [NULL]
Output: NNList knn

while anl Not Empty do
    Node N = anl.RemoveFirst;
    if N Is LeafNode then
        for each Object o in N do
            if dist(q,o) < knn.maxdist then
                Insert o in knn;
                Update knn.maxdist;
                Prune anl using knn.maxdist;
    else
        /* Non-Leaf Node */;
        for each ChildNode c From N do
            if mindist(q,c) < knn.maxdist then
                Insert c in anl;
```

Figure 6: k-Nearest Neighbor Query Search Algorithm

3.2.1 Data Partitioning Indexes

The most popular techniques for indexing data try to group or organize the data objects based on their distribution. The data space is partitioned into cells and a disk page is allocated to each cell. These cells are organized into an index tree which is used to help prune the search space during query processing. There are many such methods and some of the most commonly used will be presented in this section. They are the R*-tree, TV-tree, SS-tree X-tree and SR-tree. We will also introduce bulkload index construction, that helps improve the efficiency of these index structures when the dataset to be indexed is known in advance.

R*-tree The R*-tree [4] is a multidimensional index structure designed for indexing D-dimensional rectangular data based on their spatial location. It is one of the most successful versions of the R-tree [25] index introduced in 1984 by Gutmann . The R-tree is a height-balanced tree corresponding to a multilevel hierarchy of rectangles. Therefore it is only suitable for rectangular and point (which can be considered as a degenerated rectangle) objects.

We will introduce first the basic R-tree structure (Figure 7), since the R*-tree and many other indexing structures start from the same basic structure followed by enhancements or adaptations of it. A leaf node (Figure 7, nodes A to I) contains entries of the form (ObjP, Rect) where ObjP represents a pointer to the actual spatial object (or just a database record describing the object) and Rect is the smallest enclosing rectangle of the indexed object. All leaves are on the same level. A non-leaf node (Figure 7, nodes 1, 2, 3 and root) contains entries of the form (ChildP, MBR) where ChildP represents a pointer to a child node in the tree hierarchy and MBR is the Minimum Bounding Rectangle of all objects indexed in the subtree rooted at the child node. Therefore, a non-leaf node contains the MBR of all rectangles of the objects contained in the leaves of subtree. The root node contains the MBR of all the indexed objects. Each node corresponds to a disk page. If nodes have many entries, the tree is very wide, and almost all the storage space is used for leaf nodes containing index records [25].

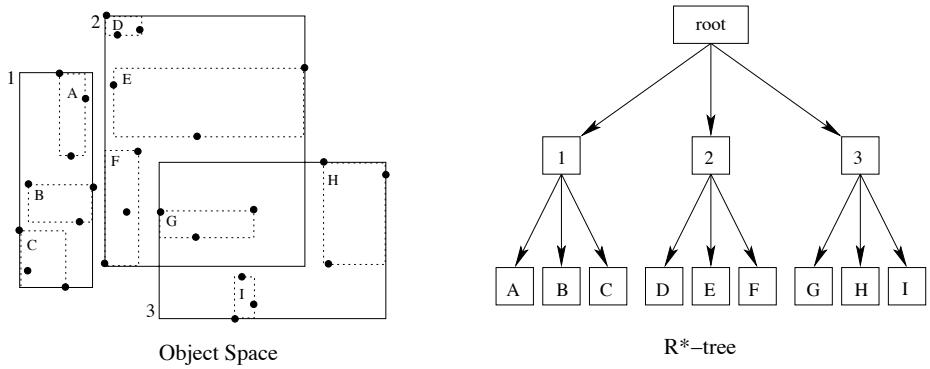


Figure 7: The R-tree structure

This indexing structure is dynamic, insertions and deletions can be intermixed with queries and no periodic global reorganization is required [4]. One drawback of this structure is that it may have highly overlapping directory rectangles (MBR). More overlap means more paths have to be searched to find one (exact match) or more objects (for nearest neighbor and range queries) as the query point will be either inside or close to many MBRs. In order to reduce overlapping, the original R-tree tries to minimize the MBR enlargement area during insertions and minimize the total area of the new nodes MBR during splits. The R*-tree improves the performance of the R-tree by introducing heuristics for minimizing the area, margin and overlap of MBRs in the case of insertions and splits. It also introduces the concept of *Forced Reinsert*, which in some cases avoids splits by reinserting the node overflow in the structure and therefore reorganizing the structure dynamically. This strategy reduces the impact of objects insertion order on structure organization. Another advantage of Forced Reinserts is that it provides a better storage utilization than the original R-tree. Due to possible overlap of rectangles, the search time depends not only on the tree height, but also on the amount of overlap. Although the R-tree was originally designed for rectangular objects, the authors show in [4] that this structure is also effective as a point access method.

The R*-tree is suitable for a small number of dimensions, but its performance decreases rapidly with increasing dimensionality [7]. Since for each dimension two values are used to represent the spatial extension of an MBR, only a few MBR can be kept on a disk page and therefore the fanout (or branching factor, describing the maximum number of entries that a node can have) is too small for an efficient search. Another disadvantage is that MBRs' overlap increases rapidly with the dimensionality, highly reducing the pruning capabilities.

TV-tree The first method proposed specifically for indexing high-dimensional data is the TV-tree [30]. This structure tries to improve the performance of R*-tree for high-dimensional feature vectors by employing a reduction in dimensionality with the use of a shift of active dimensions (Figure 8). An active dimension is a dimension that can be used for discriminating the objects at a certain level in the tree. All dimensions are initially ordered (based on their importance - defined by the user/specialist) and shifted (activated) towards the root of the tree. This shift occurs when feature vectors in a subtree have the same coordinate on the most important active dimensions. Then, those dimensions are made inactive and less important dimensions are activated for indexing. The number of dimensions used for indexing is adapted to the number of objects to be indexed and to the level of the tree. This way the nodes closer to the root use only a few dimensions for indexing. Therefore, they can store more child nodes and enjoy a high fanout even if the objects' feature vectors have many dimensions. As the

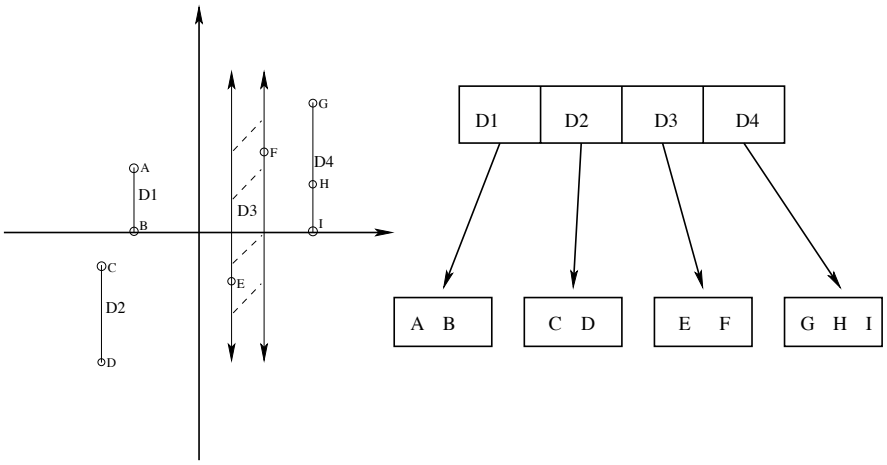


Figure 8: Example of a TV-tree with 1 active dimension

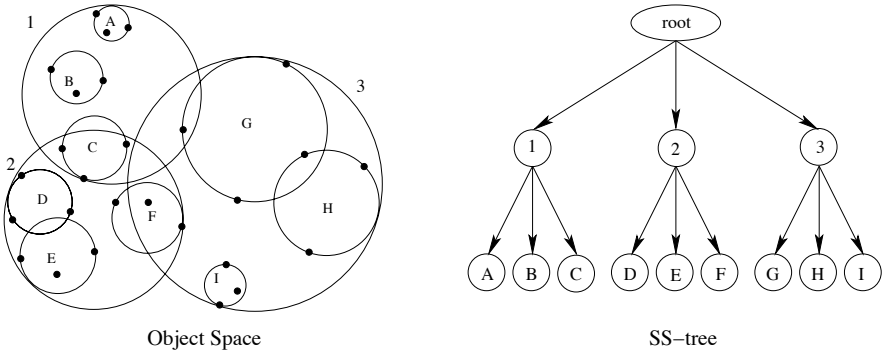


Figure 9: The SS-tree structure

indexing descends the tree, more and more dimensions are used for discriminating the indexed objects. The performance comparisons presented in the literature show that the TV-tree is much more efficient than the R*-tree.

The main drawback of this approach is that it is based on two assumptions [57]. The first assumption is that an order based on importance can be found for the dimensions of the feature vectors. The second assumption is that the feature vectors can be exactly matched based on some dimensions, especially on the first few important dimensions. If the first assumption may hold by finding an appropriate transformation of the feature vectors, the second one does not hold for real-valued feature vectors since their coordinates generally have great diversity. This assumption is valid only if the feature vectors take values in a (small) discrete set. For visual information retrieval, real-valued feature vectors are the most common case and therefore the efficiency boost of the TV-tree is lost. In this case the TV-tree is reduced to an index on only the first few dimensions. Thus, the effectiveness of this indexing structure is dependent on the application.

SS-tree The SS-tree [57] is another indexing structure designed for high dimensional point data. Its main goal is an efficient similarity indexing in order to facilitate efficient similarity queries on a dataset

of high dimensional feature vectors. The SS-tree is an improvement of the R*-tree, still sharing many common characteristics with it. In order to enhance the performance of neighborhood queries, the R*-tree is modified in two directions.

Firstly, the minimum bounding rectangles (MBR) used in the R*-tree for the region shape are transformed into minimum bounding spheres (MBS) (Figure 9). One advantage of this approach is that a data space region requires less storage space to be represented. For example, each dimension requires two values, minimum and maximum extension on each dimension, for rectangles boundary representation. If the feature vectors have D dimensions, $D \times 2$ values are required to describe a MBR. A sphere can be described by its center and its radius. Therefore, just $D + 1$ values are required to represent it, saving almost half of the storage space. By using less storage space in the non-leaf nodes, node fanout is increased and the tree height is decreased, improving the search efficiency. In the SS-tree, the center of the sphere is the centroid of the underlying points. The feature vectors are divided into neighborhoods by using centroids in the tree construction algorithm, the insertion and split algorithms. For the case of insertions, the most suitable subtree to accommodate a new entry is simply chosen based on the node whose centroid is closest to the new entry. When a node or leaf becomes full, the coordinate variance on each dimension from the centroids of its children is computed, and the dimension with the highest variance will be split.

Second, the concept of forced reinsertions introduced in the R*-tree is modified in order to enhance the dynamic organization of the tree structure. In the case of the R*-tree, when a node or leaf is full, the forced reinsert policy is used instead of the split strategy only if reinsertion has not been made at the same tree level. In the SS-tree, as improved over the R*-tree, the reinsertion policy is used instead of split policy only if reinsertions have not been made at the same node. This strategy allows more reinsertions and leads to a better utilization of the disk page storage space and more independence from the order of insertions.

In spite of these improvements, the SS-tree still suffers from a rapid degradation of its performance in high dimensionality. One of the reasons is that the volume occupied by an MBS is generally much larger than the volume of the corresponding MBR for the same undelying objects. The larger volume induces more overlap at the non-leaf level and consequently less prunning during search.

X-tree In order to overcome the overlap problem of the bounding rectangles in the R*-tree which increases with the growing dimensions, Berchtold et al. propose in [7] some improvements. The new structure is called X-tree (eXtended node tree) and it is optimized for high-dimensional vector spaces. In order to keep the directory as hierarchical as possible and avoid node splits that would result in high overlap, the concept of supernode is introduced along with a new split algorithm that tries to minimize and eliminate the MBRs overlap.

The X-tree starts from the observation that, for a large number of dimensions, the overlap introduced by the R*-tree like structures is very high and the whole structure has to be searched. Therefore, a linear organization of the directory is more efficient (takes less storage space and the pages can be fetched faster from the disk). Depending on the dimensionality of the feature space, the X-tree uses a hybrid approach trying to automatically organize the directory as hierarchically as possible without introducing overlap. If during insertions there is no other possibility to avoid overlap, then supernodes (Figure 10) are created. Supernodes are large tree nodes of variable size (multiples of a disk page size) used to avoid splits in the directory nodes that would result in an inefficient tree structure. In the case of the R*-tree, these splits would create a hierarchical structure with high overlap, and the split generated nodes would have a high probability of being all searched when one of them is searched. However, this would be inefficient compared with a linear scan of the supernode.

The number and size of supernodes increases with growing dimensions. Due to this increase, the

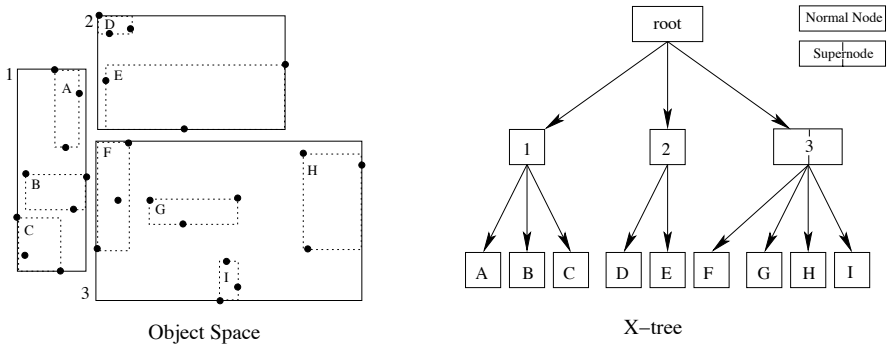


Figure 10: The X-tree structure

height of the X-tree corresponding to the minimum number of page access necessary for point queries is decreasing with increasing dimensionality of the feature space. The authors mention two possible special cases of X-tree. The first is when none of the nodes is a supernode. In this case X-tree is similar to R-tree. This may occur only for low dimensionality. The second case is that of the directory consisting of only one large supernode (root). This case occurs for highly dimensional data. The X-tree supports not only point data, but extended spatial data as well. Experimental performance shows that, on high-dimensional data, the X-tree outperforms the R*-tree and TV-tree by up to two orders of magnitude.

