A Simple Vector-Approximation File for Similarity Search in High-Dimensional Vector Spaces*

Stephen Blott
Institute of Information Systems
ETH Zentrum
8092 Zurich, Switzerland
blott@research.bell-labs.com

Roger Weber
Institute of Information Systems
ETH Zentrum
8092 Zurich, Switzerland
weber@inf.ethz.ch

Abstract

Many similarity measures in multimedia databases and decision-support systems are based on underlying vector spaces of high dimensionality. Data-partitioning index methods for such spaces (for example, grid files, R-trees, and their variants) generally work well for low-dimensional spaces, but perform poorly as dimensionality increases. This problem has become known as the 'dimensional curse'.

This paper introduces the *vector-approximation file* (VA-File) for similarity search in high-dimensional vector spaces. The VA-File overcomes the dimensional curse by following not the data-partitioning approach of conventional index methods, but rather the filter-based approach of signature files. Space—*not data*—is divided into cells, and vectors are assigned approximations based on the cells in which they lie. A VA-File contains these small, bit-encoded approximations. By scanning first all of the smaller approximations, nearest-neighbor queries need only visit a fraction of the vectors themselves. Thus, the VA-File is used as a simple filter, much as a signature file is a filter.

The performance of the VA-File is evaluated on the basis of real and semi-synthetic degree-45 vector data characterizing color features of a moderate-sized image database. We show that performance does not degrade, and even improves marginally with increased dimensionality.

1 Introduction

Classical database search is boolean: given a query q, each object either satisfies q's predicate, or it does not. Multimedia databases, on the other hand, require similarity search also called *nearest neighbor search*: find the 10 objects, say, most similar to q. Similarity search methods are fundamental to a number of application areas, including information retrieval and textual databases, spatial and geographical information systems, decision-support systems, multimedia retrieval systems [NBE+93, Fli95, CHP+95, Fag96].

*This work has been partially funded in the framework of the European ESPRIT project HERMES (project no. 9141) by the Swiss *Bundesamt für Bildung und Wissenschaft* (BBW, grant no. 93.0135).

Similarity is typically measured not on objects directly, but rather on abstractions of objects termed *features*. Feature extraction is application specific. However, the features themselves are often points in high-dimensional vector spaces [Jag91, NBE+93, FEF+94, MG95, PF96]. An example is the color similarity measure of Stricker and Orengo [SO95]. Features are degree-9 vectors of integers containing the average, standard deviation and skewness of pixel values for each of the red, green and blue channels (i.e. $3 \times 3 =$ degree-9). Nearness is defined in terms of a distance function between these points in 9-dimensional space based, for example, on Manhattan distance, or Euclidean distance. For a good feature, nearness in the vector space corresponds to some natural notion of similarity between the objects themselves. In the example above, nearness corresponds to similarity of colors.

Efficient querying of multimedia datasets, therefore, requires efficient similarity search in high-dimensional vector spaces. This paper proposes a new method based upon a *vector-approximation file* (VA-File). Rather than requiring large volumes of primary vector data to be scanned, the VA-File contains small approximations to each vector; typically these occupy only around 10% or 15% of the space of the vectors themselves. Based only on these smaller approximations, the vast majority of the vectors can be excluded from a search. Thus, the VA-File is used as a simple filter, much as a signature file is a filter [Fal85, FC87]. Queries themselves are also points in the same space as the vector data. The only query provided so far is nearest neighbor search. However, partial match, weighted search and range queries are simple extensions.

The main advantage of the new method is that the VA-File retains good performance as dimensionality increases. Indeed, as is shown in Section 4.5, performance can even improve with dimensionality. In this respect, it overcomes the dimensional curse which plagues most existing methods (see [BAK96] for discussion about R*-tree). Further important advantages accrue from the VA-File's simple flat structure. Parallelization and distribution [Sch96] are relatively straight-forward whereas most other access methods hardly enable such an implementation (e.g. [KF92] presents a parallel R-tree). Moreover, weighted search, which is important for incorporating feedback from users, partial match and conjunctive queries are well supported.

The remainder of this paper is structured as follows. Section 2 provides background material and discusses related work. Section 3 describes the VA-File, and Section 4 provides a performance evaluation. Section 5 discusses the method, and Section 6 concludes.

2 Background, Related Work, and the Dimensional Curse

Many similarity measures, particularly for multimedia data, are based on underlying vector spaces [Jag91, NBE+93, FEF+94, MG95, SO95, PF96]. In such spaces, features are d-dimensional points. An important special case for geographical systems is $d = 2$. For multimedia applications, however, dimensionality is typically somewhat higher: for example, 9 in [SO95], around 30 in [PF96], 45 in recent extensions to [SO95], and 64 in [FEF+94]. Further, the trend appears to be towards still higher dimensionality in the future. Similarity is measured in terms of 'nearness', where the classical nearness measures are the L_p-metrics: L_1 for Manhattan distance, L_2 for Euclidean distance, etc.