SR-tree The SR-tree [29] is an improvement over the SS-tree with the goal of enhancing the efficiency of nearest neighbor search in point queries. The authors start from the observation that, although a minimum bounding sphere (MBS were proposed in the SS-tree as bounding hyperobject) has a shorter diameter than the diagonal of the corresponding minimum bounding rectangle for the same underlying objects, its volume it is generally much larger. The higher the dimensionality of the feature space, the larger the ratio of MBS volume over MBR volume. A larger volume in MBSs or MBRs induces more overlap at the non-leaf level and therefore less pruning during search. A combination of both these bounding hyperobjects would give the advantage of both smaller volume and shorter diameter for bounding hyperobjects. The SR-tree (Spheres/Rectangles-tree) specifies a region of the vector space by the intersection of a bounding sphere and a bounding rectangle (Figure 11). As compared with the SS-tree, incorporating bounding rectangles allows neighborhoods to be partitioned into smaller regions, improving the disjointness among regions.

The general structure of the SR-tree is based on the R-tree, sharing many attributes with R*-tree and SS-tree, and it is represented by a nested hierarchy of bounding regions over the underlying objects. What makes the SR-tree structure distinctive is that a region is specified both by its minimum bounding sphere (MBS) and minimum bounding rectangle (MBR) of the underlying points (Figure 12). While a leaf node has a similar structure with the SS-tree, a non-leaf node describes each child with four components: a pointer to the child node, the number of points in the child node and the MBS and MBR of the child node. Since a child node entry of the SR-tree has a three times larger size than in the SS-tree and one-and-a-half than in the R-tree, the fanout of the SR-tree is one-third of the SS-tree and two-thirds of the R-tree. As mentioned before, a small fanout requires more nodes to be read on queries and may cause a reduction in query performance. Experiments have shown that although there is an increase of non-leaf nodes reads as expected, the overall total number of disk reads of the SR-tree is smaller than that of the SS-tree and R*-tree.

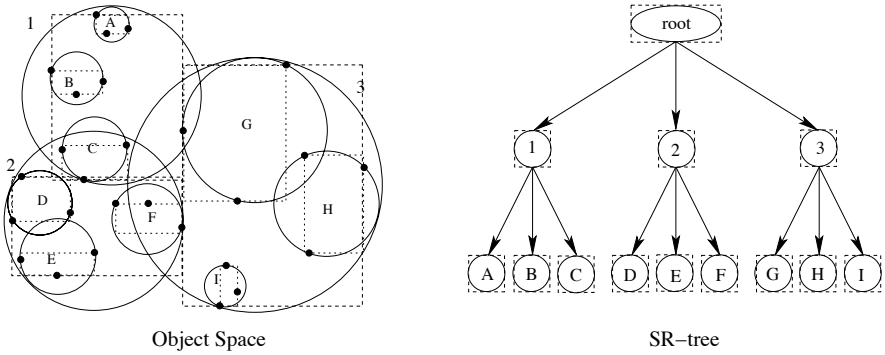


Figure 11: The SR-tree structure

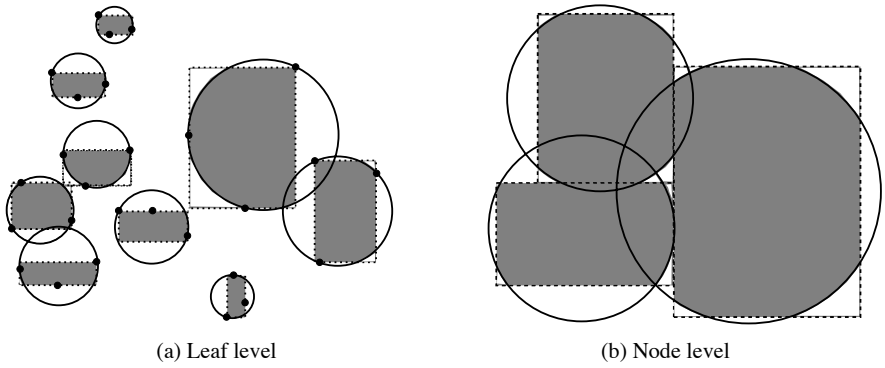


Figure 12: Bounding regions as specified by the intersection of a bounding sphere and a bounding rectangle

Another difference from the SS-tree and R-tree is for nearest neighbor queries. Because a bounding region in the SR-tree is represented by the intersection of a bounding rectangle and a bounding sphere, the minimum distance from the search point to a region is defined by the longer distance between the minimum distance to its bounding rectangle and the minimum distance to its bounding sphere. The performance evaluations have shown that nearest neighbor queries are specially effective for the SR-tree compared to other indexing structures for high-dimensional and non-uniform data sets which is the general case for real data in image retrieval.

Index Building by Bulkload Technique Typically, small high dimensional databases do not require an index structure since sequential scan search performs better than an index based search in this case. Once the database reaches a certain size, the use of an index structure is, however, worthwhile. In some fields, like Image Retrieval, it is usual to start with a large collection of data to which an index structure has to be built in order to improve the search efficiency. In either cases, it is undesirable to dynamically build an index since it has a high building time due to successive insertions and it will not take advantage of *a priori* knowledge of data.

The solution is to bulk-load the index, that is to build the index structure statically. Once built, as further data updates may be required in the future, the index structure has to allow dynamic operations to be performed. Bulk-load is not an index structure, but an operation that is applied to build a desired index structure. One can also take advantage of knowing a large amount of data items at building time.

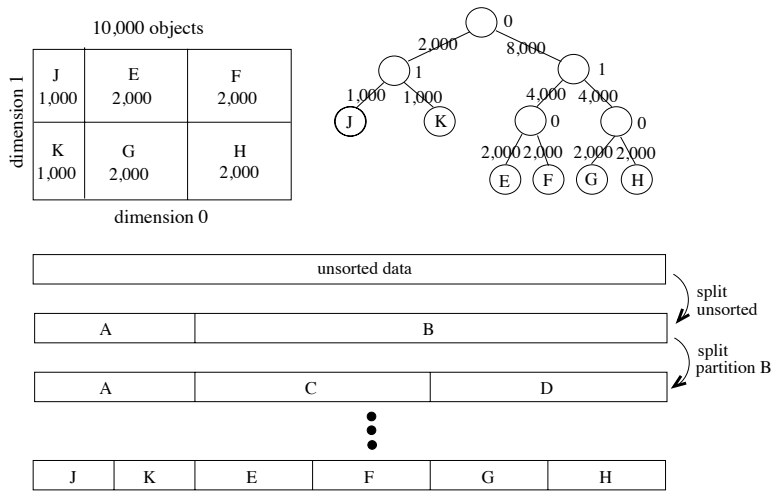


Figure 13: A 2-dimensional bulk-load partitioning (adapted from [6])

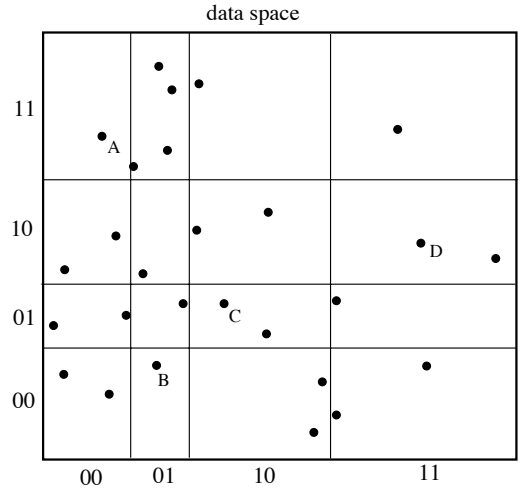
Several bulkload techniques are proposed in the literature. We will focus on those proposed in [6, 13], since it is the index building technique that we will also use in our experimental part. The proposed bulk-loading technique is applicable to R-tree-like structures. The basic idea of the technique is to split the dataset recursively in a binary split tree top-down fashion using hyperplanes as separators between partitions. A hyperplane is defined by a split dimension and a split value on the split dimension. The split value is chosen such that a certain ratio between the number of objects on the two sides of the split plane is achieved. Although the generated space partitioning is unbalanced, the resulting index structure is balanced, as the actual index is build in a bottom-up manner from the leaf nodes generated by the partitioning algorithm. An arbitrary storage utilization can be chosen. Beside the advantage of optimizing the shape of the bounding boxes, the proposed bulk-load algorithm creates an overlap-free directory. For a better understanding, Figure 13 shows a 2-dimensional bulkload partitioning in three

ways: as spatial distribution of the partitions, as the top-down split tree and the evolution in time of the partitioning process (all numerical values represent numbers of objects).

3.2.2 Space Partitioning Indexes

Another approach for indexing is based on grouping data objects based on their spatial positioning. The data space is divided along predefined or predetermined lines. Well known representatives of space partitioning indexes are quad-trees [20], k-d-b-trees [45] and grid-files [39]. Recent structures are using such vector space partitions to approximate the spatial locations of the data objects in order to accelerate the retrieval by creating a compact approximate feature search space. Such structures are the VA-file and the A-tree which will be presented in the following.

VA-file With the increase in dimensionality, the data partitioning methods tend to lose efficiency, requiring to search most of their nodes. In this case, a simple sequential scan can outperform most indexing methods since a sequential reading of all data disk pages is faster than a random read of more than 20% of data pages [56] (for many structures this happens already when the data space dimensionality is larger than 10).



vector data

A	0.2	0.8
B	0.3	0.3
C	0.4	0.4
D	0.8	0.6

approximation

A	00	11
B	01	00
C	10	01
D	11	10

Figure 14: A 2-dimensional VA-file data objects approximation example

The VA-file [11] (Vector Approximation file) is a simple flat array of vector approximations. The data space is divided into cells and each of these cells is represented by a unique bit-string of length b (user defined). Each data object is approximated by the bit-string of the cell in which it falls (Figure 14). This object approximation is equivalent to a quantization scheme. For nearest neighbor queries, all these approximations are scanned and for each approximation a lower and upper bound is computed for the distance between the object and query (Figure 15). These bounds are then used to eliminate some of the vectors from the search, generating a candidate set sorted by their lower distance. These candidates are then loaded from disk and the accurate distances between query and vectors are computed. If too many candidate vectors remain, the performance gain due to compact representation of approximation is lost. Since we already know the lower bound for each candidate, not all of them have to be visited. Once the accurate k^{th} neighbor distance is smaller than the lower bound of approximated distance, the search stops.

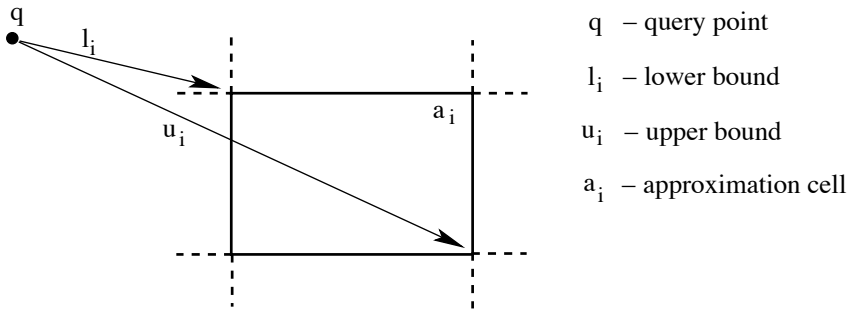


Figure 15: Lower and upper bounds for query processing

Based on the number of points and data distribution the user has to decide *a priori* how many bits (b_i) to allocate for a certain dimension (d_i). Then dimension d_i is split into 2^{b_i} regions ($2^{b_i}+1$ partition points are required). The partition points have to be chosen in such a way that the obtained regions are equally full. Typically, b_i is a small integer in the range 2-8. The length b of the approximation bit-string is the sum of the approximation lengths b_i over all dimensions. Each approximation cell is represented by an unique bit-string of length b. Each data point is approximated by the bit-string of the cell into which it falls. In Figure 14 this transformation is illustrated for four data points ($b_i = 2$ for each dimension).

The partition points are dividing the vector space in 2^b partitions. The total number of partitions is very large. For example, for a 64 dimensional feature space and 1 bit per dimension (that is each dimension is split in two partitions), we have 2^{64} partitions. If the total number of objects is 10^6 – less than 2^{20} –, then most partitions will be empty in our space. As the number of partitions is very large compared with the number of points, the probability that two or more objects will share the same cell, respectively approximation, is very low. Consequently, rough approximations can be used without the risk of collisions. The VA-file benefits from the spareness of a high-dimensional data space as opposed to data partitioning methods.

The main disadvantage of this approach is the requirement of *a priori* decisions regarding the number of bits per dimension (or total number of bits per approximation) and the partitioning points. If data distribution is modified in time, new partition points are required in order to keep equally full partitions. In order to precisely determine the partitioning points, the entire data set has to be analyzed, which is too costly for the case of insertions, deletions and updates. A solution is to determine these points stochastically by sampling. Another one is to keep these regions fixed during insertions, deletions and updates but this solution is safe only if we know that data distribution does not fluctuate. The authors show that for dimensionality larger than 10, the VA-file can outperform most other methods. Also, the performance of this method improves with dimensionality increase.

A-tree An approach that combines both data and space partitioning techniques is the A-tree [48]. The basic idea of this structure is to combine both the pruning advantages of tree-like indexes and the compact representation of position approximations as in the VA-file. The main contribution in this respect is the introduction of the Virtual Bounding Rectangle (VBR), which is an approximate representation of an MBR or data object (see Figure 16). A VBR is used to approximate children MBRs relative to their parent MBR (which is the MBR of all children MBRs). A child's VBR is represented by its two endpoints relative to the parent's MBR. The basic idea is to quantize the position of a child

MBR relative to its parent MBR, so that an approximate position of child MBR is known while saving node space. Figure 16 shows a 2-dimensional example of parent MBR (pMBR), child MBR (cMBR) and child VBR (cVBR). For the child VBR, 3 bits ($2^3 = 8$ regions) per dimension are used to quantize the child MBR relative to its parent MBR. While 16 bytes (2 points x 2 dimensions/point x 4 bytes/float = 16 bytes) would be required to represent the child 's MBR, less than 2 bytes (2 points x 2 dimensions x 3 bits/point = 12 bits < 2 bytes) are enough to represent the child's VBR in 2-dimensional case. Therefore, intermediate nodes have a large fanout, improving search efficiency.

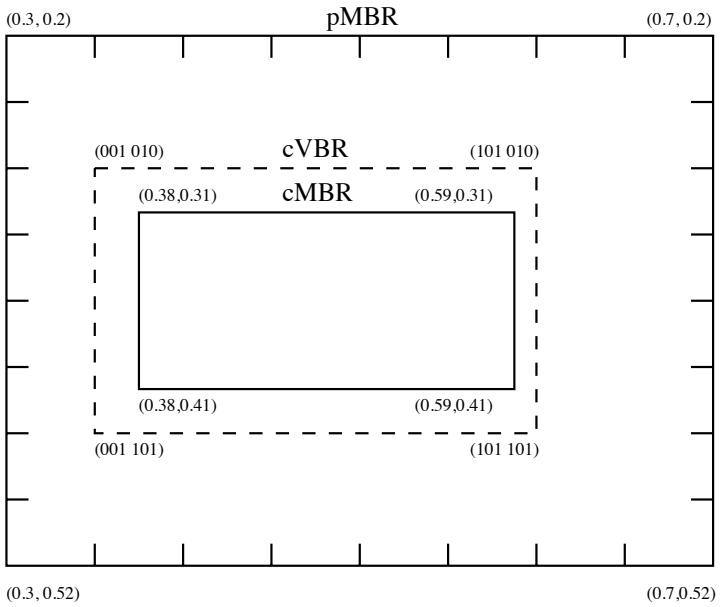


Figure 16: A 2-dimensional example of Virtual Bounding Rectangle

Compared to the VA-file, another advantage is that the quantization function dynamically adapts to the data distribution. In the A-tree representation, an index node contains the exact representation of the MBR of the node and the relative approximation of the MBRs of the node's children. This way, by reading just one A-tree node, we get partial information about the bounding regions of two levels. A drawback of approximating MBRs is that approximation error may lead to reduced pruning in searching if a query point falls outside the MBR but inside the approximation of the MBR.

The authors have used the SR-tree as a start point in the A-tree design. Since their experiments have shown that the effect of the Minimum Bounding Spheres is reduced for high dimensionality, MBSs are no longer stored in the A-tree nodes. In its structure, the A-tree is using four types of nodes. The data nodes contain objects feature vectors and pointers to the actual objects. Each leaf node corresponds to a data node and contains the MBR of the data node, a VBR approximating each feature vector and a pointer to the data node. An intermediate node contains the MBR of its children MBRs and for each child, its VBR, number of objects in child, child's objects centroid and a pointer to child. The root node is like an intermediate node, with the exception that it no longer contains the MBR bounding all children since this MBR would be close to the whole data space which is known in advance.

In order to apply the nearest neighbor algorithm on the A-tree, two nearest neighbor lists are used. One is the usual list found also in the other algorithms, containing candidate nearest neighbor points based on the real distance. The other list contains the candidate nearest neighbors based on the distance from the query point to the VBRs of data objects. With the help of the second list, data objects are

filtered and fewer data nodes are accessed. The reported results show that the A-tree is up to 75% more efficient in page accesses than the SR-tree and VA-file for 64 dimensions.

4 Indexing with Constraints

4.1 Motivation

Nowadays, large collections of information from many fields need to be indexed for faster retrieval, but sometimes, this information is too complex to be indexed directly. Often, features of the data are extracted and these features, usually represented as vectors, are used further on for indexing and retrieval. The generated feature space could have only a few dimensions or its dimensionality could be very large. Much research have been focused on indexing techniques for feature vectors (see Section 3), but they are very general with respect to indexed data. This generality is good since it allows an approach to be applied in many fields. On the other hand, a solution which is very particular to a certain problem can provide a great efficiency boost over other methods, while losing its relative advantage by being applicable only to a reduced set of problems.

This section will present our research with respect to image features indexing by taking into account *a priori* knowledge about data constraints in the feature space, in particular its distribution. We will show how simple yet powerful observations about the data distribution can help improve the efficiency of the retrieval process. Although our research is focused on the use of Global Color Histograms for Image Information Retrieval, the proposed solutions are applicable whenever the data can be represented as normalized histograms or the feature vectors have similar constraints. In the context of multimedia data, other applications include the representation of shapes of objects using different types of histograms. For instance, in [2] different methods for decomposing the enclosing sphere of an objects into a number of cells (e.g. concentric shells), corresponding to histogram bins, are proposed. The value of a bin in these histograms is the percentage of the object that falls into the corresponding cell. Experiments have shown this approach to work well, particularly when applied to 3-dimensional objects in the context of molecular biology. Another means to represent shapes using distance histograms was proposed in [19], and has been shown to provide effectiveness comparable to that obtained with more elaborate approaches such as using Fourier coefficients.

4.2 Feature Representation

To start, we will introduce the data preprocessing phase required for the indexing process. The first step is the feature extraction. As mentioned earlier, our data will consist of images and we decided to use Global Color Histograms as image features of choice. GCHs provide the type of constraint we want to study and take advantage of, while being easy to understand and extract. In addition, color histograms are extensively used in the image indexing research (see Sections 1 and 3).

As presented in Section 3, Global Color Histogram is the color distribution obtained by discretizing the color space used by an image and counting the number of pixels that fall into each bin. A color image using the RGB color model has a very large number of colors and therefore we perform a color reduction (discretization) to a lower dimensional color space before extracting the GCH. We chose 64 to be the number of colors to use in most experiments since this discretization captures reasonably the color distribution within our image without extending too much the dimensionality of the feature vectors. 64 also represents the maximum dimensionality used in many indexing techniques. It is also one of the most frequently used color spaces for color histograms in image information retrieval [15, 18, 36, 37, 40, 48]. In addition, it has been argued that a larger number of colors introduce noise in the sense that the notion of dominant color may be lost, while a smaller number might not capture

accurately the color variation in an image. It is important to note that our work does not depend on any particular dimensionality – in fact we aim at higher values. In order to reduce the RGB color space to 64 colors, we approximate each color by the representative of the RGB space partition where it falls into. That is, for reduction to 64 colors, the R, G and B color axis are divided into four partitions each. This will generate $4^3 = 64$ partitions in the RGB cube. Then, each color is approximated by the representative color of the partition where it falls into. The Global Color Histogram is computed by counting how many pixels belong to each color in the lower (64) dimensional space and normalizing the obtained values. Therefore, each GCHs bin represents the percentage of pixels of the corresponding color found in the target image.

4.3 Experimental Sets

The experimental evaluation consists of tests performed on real and synthetic data. Synthetic data consists of 300,000 (300k) data points with either 16, 32 or 64 dimensions. As real data sets, we use two image collections. The first data collection contains around 60,000 (60k) images from the Corel Gallery [16]. The second real data collection consists of around 110,000 (110k) TV snapshot images. The GCH is extracted for each image and used as feature vector. The generated data space has 16, 32 or 64 dimensions, depending on the level of quantization applied on the original images. Since the read disk operations (in the following noted as page accesses or IOs) are several times more costly than CPU and main memory operations such as computing the query-to-object and query-to-MBR distances, we will use the number of disk operations (page accesses) as a measure of performance in our experiments.

4.4 Feature Constraints

Directly from the color space reduction and GCH normalization, three important observations follow. Let us denote the D-dimensional GCH as:

$$H(I) = (c_1, c_2, ..., c_D), \tag{7}$$

where I represents an image in a D-dimensional color space and c_i represents the normalized number of pixels from image I corresponding to color i.

The first important observation is:

$$(\forall i) \quad 0 \leq c_i \leq 1 \ , \quad i = 1 : D \tag{8}$$

In other words, the data space is enclosed in a D-dimensional hypercube with each dimension taking values in the range 0 to 1. This is a trivial observation and all previous indexes using GCHs as feature vectors take advantage of it in a way or another.

The second observation is:

$$\sum_{i=1}^{D} c_i = 1 \tag{9}$$

The vector coordinates of the feature space are not independent, i.e., their sum is always 1. This gives us a different perspective of the data space. From the perspective of our space as a hypercube, now we know that we have a skewed data distribution in the hypercube. Actually, our data space is just a hyperplane intersecting the hypercube. Figure 17 shows these constraints for a 3-dimensional histogram feature space. To our knowledge, this observation has never been taken advantage of.

A third observation is that the maximum Euclidian distance between two points in this space is bounded and does not depend on the dimensionality of the space. Let p and q be two points in our

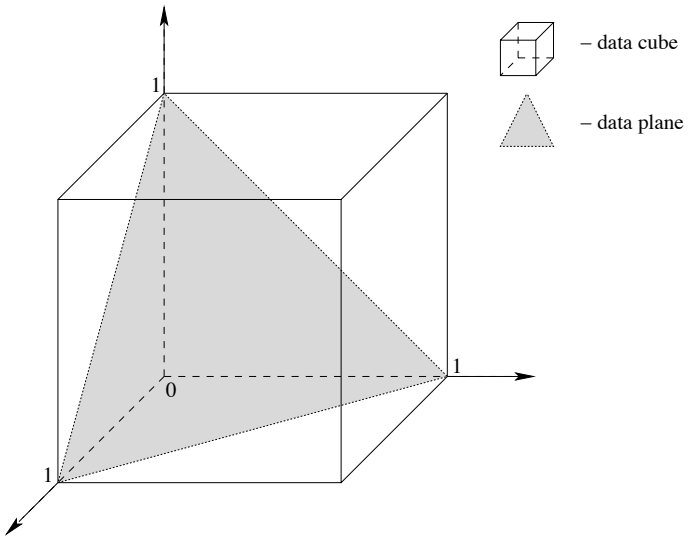


Figure 17: 3-dimensional GCHs feature space representation

constrained D-dimensional space, $p = (p_1, p_2, ..., p_D)$ and $q = (q_1, q_2, ..., q_D)$. Then,

$$d_{L_2}(p,q) = \sqrt{\sum_{i=1}^{D} (p_i - q_i)^2} \leq \sqrt{(\sum_{i=1}^{D} p_i)^2 + (\sum_{i=1}^{D} q_i)^2} \leq \sqrt{2} \tag{10}$$

While in a non-constrained D-dimensional unit-cube space the maximum distance between two points is $\sqrt{D}$, the histogram's constraint imposes a $\sqrt{2}$ maximum distance. Therefore, the distribution of neighbor distances in our constrained space is different than in a general space. This observation affects the performance of the nearest neighbor search algorithm, as we will see in the forthcoming sections. As the maximum possible distance between two points is decreased, more points are likely to be found in the same distance range from a query. As a result, the MBRs of many space regions have to be searched to determine the accurate neighbors during index search.

Since it will be used in the following sections, let us define here also the Euclidian distance between a point and a Minimum Bounding Rectangle (MBR) in a D-dimensional space. In general, when using a tree-based access structure, for processing a nearest neighbor query, the tree is traversed in a top-down manner. Sub-trees can be pruned if the objects contained in this sub-tree cannot be closer to the query than a current nearest neighbor candidate found earlier in the search. Hence, a tight lower bound for the distance between a query and the MBRs (enclosing the points and MBRs in lower levels of the tree) is of fundamental importance for the query efficiency. Currently, such a lower bound is obtained by computing the smallest possible distance between the query point and the boundary of the MBR. Formally, let $q = (q_1, ..., q_i, ..., q_D)$ be a (query) point and $B = (l_1 \rightarrow u_1, ..., l_i \rightarrow u_i, ..., l_D \rightarrow u_D)$ an MBR of a data space region, where l_i (u_i) represents the lower (upper) bound for dimension i and $l_i \rightarrow u_i$ is defined as MBR's dimensional extension (referred simply as extension in the following). Then,

$$d_{MBR}(q,B) = \sqrt{\sum_{i=1}^{D} max(l_i - q_i, 0, q_i - u_i)^2} \tag{11}$$

This distance is 0 if the query point q is inside the MBR; otherwise it is the minimum distance between q and any point on the face of the MBR, which is closest to q. Clearly, this lower bound measures

the minimum distance between the query point and the sides of the MBR, assuming that the bounded points can be anywhere inside the MBR.