In the following, dimensions are assumed to be independent from each other. However, real data in high-dimensional space are often correlated and clustered, and consequently the data oc-

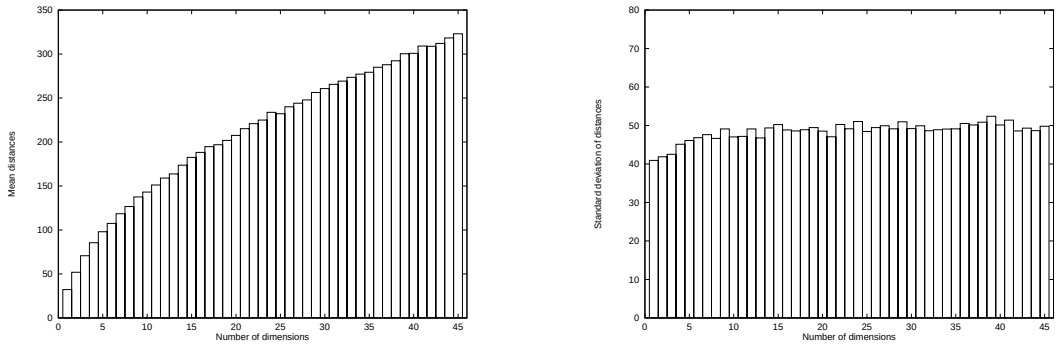


Figure 1: Mean distances increase, while standard deviations remain unchanged

cupy only a subspace. Therefore, a variety of transformations such as Singular Value Decomposition (SVD) [PFTV88, Str80], Karhunen-Loève (KL) [FL95, Fuk90] or FastMap [FL95] must be applied to reduce the effective dimensionality [Pra91]. We assume that such transformations are done during feature extraction and that dimensions are not correlated.

2.1 Data-Partitioning Index Methods

There exist a large number of index structures for vector spaces [NHS84, Gut84, SRF87, BKSS90, LJF94, BAK96, Lom90, ELS95], and a relevant survey has been provided by Bentley and Friedman [BF79]. Most of these methods are based on variations of the following approach. Space is partitioned into cells, and a page is allocated for each cell. Data is partitioned, and vectors (or references to vectors) are allocated to the pages corresponding to the cells in which they lie. An index structure over these pages is built—which can be a tree [BKSS90] or a directory [NHS84]—and this index is used to prune search spaces during query processing.

Similar approaches have also been proposed for the more general class of metric spaces [Chi94, Bri95, CZR96]. Metric spaces are those for which a distance function satisfying the triangle inequality exists. The class of metric spaces encompasses vector spaces, however it also includes other measures such as the edit-distance between strings. Indexes for metric spaces typically partition data based on proximity to one or more partition objects. Since the L_p metrics satisfy the triangle inequality, these methods also can be applied for vector spaces. On the other hand, metric spaces may be mapped to some lower dimensional vector space [FL95] which can be efficiently indexed by traditional vector space index structures.

The common aspect of all these approaches is that data is partitioned, and this partitioning is used to prune the search space during query processing. Therefore, we refer to these approaches collectively as *data-partitioning index methods.*

2.2 The Dimensional Curse

Data-partitioning methods generally work well at low dimensionality. However, performance can degrade badly—even exponentially—as dimensionality increases. This phenomenon has been termed the 'dimensional curse'. Its cause is the following. The classical L_p metrics are defined by the equations:

$$L_p(\vec{v}_i, \vec{v}_{i'}) = \left(\sum_{j=1}^{d} |v_{i,j} - v_{i',j}|^p \right)^{\frac{1}{p}} \quad (1)$$

In particular, these metrics are based on sums of terms drawn from independent distributions. As dimensionality increases, the Central Limit Theorem asserts that such sums tend towards a normal distribution. Moreover, the mean increases, while the standard deviation remains roughly unchanged. This effect is illustrated experimentally in Figure 1, which shows the increasing means (left) and unchanged standard deviations (right) for a large of set distance computations. The data in this case was synthetic, drawn from different independent, normally-distributed domains in each dimension.

This effect has a serious impact on data-partitioning index methods. As dimensionality increases, all points begin to appear almost equidistant from one-another. They are effectively arranged in a d-dimensional sphere around a query, no matter where the query is located. The radius of the sphere increases, while the width of the sphere remains unchanged, and the space close to the query point is empty. This makes it increasingly improbable that a pre-computed partitioning of data can in fact usefully prune search spaces, since a different clustering is necessary to support each query. Our practical experiments with an R*-Tree show that this approach is already questionable with dimensionality as low as 5, and unlikely to be of any practical benefit with dimensionality beyond the mid-teens (see Section 4.2). The performance problems with R*-trees arise from the increasing overlap in the directory as the dimensionality grows [BAK96]. Overlap in the internal nodes directly influences the search performance as with increased overlap more paths have to be observed and therefore more random, page-oriented I/O results.

An important implication of this discussion is that the dimensional curse is not a property of an index method alone. Rather, the difficulty stems from the combination of the distance metric and data partitioning. Following this observation, the VA-File described here is based not on data partitioning, but rather on a hybrid of *space*-partitioning and signature methods.