4.5 Improved Distance

4.5.1 Theoretical Analysis

Based on our second observation (Equation 9), we will show that the lower bound distance used for pruning subtrees in indexing methods that use Minimum Bounding Rectangles as partitioning objects can be computed more realistically. We can compute a tighter lower bound distance for pruning than d_{MBR} defined in Equation 11. Therefore, we can provide improved pruning efficiency for several indexing methods (such as R*-tree, X-tree, SR-tree and others) when using normalized histograms as indexing objects. We will denote the new distance as d^C_{MBR} in the following.

Similar to the original d_{MBR}, our new minimum distance d^C_{MBR} is 0 if the query is inside the MBR. Since a query point q which does not satisfy the constraint of the histogram is not meaningful, each query point must be located on the data hyperplane. A tighter lower bound for the distance between a query point q and an MBR than d_{MBR} is the distance between q and the closest possible point on the intersection of the data hyperplane with the boundary of the MBR that is closest to q. This distance can be easily computed by first calculating the point on the face of the MBR that satisfies the original distance. If this point p does not satisfy the constraint in Equation 9, p is not on the data hyperplane, and a better minimum distance estimation can be found. The point can be computed by a simple procedure that (iteratively if necessary) adjusts the coordinates of p equally toward the data hyperplane, subject to the constraint that a coordinate cannot be adjusted beyond the limits of the MBR.

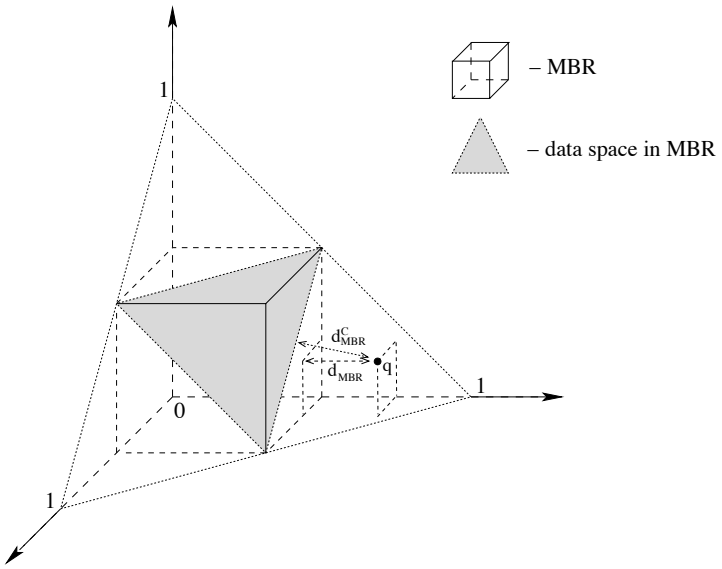


Figure 18: Original and improved query-to-MBR distances

To better understand our improved lower bound distance we will use again the 3-dimensional color space example, since it is easier to visualize. In Figure 18, q represents the query point (contained in the constrained data space), d_{MBR} represents the original lower bound distance to the MBR of a data space region as computed in MBR-based indexing methods, while d^C_{MBR} represents the new lower bound distance to a possible data point within the MBR. While usual indexing methods assume that

data points could exist anywhere inside an MBR, by acknowledging the constraints of our data space we compute a better approximation of the lower bound distance to a possible data point inside the MBR. In other words, d_{MBR} represents the shortest distance between a query and an MBR, while d^C_{MBR} is the shortest distance between a query and a possible constrained data space point included in the MBR. The relation

$$(\forall\, q)\ \ (\forall\, B) \quad d_{MBR}(q, B) \leq d^C_{MBR}(q, B) \tag{12}$$

where B represents an MBR and q a query point, always holds.

Proof: Let assume that equation 12 is false. Then:

$$(\exists\, q) \quad d^C_{MBR}(q, B) < d_{MBR}(q, B) \tag{13}$$

Also, by definition of d^C_{MBR},

$$(\exists p)\ \ p \in B\ \ d_{L_2}(q, p) = d^C_{MBR}(q, B) \tag{14}$$

From here, it follows that

$$(\exists p)\ \ p \in B\ \ d_{L_2}(q, p) < d_{MBR}(q, B) \tag{15}$$

But, by definition, d_{MBR} is the smallest distance from q to any point on the MBR B. Therefore, the above equation is false and our assumption is also false. So, equation 12 must be true. •

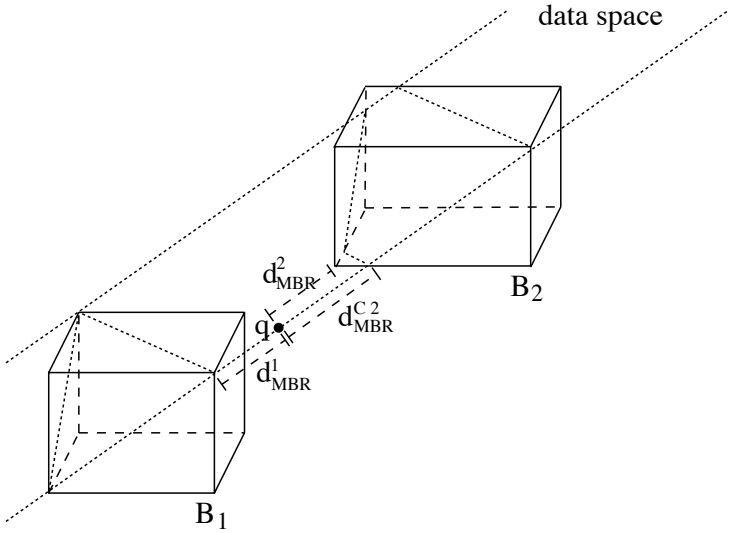


Figure 19: Improved MBRs search order by using the improved distance

An important effect of the improved query-to-MBR distance is a changed order in which the MBRs are considered in a nearest neighbor search. Figure 19 shows a 3-dimensional vector space example where the search order is modified. B_1 and B_2 represent two bounding rectangles of regions from the data space, q is the query object, d^1_{MBR} is the original query-to-MBR distance for B_1 (equal with the improved d^{C1}_{MBR} for MBR_1), d^2_{MBR} is the original query-to-MBR distance for B_2 and d^{C2}_{MBR} represents the improved query-to-MBR distance for B_2. The three distances satisfy the following equation (consistent with Equation 12):

$$d^1_{MBR} = d^2_{MBR} < d^{C2}_{MBR} \tag{16}$$

Let us assume that B_1 is searched first (note that this assumption always holds when using the improved distance in our example). Since d^1_{MBR} is the minimum query-to-MBR distance, the k^{th}

neighbor distance after searching B_1 cannot be smaller then d^1_{MBR}. When using the original distance function, d^2_{MBR} is equal with d^1_{MBR} and, therefore, B_2 has to be searched as well. By using the improved distance, B_2 does not have to be searched if the k^{th} neighbor (maybe found while searching B_1) distance is smaller then d^2_{MBR}. Thus, by using the improved query-to-MBR distance derived based on the constrains of the data, we can improve pruning during the index search.

As this example suggests, an even more important effect of the improved distance is that some MBRs (sub-trees) may not be considered at all during search, where they had to be considered with the original distance. Although we do not expect this effect to be extremely large, there is a potential for consistently saving disk page accesses. In particular, with increasing dimension, the difference between the usual distance and our new distance ($d^C_{MBR} - d_{MBR}$) will become smaller on average. An intuition for this is the way the new distance is computed – using the closest possible point $p = (p_1, ..., p_i, ..., p_D)$ to a query point $q = (q_1, ..., q_i, ..., q_D)$, where p is located on the surface of the MBR and has the original distance to q. If p is not on the data hyperplane, then $\sum_{i=1}^{D} q_i - \sum_{i=1}^{D} p_i = \epsilon > 0$. A new point p^c on the intersection of the MBR and the data hyperplane is then computed by adjusting the coordinates of p equally toward the data hyperplane without leaving the MBR boundary. As dimensionality D increases and ϵ is tentatively equally divided among all dimensions, the distance between p and p^c decreases. Consequently the the difference between d^C_{MBR} and d_{MBR} decreases.

4.5.2 Experimental Evaluation

To test the effect of our improved distance on the number of page accesses, we performed tests with an X-tree like index structure built using the bulkload technique explained in Section 3.2.1. Since the improved distance can be used in combination with any indexing structure using MBRs as bounding regions, we used a straightforward split policy for index building (the data partition is split into two equal subsets at each step during index bulkload). For all experiments the average result over 1000 100-nearest neighbor search queries is computed. The calculated measure is the number of page accesses required to find the nearest neighbors when using the original and when using improved query-to-MBR distances during index search. Also, we use 64 dimensional feature vectors unless otherwise noted. The figures display the percentage of page accesses saved when using the improved distance. We also varied the page size - that influence the MBRs sizes and number of pages - and the number of objects (points) to see how these affect the percentage of saved page accesses. We used 4096 (4kb), 8192 (8kb) and 16384 (16kb) bytes as possible disk page sizes.

In the first set of tests we performed on all three datasets, we obtained around 5% improvement in the number of page accesses required (Figures 20, 21 and 22). As the total number of pages increases (with decrease in page size for the same dataset size), the distance improvements become more significant for all datasets.

To test the effect of the number of nearest neighbors on the distance improvement, we used the Corel data set with a fixed number of objects (60k), while varying the number of nearest neighbors searched for. The percentage of saved page accesses slightly decreases with the number of nearest neighbor objects searched for (Figure 23). As observed in the previous set of tests, smaller page sizes leads to higher savings.

Figure 24 shows the effect of feature vector's dimensionality on the distance improvements. As the dimensionality increases, the gain obtained using the improved distance decreases. This result is consistent with our conclusion from the previous section.

Overall, the improvement in the number of saved page accesses by using the proposed distance it is not very high, but it is increasing with the number of objects in the data set and number of data pages that have to be searched. Unfortunately, as data dimensionality increases, the advantage of the improved distance over the original one decreases.

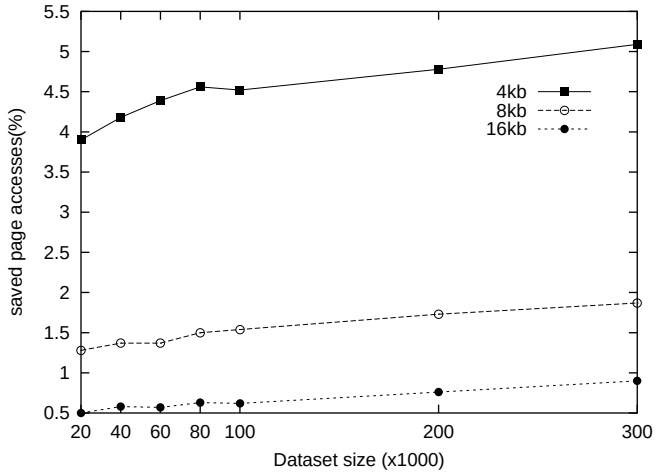


Figure 20: Distance improvements with dataset size - uniform dataset

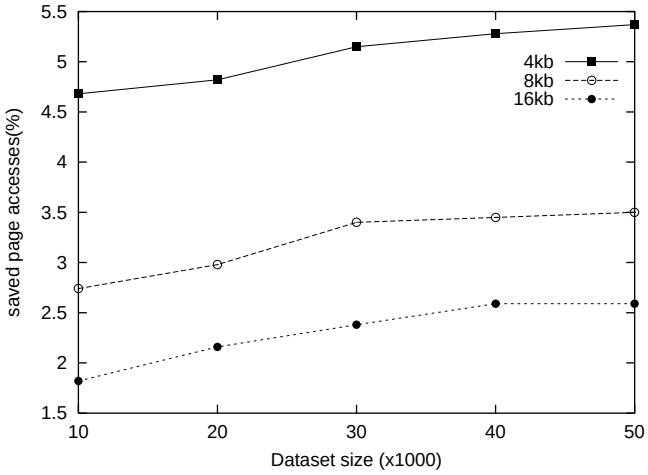


Figure 21: Distance improvements with dataset size - Corel dataset

4.6 Index Optimization

4.6.1 Theoretical Analysis

Before introducing our other observations and improvements regarding data indexing, let us present the basic indexing method that we will as a starting point. As discussed in Section 3.2.1, it is inefficient to build an index by dynamic insertions of a large number of data objects when all data is available *a priori*. As this is a common case for Image Retrieval (creating an index for an already existing collection of images), we assume that we have the data collection beforehand. We start from this assumption and create our index by using bulkload as described in [6]. Bulkload is not an indexing structure by itself, but a technique for efficiently building an index when an initial set of data exists. Once an index is built by bulkload, all algorithms used by dynamic index structures such as insertions, deletions, range or nearest neighbor searches and others can be directly applied on the resulting structure. The advantage of the bulkload technique over dynamical index building is that one can easily take advantage of information about the distribution of the data at building time. As shown in [6], the bulkload technique is efficient

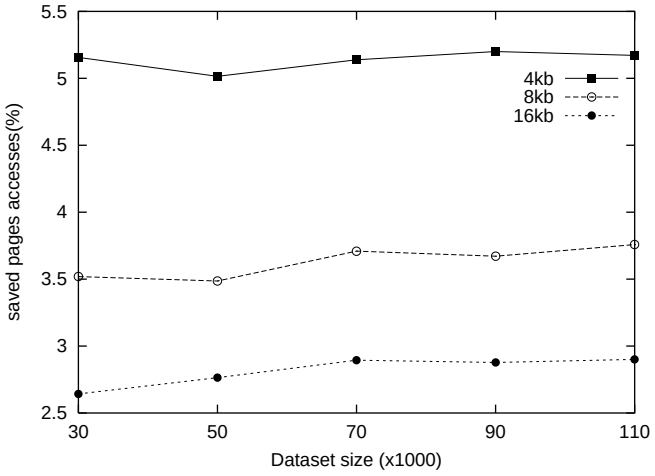


Figure 22: Distance improvements with dataset size - TV dataset

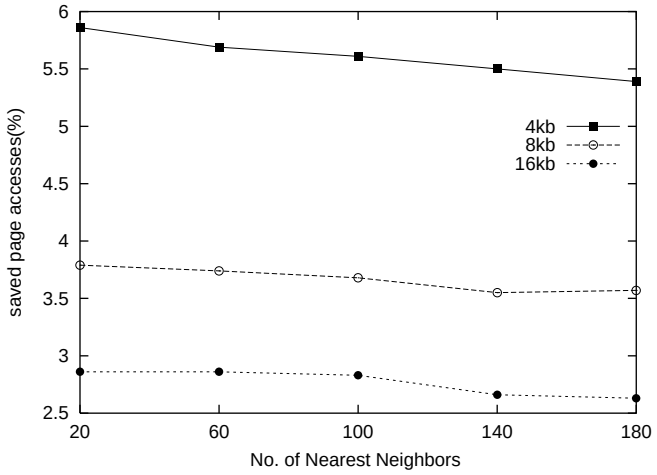


Figure 23: Distance improvements with no. of neighbors - Corel dataset

for both building and searching the index. Basically, the idea is to split the data space recursively in a top-down fashion, using hyperplanes as separators between partitions. A hyperplane is defined by a split dimension (the normal vector on the hyperplane) and a split value along that dimension (the actual position of the hyperplane). The basic indexing structure we choose is the X-tree [7]. Combined with bulkload construction, the X-tree index has the advantage of overlap-free partitions. The bulkload technique allows also to target a desired storage utilization. Due to the large dimensionality of our feature vectors, the X-tree is reduced to a list of leaf nodes and a large supernode (for a description of the high dimensionality effects on the X-tree, see Section3.2.1 and [7]). Since we are only interested in query performance and not in dynamic insertions, we use a near 100% node storage utilization.

Several aspects of an index structure such as fanout, tree height, split policy, node size, node overlap and node characteristics are important for query performance. Many of these aspects are dependent on the index type. One of the most important factors influencing the query performance is the split policy (determined by the split dimensions and the split values), which is independent of the target tree structure. We will show that the range of the feature values, feature vector's constraints and the object

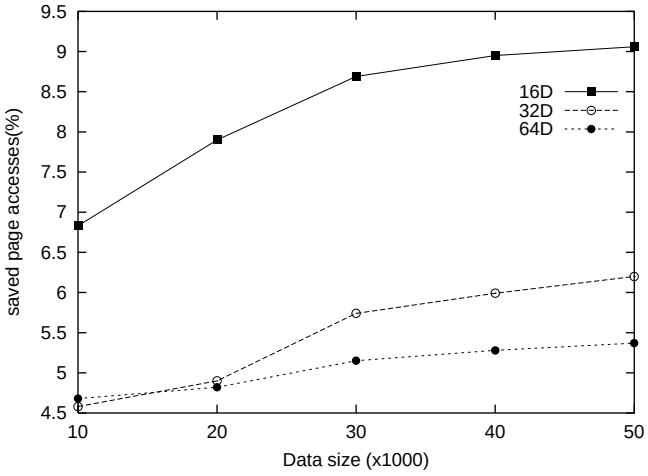


Figure 24: Distance improvements with dimensionality - Corel dataset

value's distribution allow the design of a new split policy for our constrained data space which improves significantly the query performance.

An important observation regarding high-dimensional indexing [12] is that for uniformly distributed features in a high dimensional unit hypercube we cannot have a split in all dimensions of the data space. For example, in an uniform 32-dimensional data space, if there is only one split on each dimension, we would require $2^{32} = 4,294,967,296$ pages to hold each partition in a separate page. Therefore, the D-dimensional data is usually split in $D' < D$ dimensions. The MBRs of the resulting partitions are restricted in D' dimensions while in $D - D'$ dimensions the MBRs are covering the whole range of possible values in those dimensions. The number of split dimensions D' can be determined from the number N of objects in the dataset and the average number of data points per index page C_{eff}:

$$D' = log_2 \left(\frac{N}{C_{\mathit{eff}}} \right) \tag{17}$$

For instance, for $N = 1,000,000$ feature vectors and $C_{\mathit{eff}} = 10$ feature vectors per page, we need to perform at least 17 splits to be able to fill index pages, and no more than 19 splits to have at least one object per page/partition, since $2^{20} > 10^6$.

Another observation is about the extension of the nearest neighbor distance in an uniformly distributed high-dimensional unit hypercube data space. If we want to find the k-nearest neighbors in an N point dataset in D-dimensional space, the side length l of the hypercube containing the neighbors is:

$$l = \sqrt[D]{\frac{k}{N}} \tag{18}$$

Proof: We assume an uniform D-dimensional unit hypercube space. Let k be the number of neighbors, N be the total number of points in the dataset, $V_k = l^D$ be the volume occupied by the k neighbor hypercube of side length l of and $V_D = 1^D$ be the volume of the unit hypercube space. Then:

$$V_k = \frac{k}{N} \times V_D \;\; \Rightarrow \;\; l^D = \frac{k}{N} \times 1^D \;\; \Rightarrow \;\; l = \sqrt[D]{\frac{k}{N}} \tag{19}$$

•

For a 32-dimensional uniform space with $N = 1,000,000$ points and $k = 20$-nearest neighbors required, the nearest neighbor hypercube side length is $l = 0.71$, which is larger than half of the data space extension for each dimension. To improve the query performance, it is essential to have as few data MBRs as possible intersecting the k-nearest neighbor hypercube. Therefore, it is important to have the MBRs extents restricted in as many dimensions as possible.

Let us first introduce an observation regarding the uniformity of the constrained data space. In an unconstrained multi-dimensional space, data is uniformly distributed in the whole space when data is uniformly distributed along each space dimension. Thus, one can generate uniform distributed data in an unconstrained multi-dimensional space by generating uniform distributed values along each dimension. In order to obtain an uniform distribution of data in the constrained space a naive approach would be to generate uniformly distributed points in the unconstrained space and select those that fulfill the constraint. However, the distribution function of the number of objects in the constrained space N_{CS} based on number of objects in the unconstrained space N_{US} is:

$$P(N_{US}) = \frac{Vol_{CS}^{D}}{Vol_{US}^{D}} \times N_{US} \tag{20}$$

where Vol_{CS}^{D} (Vol_{US}^{D}) represents the volume of the constrained (unconstrained) D-dimensional space. Since the constrained space is a hyperplane in a hypercube, $Vol_{CS}^{D} = 0$ in the D dimensional space (the constrained space extends freely in only $D-1$ dimensions) and therefore, $P(N_{US}) = 0$. Therefore, it is not feasible to generate uniform distributed points in the constrained space by this approach.

Input : Dimensionality D, Constraint Value $c = 1$, Point $p = (0, 0, ..., 0)$
Output: Data Space Point p

while $c \neq 0$ **do**
    $value$ = `Random()` * c;
    Choose dimension dim such that $p[dim]$=0;
    $p[dim] = value$;
    $c = c$ - $value$;

Figure 25: Uniform data generation in constrained space

To obtain uniformly distributed data in the constrained space, we use the algorithm presented in Figure 25. Note that function `Random()` generates a random real value in the range 0 to 1. At each step the algorithm generates a random number between 0 and the maximum value allowed such that the constraint holds, taking into account the already assigned dimensional values. Note that although this procedure will generate uniform distributed data points in our constrained space, the marginal value distribution on each dimension taken individually is not uniform.

Since the sum of a feature vector values over all dimensions is constant (Equation 9), the values on most dimensions have to be low. When this constant is 1 as in the GCHs case, the average value of a dimension in a feature vector is $1/D$ in a D-dimensional vector space (Figure 26). For instance, for data distributed uniformly in the constrained space and 64 dimensions, the average dimensional value is 0.016. Therefore, the marginal distribution of data point values is more dense in the lower value range and this density is increasing with the feature space dimensionality increase (Figure 27 (a) and (b)). As we shall see, this implies properties that can be further explored.

To understand the effect of constraints on the new MBRs created once a split is performed, we reduce again the data space to the 3-dimensional case (Figure 28). Let our initial MBR be $B = (l_x \rightarrow$

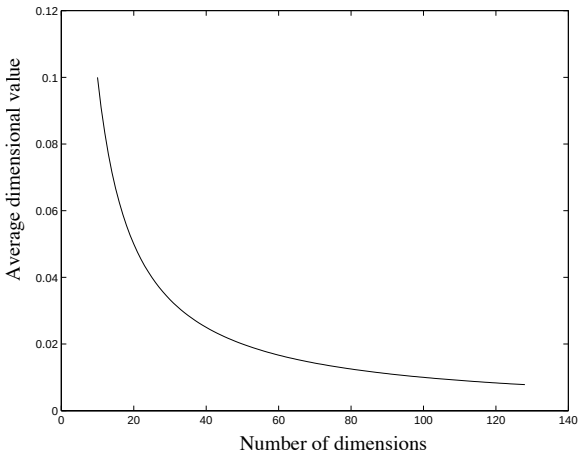
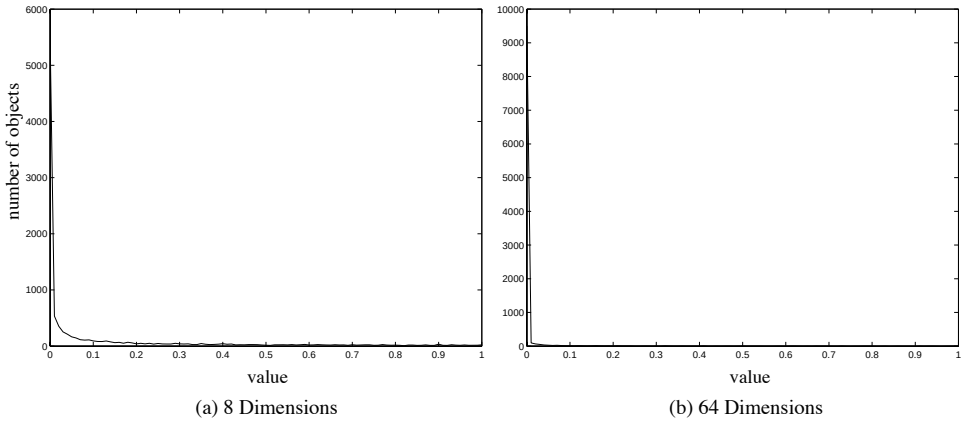


Figure 26: Average value of a dimension in a feature vector with increasing dimensionality



(a) 8 Dimensions (b) 64 Dimensions

Figure 27: Average marginal value distribution of 10,000 objects in an 8(a) and 64(b) dimensional uniformly distributed constrained space

$u_x, l_y \rightarrow u_y, l_z \rightarrow u_z$), where l (u) stands for the lower (upper) bounds. As our example starts with the whole space, $l_x = l_y = l_z = 0$ and $u_x = u_y = u_z = 1$, but the effects presented in the following are not related with these particular values.

Let us first choose to split dimension z at split value s_z, where $l_z \leq s_z \leq u_z$. The dataset is split into two partitions (Figure 28 (a)). One partition, with B_2 as bounding region, contains all data points with dimensional value on the z axis between l_z and s_z. The other one, with B_1 as bounding region, contains all data points with dimensional value on the z axis between s_z and u_z. Although only one dimension is explicitly reduced by the split, due to the data constraints, all dimensional bounds are restricted for $B_1 = (l_x \rightarrow u'_x, l_y \rightarrow u'_y, s_z \rightarrow u_z)$, where u'_x and u'_y represent the induced upper bounds. On the other hand, for $B_2 = (l_x \rightarrow u_x, l_y \rightarrow u_y, l_z \rightarrow s_z)$, only the bound on the dimension that is split is affected.

Note that although this split generates overlap-free MBRs, the projections on some dimensions of the two generated MBRs do overlap (further referred as dimensional overlap). The dimensional overlap of the two generated MBRs is equal to the B_1 extensions on dimensions x and y. Since the distance between the query point and the MBRs resulting from the split is a sum of distances in each dimension,

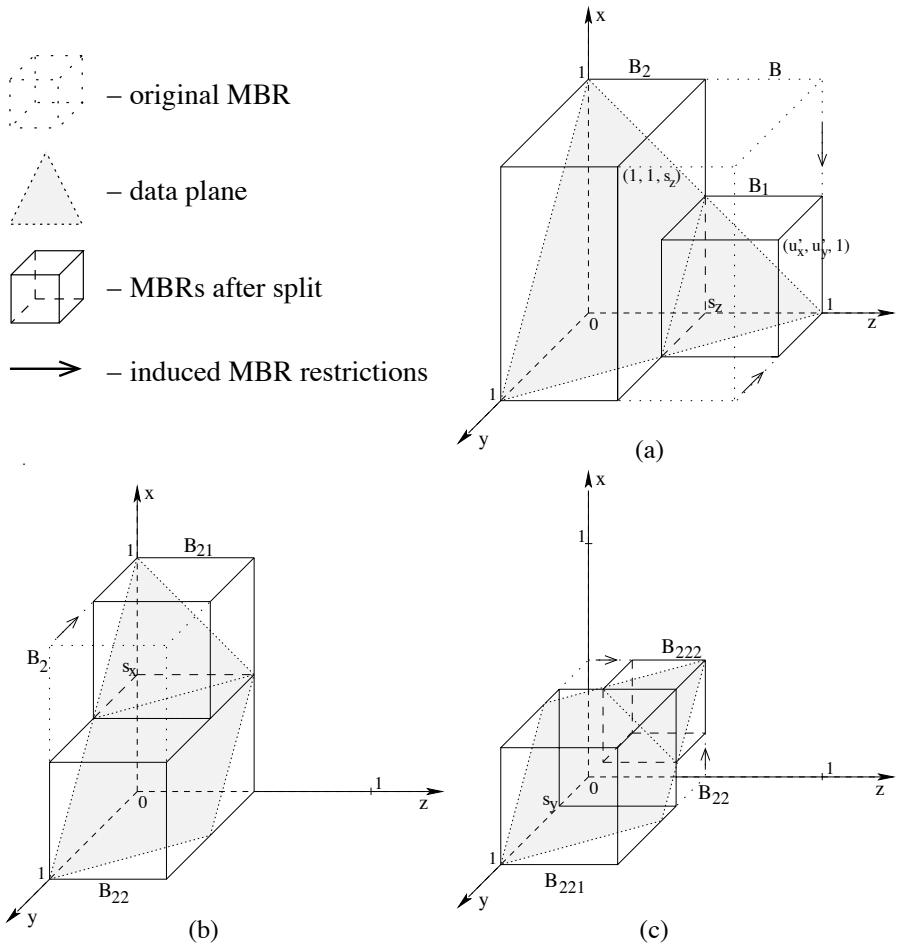


Figure 28: Illustration of the effect yielded by induced splits (for the sake of clarity only the MBRs being split are shown)

the more the MBRs differ in each dimension, the larger the difference in the distance between them and the query point. This yields a higher degree of dissimilarity between them with respect to the query and consequently a smaller probability that both would be searched during the nearest neighbors query process. Hence, the importance in reducing the dimensional overlap when splitting an MBR.

If s_z is close to u_z, then B_1 will be very restricted on all dimensions. Though only one dimension is chosen to be split not only that facet of the MBR is affected, but indirectly others are too, hence inducing an effect similar to multiple simultaneous splits. If s_z is close to l_z, the extent of the dimensional overlap is higher as the extensions of B_1 are larger. In this case, the nearest neighbor distance from a query point inside B_2 will have a higher probability to intersect B_1 as the Euclidean distance from the query to B_1 will mostly depend on their distance on dimension z (see Equation 11). Therefore, it is better to choose a split value closer to the upper bound of the chosen split dimension in order to affect the extensions of all dimensions as much as possible.

This split induces MBR restrictions for B_1, but B_2 still extends on most dimensions as much as the initial MBR B and dimensionally overlaps with B_1 in all dimensions but z. In order to reduce the dimensional overlap further on, the split policy should select other dimensions for splitting B_2 than the one already split.

If B_2 is now split on dimension x, two new partitions are generated, with B_{21} and B_{22} as their bounding rectangles (Figure 28 (b)). Again, the MBR bounding the partition created in the higher value range from the split value (B_{21}) is further restricted in other dimensions beside the split dimension due to data constraints. Note that if s_x would be closer to u_x, the induced MBR restrictions would be present on both y and z dimensions, and not just y as we have in the presented example.

Finally, consider B_{22} is further split on dimension y (Figure 28 (c)). The new bounding regions are B_{221} and B_{222} for the higher, respectively lower, value range on the split dimension. In this split example, the data constraints induce additional MBR restrictions to the MBR bounding the data points from the lower value range. If the split value is chosen closer to the u_y, then the induced MBR restrictions will act on the MBR bounding the other partition.

It is hard to predict which MBRs will suffer the induced restrictions. In fact, the same sequence of splits shown in Figure 28, could yield quite distinct MBRs, with the induced constraints acting in different ways (and possibly MBRs), depending on the split values and dimensions chosen at each split. Nevertheless, it is important to have the split value closer to the upper value bounds on the split dimension in order to induce stronger MBR restrictions.

Based on the observation regarding the increased density values in the lower range of each dimension, we can draw another strategy for the split policy. If the split policy would choose the split value close to the lower bound of the split dimension, then the distance from many points on both sides of the split to the split value on the split dimension will be very small. Recall that lower value range has a higher density of objects (Figure 27). Therefore, it is better to choose the split value towards the higher bound on the split dimension of an MBR. This observation is consistent with the previous one regarding the MBRs split value in order to increase the induced constraints in non-split dimensions.

Even though we have concluded that the split value should be close to the upper bound of the split dimension, an interesting question is how close that should be. If the newly created MBRs are too "thin", i.e., the split value is too close to the upper bound, the radius of the nearest neighborhood (as suggested by Equation 18) would cause the query process to search through possibly many neighboring MBRs, hence decreasing query performance. Therefore we have two competing splitting strategies. On the one hand, we want the split value to be close to the split dimension upper bound, on the other hand we do not want it to be "very" close, i.e., we do not want to create thin MBRs as mentioned above. How to achieve a good balance is the question we try to answer in what follows.

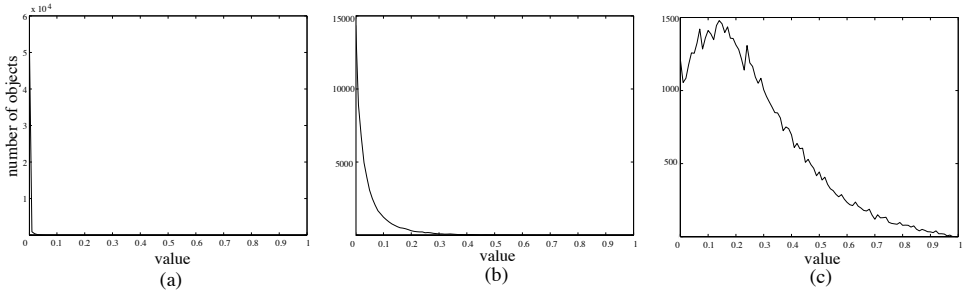


Figure 29: Average marginal value distribution of Corel dataset objects in 64 dimensions

For the real data sets, obtained from collections of images, the marginal distribution is different than the uniform one. Most colors have a dimensional distribution close to the one of the uniform data in the constrained space (Figure 29(a)), a few colors are in very small quantities or almost do not show at all, a few appear more often (Figure 29(b)) and one color tends to have a distribution closer to the uniform dimensional distribution (likely background color) (Figure 29(c)). Although the feature vector values constraint is valid, this variation in the marginal (dimensional) value distribution makes the split

policy for uniform data inappropriate, as we will show later on in the experimental evaluation.

4.6.2 Unbalanced Split Policy

In order to improve the retrieval process, based on our observations with respect to data partitioning, we define a recursive algorithm. The algorithm consists of a recursive procedure that takes a data set and, based on the characteristics of its MBR, splits it into two smaller partitions with overlap-free MBRs. The procedure is further applied on each of the generated MBRs until each partition fits into an index page. In the splitting process, three procedures are used to determine the split policy. The first one, `choose_split_dimension()`, selects the split dimension d_s. Assuming the MBR extension on dimension d_s is l, `find_split_ratio()` finds the split ratio s_r for the length of the resulting extents. That is, one extent will cover the first $s_r \times l$ of the original extent, and the other will cover $(1 - s_r) \times l$. This ratio could be a constant number or adaptive based on the current step of the algorithm. Finally, `find_split_position()` computes the actual dimensional split value as close as possible to satisfy the split ratio, and such that each partition results in a number of nearly full pages. A pseudo-code version of the algorithm is presented in Figure 30.

Algorithm: `create_partitions`(B_S)

Input : A set of feature vectors S, the MBR of S B_S, the capacity of an index page *psize*

Output: An X-tree supernode with links to all leaf (data) index pages

if *no_of_objects*(B_S) < *psize* **then**
    `create_page` (B_S);
    `STOP` ;
d_s = `choose_split_dimension` (B_S);
s_r = `find_split_ratio` (d_s, B_S) ;
s_p = `find_split_position` (s_r, d_s, B_S);
(B_1, B_2) = `split_MBR` (B_S, s_p, d_s);
`create_partitions` (B_1);
`create_partitions` (B_2);

Figure 30: Bulkload index creation using Top-Down dataset partitioning

Beside the three split related procedures presented above, there are a few others like `create_page()` that takes the data set partition and fills a newly created page, computes the page's MBR and adds it to the directory super-page, and `split_MBR()` that separates the data set partition according to the split dimension and split value into two subpartitions. Therefore, that data set is partitioned in a top-down tree like manner.

The two procedures that most affect the efficiency of the index structure are `choose_split_dimension()` and `find_split_ratio()`. For the first one, our previous discussions suggest that choosing the split dimension based on the maximal extension is a straightforward choice allowing more freedom in selecting the split value and allowing to impose a larger constraint in the other dimensions for the MBR bounding the data points from the higher value range. Also the data density variation could be larger, thus allowing a better split (for the same number of objects, a larger extension implies a lower density). For `find_split_ratio()` there are several possibilities since, as argued above, this ratio is highly dependent on the data and it should optimize competing criteria. One option is to find a constant ratio (e.g. that is closer to the higher bound as mentioned before) that performs best on average. The other possibility is that of an adaptive ratio based on the current step of the recursive partitioning.

According to our analysis, an efficient split should choose different split dimensions as often as possible and a split value closer to the upper bound of the MBR on the split dimension. The induced MBR restrictions are then be maximized. Due to the marginal distribution (see Figure 27), the resulting MBRs are be large enough so that they are not affected by the negative effect of large nearest neighbor distance during retrieval process. For uniformly distributed data in our constrained space, the split policy is straightforward to implement. As split dimension we choose the one with the maximal extension. As data has the same distribution in all dimensions, this strategy for selecting the split dimension will therefore effectively alternate among all dimensions. As split value, the closest possible value to the upper bound such that near 100% node utilization is obtained would be to make the split at one page distance from the upper bound. In other words, the algorithm selects all dimensions for splitting in round-robin fashion and, for each split phase, takes enough objects that have the marginal value on the split dimension close to the upper bound in order to fill a new page. In the following we will refer to this strategy as *last page* (LP) split.

Experimental evaluation shows that this split policy is efficient for uniform data in the constraint space. However, it may not be optimal for real data due to differences in the marginal distributions (Figure 29). In what follows we will show experimentally that, due to the complex and contradictory conditions that have to be fulfilled for real data, a constant split ratio of 80% gives good results on average. In the following we will refer to this strategy as *ratio* split.

4.6.3 Experimental Evaluation

To compare the proposed index optimizations with other indexing structures, we decided to use state-of-the-art indexing structures: the SR-tree [29], one of the best data partitioning index techniques and the A-tree [48], to our knowledge the best of space partitioning indexing techniques. We used the original implementations of the SR-tree and A-tree as provided by their authors.

To show the performance of our proposed indexing technique, we have performed many tests on both synthetic (uniform) and real data sets. Our tests have mainly focused on the real sets since we consider the applicability of our solutions very important. To denote our technique, we are using the notation *X* (since the created index is an extreme version of the X-tree index structure for high dimensional data) followed by the split ratio or by *LP* for Last Page split (e.g., *X-80%* or *X-LP*). To show the independence of the proposed index structure from the new distance, our structure will use the original distance function. For the other two indexing structures we compare to, we use their own names (*A-tree* and *SR-tree*). Unfortunately, the code provided by the authors of the A-tree index structure did not work with datasets larger than 100,000 data points. Therefore, in some of the graphs presenting results for various data sizes, the A-tree curve line stops prematurely.

As the code for the three indexing structures comes from different sources using different implementation choices, we will use only comparisons based on number of disk accesses during search. Unless otherwise specified, we have use the average over 1000 100-nearest neighbor search queries. Also, we use 64 dimensional feature vectors with the exception of the graphs where feature vectors size is varied.

In all tests, the query objects are randomly taken from the test datasets, in order to query the sets with something representative to their content.

To investigate our intuition about the superiority of Last Page Split for uniform data, we have performed several tests for three dataset sizes: 20000 (20k), 100000 (100k) and 300000 (300k), using both LP Split and different ratio splits. Figure 31 shows the ratio between number of page accesses required when using different split ratios over number of page accesses required when using the Last Page Split policy. As the graph shows, a 95% split ratio returns results very close to the LP Split. As explained in Section4.6.1, when using a split ratio we choose the actual split value as accurate as possible to acquire the split ratio, with the constraint of obtaining a near 100% page utilization. For instance,

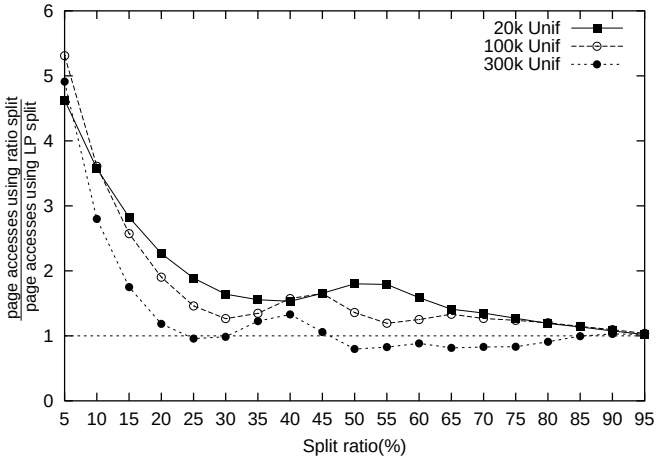


Figure 31: Number of page accesses using ratio split over LP split - uniform dataset

when using a 95% split ratio, we may not obtain enough points to (nearly) fill a page. At this moment, we need to go below the (95%) split point in order to fill the page, i.e. the 95% split ratio behaves as the LP split. Indeed, such behaviour can be seen in Figure 31. The LP Split policy proved to be the best for 20k and 100k set, but there were better ratio splits than LP split for the 300k size uniform dataset. As the dataset size increases, the objects' density in the dimensional high end increases. Thus, the MBRs created by the LP Split are too narrow and a query falling inside such an MBR will intersect all other narrow MBRs in the vicinity of its MBR. Nevertheless, since for most uniform dataset sizes we tested, the LP split policy performs better than ratio split policy, all test involving uniform sets will use this policy.

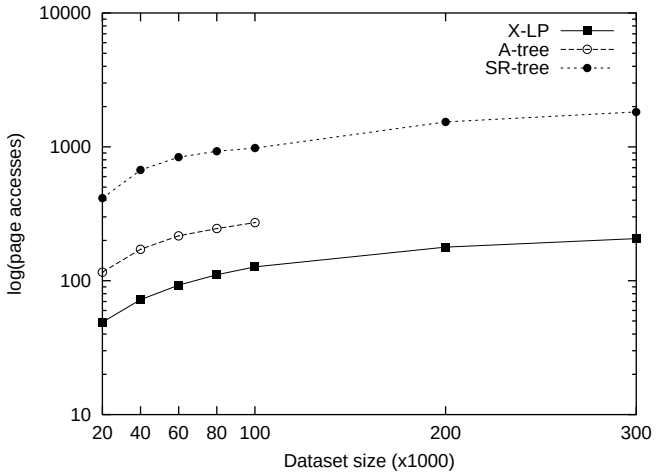


Figure 32: Number of page accesses for varying data partition sizes - uniform dataset

Figure 32 shows the number of page accesses for the three indexing structures for several dataset sizes on 64 dimensional uniform data. The page size is 4kb for all index structures. Our approach uses the Last Page split strategy and it is consistently better than the A-tree and SR-tree. Unfortunately, the A-tree did not allow us to extend our comparison to larger data sets. In average, the SR-tree accessed 8.61 times more disk pages then our structure, while the A-tree required 2.29 times more disk

page reads. As the number of disk accesses for the SR-tree is considerably higher than for our approach and the A-tree, we use a logarithmic scale for the *y-axis* not only here, but also whenever we compare against the SR-tree.

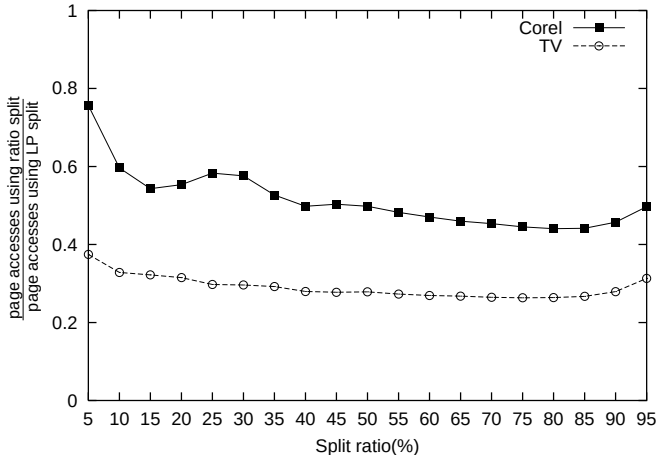


Figure 33: Number of page accesses using ratio split over LP split - real datasets

As mentioned during our theoretical analysis, we believe the Last Page split is not suitable for real datasets due to a different dimensional distribution of data objects. In order to test this, we have performed tests on the real sets using both the Last Page split and ratio based splits. As Figure 33 shows, the ratio based split is better for any split ratio for both real sets. For Corel collection and TV snapshots we used all data objects available, 4kb page size and 64 dimensional feature vectors. The improvements in page accesses of ratio split over the Last Page split is larger for the TV dataset (up to 4 times less page accesses) than for Corel set (around 2 times less page accesses). This is due to the larger size of the TV dataset - almost twice the number of objects in the Corel dataset, as well as a different objects distribution.

To determine the best split ratio for real datasets, we have performed tests on various combinations of data dimensionality and page sizes for Corel (see Figure 34) and TV (see Figure 35) datasets. Based on these experimental observations, we determined that an 80% split ratio results in the lowest page access requirements on most cases for both datasets. For the cases when the 80% split ratio is not the best, the differences in page accesses between using 80% ratio and the best ratio is very small. Therefore, in all following tests involving our approach on real data sets we will use an 80% split ratio.

Figures 36 and 37 show the number of page accesses for varying data subset sizes taken from Corel and TV datasets. We used 64 dimensions and 4kb page size. Our approach performs better than the A-tree and SR-tree. The improvements are increasing with increasing number of objects in the dataset. While the improvements of our approach are smaller compared to the A-tree (up to 45% more page accessed by the A-tree), they are substantial with respect to the SR-tree (up to 500% more pages accessed by the SR-tree).

To study the scalability with respect to increasing dimensions of the three indexing structures, we have extracted feature vectors of different dimensionality from the Corel collection. The disk page size was again set to 4kb and we used all objects available in the dataset. As Figure 38 shows, the number of page accesses increases with dimensionality increase, while the advantage of our method over the two other indexing structures remains stable (around 30% (80%) less page accesses for X-80% than the A-tree (SR-tree)).

When increasing the number of nearest neighbors, the improvement of our method is slowly de-

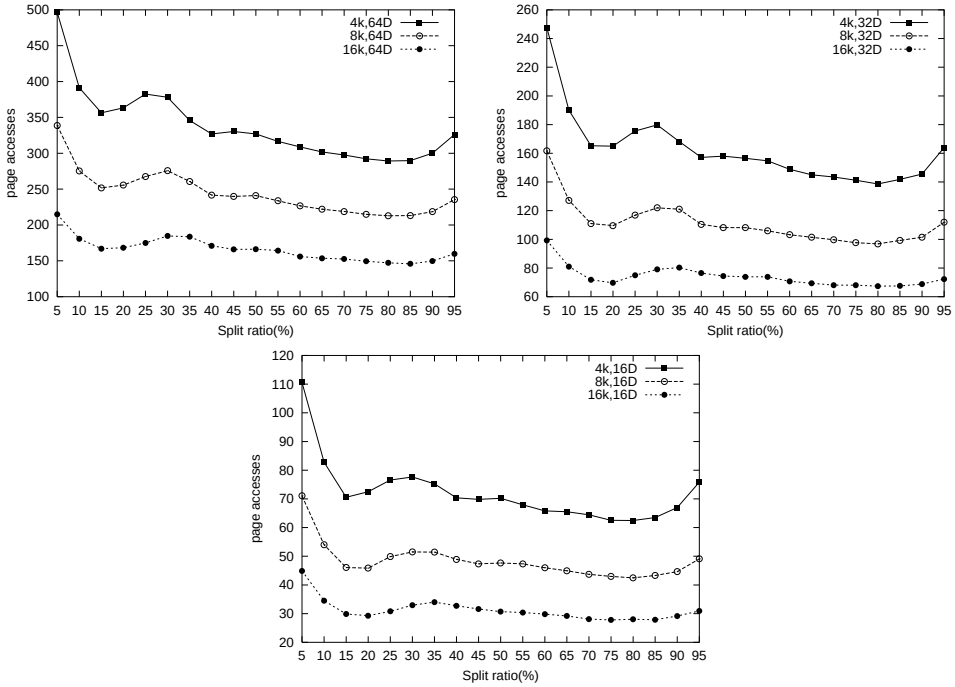


Figure 34: Split ratio efficiency for several combinations of data dimensionality and page sizes - Corel dataset

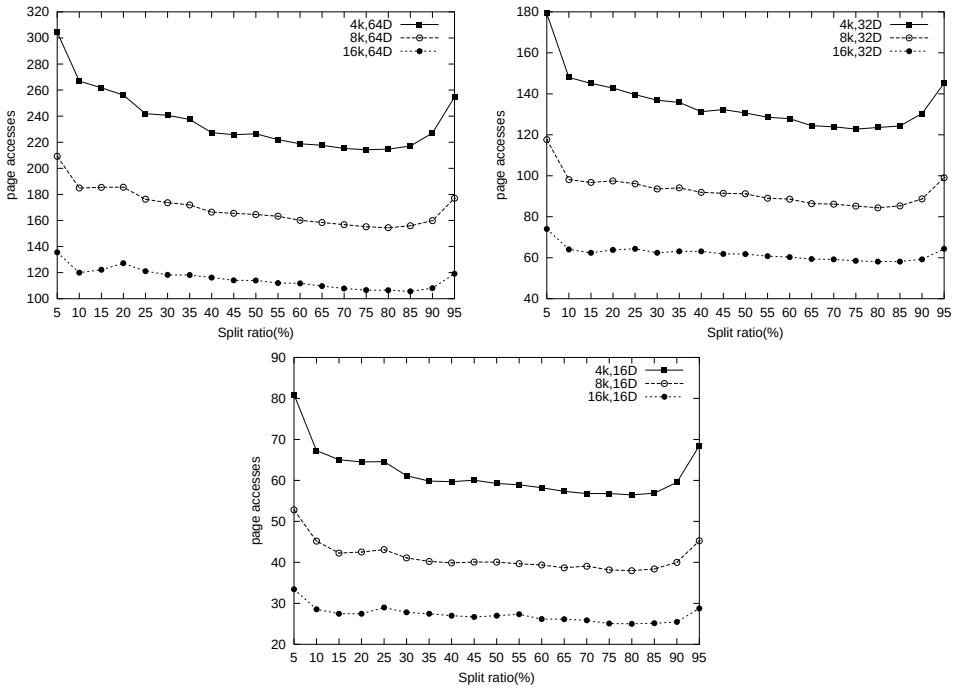


Figure 35: Split ratio efficiency for several combinations of data dimensionality and page sizes - TV dataset

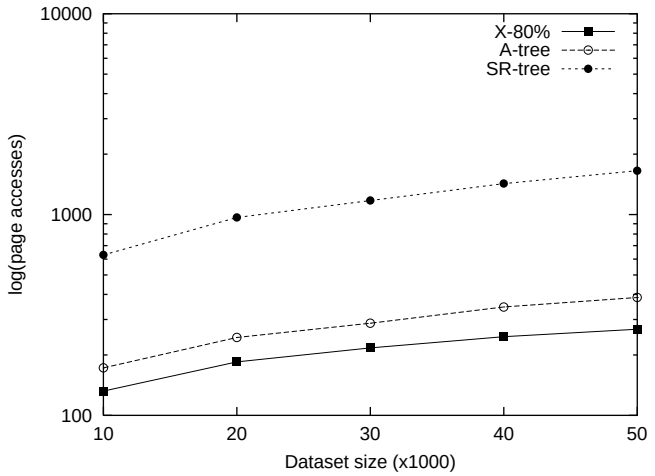


Figure 36: Number of page accesses for varying data set sizes - Corel dataset

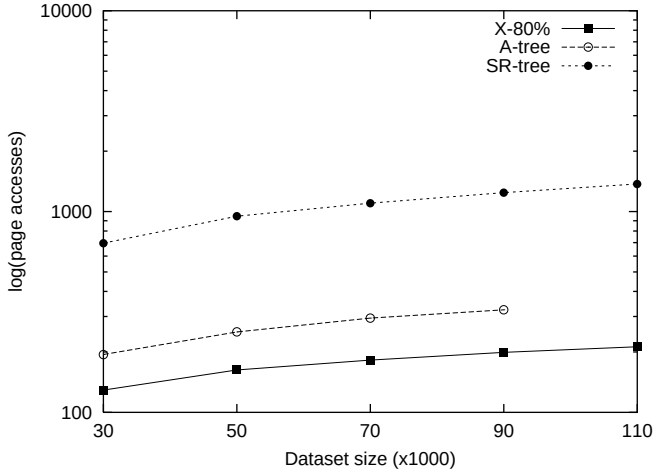


Figure 37: Number of page accesses for varying data set sizes - TV dataset

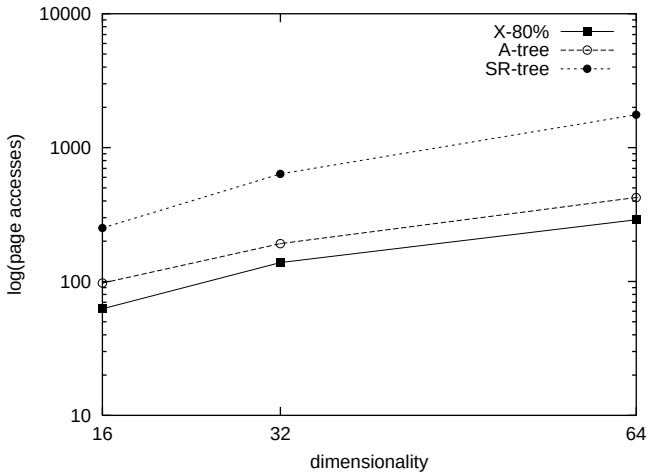


Figure 38: Number of page accesses for varying number of dimensions - Corel dataset

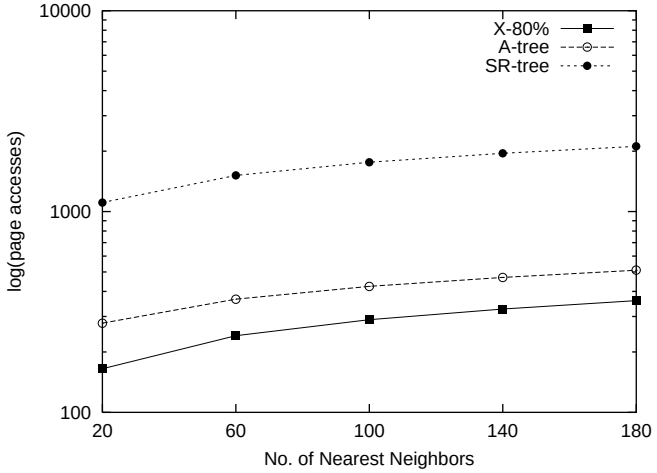


Figure 39: Number of page accesses for varying number of neighbors - Corel dataset

creasing, while still being significantly better even for 180 nearest neighbor search. Although Image Retrieval Web engines search for around 15 to 25 nearest neighbors for various reasons, in all previous experiments we have used 100 nearest neighbor search, as we believe that 100 is a reasonable number of nearest neighbor images that a user is willing to visually check. While our method proves to be the best every time, its advantage over the other two indexing structures is even larger if we chose a smaller number of neighbors in all experiments. For example, while for 100 neighbors search the X-80% accesses 31.79% less pages than the A-tree, for 20 neighbors the saving increases to 40.73% less page accesses.

Overall, our method consistently retrieved the nearest neighbors using the smallest number of page accesses among all tested indexing techniques. For uniform data, our theoretical analysis about the ideal split proved to be correct. For real data we determined experimentally that an 80% split is a good choice for both real sets and different combinations of data dimensionality and disk page sizes. As shown by all experimental results, the improvements shown by our method over the SR-tree and A-tree index structure are consistently better and stable with respect to variation in dataset size, length of feature vectors and total number of pages (determined by page size variation).

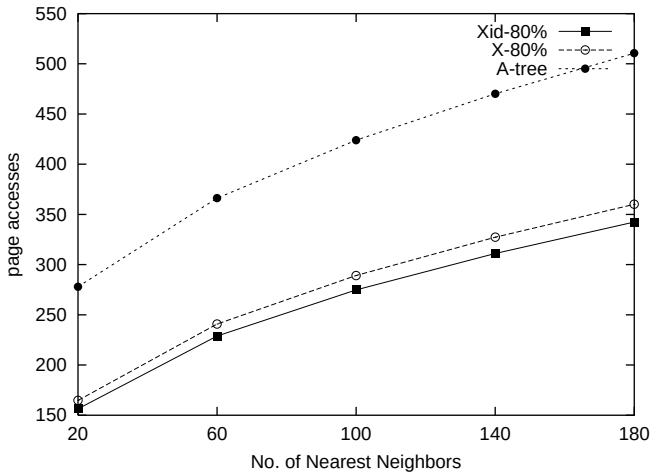


Figure 40: Overall number of page accesses for varying number of neighbors - Corel dataset

Since in our research we have also proposed a new distance function, we decided to use it simultaneously with the proposed index to show the overall gain obtained. We know from Section4.5 that the improvements brought by the new distance are consistent regardless the variation of different parameters (such as data dimensionality, page size, number of neighbors a.o.). Therefore, we will show the overall results of combining the new distance and the new index for only one parameter, that is the number of nearest neighbors. Figure 40 shows a comparison among the number of page accessed using the A-tree index, our X-80% index and the X-80% index combined with the use of the improved query-to-MBR distance (denoted as Xid-80%). We used the Corel dataset with 64 dimensions and a 4kb page size. Since we are not comparing to the SR-tree which was always much slower, this graph does not use a logarithmic scale. As the number of nearest neighbors increases, the proposed distance helps improve the number of page accesses obtained by X-80% algorithm, making the overall gain of our proposed solution over the existing indexing structures even larger (Xid-80% accesses 36.5% less pages than the A-tree as compared with the X-80% which accesses 33% less pages than the A-tree). Although the improvement brought by the use of our proposed distance are low as also shown in Section 4.5, it helps the proposed index structure improve even more over the existing structures.

5 Conclusions

5.1 Summary & Contributions

In our research we have explored the use of *a priori* knowledge about the data to be indexed in order to enhance the performance of retrieval. Based on a concrete example – use of color histograms in Image Information Retrieval, we have shown how constraints in normalized histogram data can be used at both index building and retrieval time to improve the search efficiency.

The experimental results show that the proposed solution is efficient and improves over existing state-of-the-art techniques. Experiments were performed on uniform distributed data as well as two real image collections. For all tests, our proposed algorithm outperformed other indexing techniques in the number of page accesses required during retrieval. We have also shown that a solution that proves efficient for the uniform data set, it is not necessary the best for real collections. Therefore, a parameter (the split ratio) has to be adapted for the type of collection the user is dealing with. For our technique, we have shown that the split policy is capital for achieving good performances. While a Last Page split is the best for uniform data, an 80% split proved to be the best choice for indexing of both real datasets.

We have also proposed a new definition of a distance function that proved to be is more accurate than the usual one in approximating the distance between a query point and a possible data point inside an MBR. Experiments showed that although the number of page accesses saved by using the new distance is not very large, the distance can help improve an indexing technique based on Minimum Bounding Rectangles.

Finally, the overall performance of our indexing technique is shown to be efficient and stable with respect to variations in number of data objects, page size, feature vector dimensionality and variation in number of nearest neighbors to be searched for. Thus, this technique should be valuable in practical applications.

5.2 Future Research

The scope of this report was to explore how one can take advantage of constraints existing inherently in a data collection in order to improve the efficiency for nearest neighbor queries. In the course of our exploration we have touched several other related issues that could lead to promising and challenging areas for future research. These would include:

- Determining an adaptive split ratio during index building. It may be based on the data distribution in the current step of index construction or it may take into account what split choices have been taken in the previous steps.
- In this report, we have taken advantage of the data constraints at bulkload building time. Since the *a priori* knowledge that we used does not depend on the number of objects but on the object's characteristics, an index structure using dynamic objects insertion may be able to take advantage of the existing constraints as well.
- While the proposed distance function has not improved very much our indexing technique, there may be indexing techniques using other data organization policies where the distance may be more helpful.
- It would be an interesting issue to investigate similar optimized split policies and the proposed distance definition within index structures using quantization techniques, such as the A-tree.

References

[1] Adobe Systems, Inc., Color management systems - technical guide. www.adobe.com/support/techguides/color/colormanagement/cmsdef.html.

[2] M. Ankerst, G. Kastenmullera, H.-P. Kriegel, and T. Seidl. 3d shape histograms for similarity search and classification in spatial databases. In *Proceedings of the International Symposium on Advances in Spatial Databases*, pages 207–226, 1999.

[3] J.R. Bach, C. Fuller, A. Gupta, A. Hampapur, B. Horowitz, R. Humphrey, R. Jain, and C.F. Shu. The Virage image search engine: An open framework for image management. In *Proceedings of SPIE Storage and Retrieval for Image and Video Databases*, pages 76–87, 1996.

[4] N. Beckmann, H. Kriegel, R. Schneider, and B.Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[5] S. Berchtold, C. Böhm, D.A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 78–86, 1997.

[6] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 216–230, 1998.

[7] S. Berchtold, D.A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the International Conference on Very Large Databases*, pages 28–39, 1996.

[8] A. Del Bimbo. *Visual Information Retrieval*. Morgan Kaufmann Publishers, Inc., 1999.

[9] A. Del Bimbo, W.-X. He, and E. Vicario. *Image Description and Retrieval*, chapter Using weighted spatial relationships in retrieval by visual content. Plenum Press Publishers, 1998.

[10] A. Del Bimbo, M. Mugnaini, P. Pala, and F.Turco. Visual querying by color perceptive regions. *Pattern Recognition*, 31(9):1241–1253, 1998.

[11] S. Blott and R. Weber. A simple vector approximation file for similarity search in high-dimensional vector spaces. Technical report, Esprit Project HERMES, 1997.

[12] C. Böhm. *Efficiently Indexing High-Dimensional Data Spaces*. PhD thesis, Department of Computing Science, University of Munich, 1998.

[13] C. Böhm and H.-P. Kriegel. Efficient bulk loading of large high-dimensional indexes. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery*, pages 251–260, 1999.

[14] BSIm web page. http://db.cs.ualberta.ca/BSIm/.

[15] N.G. Colossi and M.A. Nascimento. Benchmarking access structures for high-dimensional multimedia data. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, pages 1215–1218, 2000.

[16] Corel Corp. Corel Gallery 1,000,000. http://www.corel.com.

[17] J. Dowe. Content-based retrieval in multimedia imaging. In *Proceedings of SPIE Storage and Retrieval for Image and Video Databases*, pages 164–167, 1993.

[18] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994.

[19] S. Fan. Indexing and retrieving shapes via distance histograms. Master's thesis, Department of Computing Science, University of Alberta, 2001.

[20] R. Finkel and J. Bentley. Quad-trees: A data structure for retrieval on composite keys. *ACTA Informatica*, 4:1–9, 1974.

[21] J.D. Foley, A.V. Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison-Wesley Publishing Company, Inc., 1990.

[22] T. Gevers and W.M. Smeulders. A comparative study of several color models for color image invariant retrieval. In *Proceedings of the International Workshop od Image Database and Multimedia Search*, pages 17–23, 1996.

[23] R.C. Gonzalez and R.E. Wood. *Digital Image Processing*. Addison-Wesley Publishing Company, Inc., 1993.

[24] V.N. Gudivada and V.V. Raghavan. Design and evaluation of algorithms for image retrieval by spatial similarity. *ACM Transactions on Information Systems*, 13(2):115–144, 1995.

[25] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[26] J.L. Hafner, H.S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):729–736, 1995.

[27] C.E. Jacobs, A. Finkelstein, and D.H. Salesin. Fast multiresolution image querying. *Computer Graphics*, 29:277–286, 1995.

[28] H.V. Jagadish. A retrieval technique for similar shapes. In *Proceedings of the ACM SIGMOD*, pages 208–217, 1991.

[29] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 369–380, 1997.

[30] K.-I. Lin, H.V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.

[31] S. Lin. An extensible hashing structure for image similarity searches. Master's thesis, Department of Computing Science, University of Alberta, 2000.

[32] G. Lu. *Multimedia Database Management Systems*. Artech House Publishers, Inc., 1999.

[33] B.S. Manjunath and W.Y. Ma. Image indexing using a texture dictionary. In *Proceedings of SPIE conference on Image Storage and Archiving System*, pages 288–298, 1995.

[34] IBM/NASA's Satellite Image Retrieval System web page. http://maya.ctr.columbia.edu:8080/.

[35] R. Mehrotra and J.E. Gary. Similar-shape retrieval in shape data management. *IEEE Computer*, 28(9):57–62, 1995.

[36] M.A. Nascimento and V. Chitkara. Content-based image retrieval using binary signatures. In *Proceedings of the ACM Symposium on Applied Computing*, pages 687–692, 2002.

[37] M.A. Nascimento, E. Tousidou, V. Chitkara, and Y. Manolopoulos. Color based image retrieval using signature trees. Technical report, Department of Computing Science, University of Alberta, 2001.

[38] W. Niblack, R. Barber, W. Equitza, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, and C. Faloutsos. The QBIC project: Querying images by content using color, texture and shape. In *Proceedings of SPIE Storage and Retrieval for Image and Video Databases*, pages 173–187, 1994.

[39] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(2):38–71, 1984.

[40] G. Pass, R. Zahib, and J. Miller. Comparing images using color coherence vectors. In *Proceedings of the ACM International Conference on Multimedia*, pages 65–73, 1996.

[41] A. Pentland, R.W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. *International Journal of Computer Vision*, 18(3):233–254, 1996.

[42] Photobook Image Search System web page. http://www−white.media.mit.edu/∼tpminka/photobook/.

[43] QBIC web page. http://wwwqbic.almaden.ibm.com/.

[44] RetrievalWare Image Search System web page. http://vrw.excalib.com/cgi-bin/sdk/cst/cst2.bat.

[45] J. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[46] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 71–79, 1995.

[47] Y. Rui, T. Huang, and S.F. Chang. Image retrieval: Past, present, and future. In *International Symposium on Multimedia Information Processing*, volume 10, pages 1–23, 1997.

[48] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *Proceedings of the International Conference on Very Large Data Bases*, pages 516–526, 2000.

[49] L. Shapiro and G. Stockman. *Computer Vision*. Prentice Hall Publishing Company, Inc., 2001.

[50] J.R. Smith and S.-F. Chang. Automated binary texture feature sets for image retrieval. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 2239–2242, 1996.

[51] J.R. Smith and S.-F. Chang. VisualSEEk: a fully automated content-based image query system. In *ACM Multimedia*, pages 87–98, 1996.

[52] R.O. Stehling, M.A. Nascimento, and A.X. Falcao. On "shapes" of colors for content-based image retrieval. In *Proceedings of the International Workshop on Multimedia Information Retrieval (MIR'2000)*, pages 171–174, 2000.

[53] M.A. Stricker and M. Orengo. Similarity of color images. In *Proceedings of SPIE: Storage and Retrieval for Image and Video Databases*, pages 381–392, 1995.

[54] Virage's VIR Image Engine system web page. http://www.virage.com/cgi-bin/query-e.

[55] VisualSEEk Image Search System web page. http://www.ee.columbia.edu/∼sfchang/demos.html/.

[56] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the International Conference on Very Large Databases*, pages 194–205, 1998.

[57] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 516–523, 1996.