2.3 Signature Methods

Signature indexing methods have been developed for text databases, and multi-key indexing [Fal85, FC87]. The basic idea is the following. A *signature* is a small abstraction of (or approximation to) an object, which is typically encoded as a bit sequence. Given the signatures of a set of objects and of a query, efficient bit-wise operations can be used to eliminate many objects which cannot possibly match the query. Thus, a signature file is in effect a filtering technique. Queries are first compared with all the smaller signatures. The result—hopefully—is a far smaller set of candidate objects, which might still however contain false drops. For these remaining candidates, the objects themselves are visited to determine the final answer set.

d	number of dimensions	j	ranges over dimensions, $j \in \{1, \ldots, d\}$
v	number of vectors	i	ranges over vectors, $i \in \{1, \ldots, v\}$
$\vec{v}_i$	i-th vector	$v_{i,j}$	j-th component of $\vec{v}_i$
b	number of bits per approximation	b_j	bits per approx. in dimension j
a_i	approximation for $\vec{v}_i$	$r_{i,j}$	region into which $\vec{v}_i$ falls in dim. j
$p_j[k]$	k-th partition point in j-th dim.	n	number of objects in result set
$\vec{v}_q$	a query vector	L_p	distance function $L_p(\vec{v}_i, \vec{v}_q)$
l_i, u_i	bounds: $l_i \leq L_p(\vec{v}_q, \vec{v}_i) \leq u_i$	$l_{i,j}, u_{i,j}$	contribution to l_i, u_i for dimension j

Table 1: Notational summary

2.4 The VA-File Approach

Similarly to signature methods, the *vector approximation file* (VA-File) proposed here does not partition data; rather it works as follows. First, the vector space—*not the data*—is partitioned into cells, and these cells are used to generate bit-encoded approximations for each vector. The method used for this stage is similar to those of grid files [NHS84], and partitioned-hashing methods [Ull88], although the heuristics governing cell occupancy are very different. The VA-File itself is simply a flat array of all the approximations. For query processing, all of these approximations are scanned. Each approximation determines a lower and an upper bound on the distance between a vector and the query, and these bounds frequently suffice to eliminate vectors from the search. Notice that the flat file structure and the search algorithm are fundamentally different from those of the data-partitioning methods above. Although perhaps counter-intuitive, this is the key difference here through which the dimensional curse is avoided.

3 The VA-File: Structure, Bounds and Search

This section describes the VA-File method, and is organized as follows. Firstly, the VA-File structure is described. Then, Section 3.2 describes the use of approximations to derive lower and upper bounds on the distances between queries and points. Finally, based on these bounds, Section 3.3 presents two alternative search algorithms.

Assume v is the number of vectors, and d the number of dimensions. We use $i \in \{1, \ldots, v\}$ to range over vectors, and $j \in \{1, \ldots, d\}$ over dimensions, although explicit quantification is usually omitted. Hence, $\vec{v}_i$ denotes an individual vector, and $v_{i,j}$ the j-th component within $\vec{v}_i$. Let a_i denote the approximation of $\vec{v}_i$, and let b be the number of bits required in each approximation. All notation is summarized in Table 1.

3.1 The VA-File Structure

For each vector, a b-bit approximation is derived. The first step is to determine the number of bits b_j to allocate to each dimension j. We use here an even distribution. Hence, the relationship between

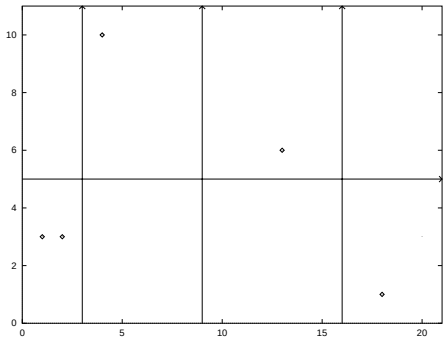


Figure 2: A two-dimensional example ($d = 2$, $b = 3$)

the number of bits (b) and the number of dimensions (d) determines b_j as follows.

$$b_j = \left\lfloor \frac{b}{d} \right\rfloor + \begin{cases} 1 & j \leq b \bmod d \\ 0 & otherwise \end{cases} \tag{2}$$

Typically, b_j is a small integer: say 3, 4, or 5.

The number of bits in each dimension is used to determine partition points, and hence regions within each dimension. In particular, there are 2^{b_j} regions within dimension j, requiring $2^{b_j} + 1$ partition points. The running example in Figure 2 illustrates this. Assuming $d = 2$ and $b = 3$, the number of bits per dimension is given by $b_1 = 2$ and $b_2 = 1$. That is, there are two bits for the x dimension, and one for the y. Hence there are four regions requiring five partition points in the x dimension, and two regions requiring three partition points in the y dimension. Partition points $< p_j[0], p_j[1], \ldots, p_j[2^{b_j}] >$ are computed such that each dimension j is partitioned into into 2^{b_j} equally-full regions. This can be done either precisely by analyzing the entire data set, or stochastically by sampling. The partition points in the example are $< 0, 3, 9, 16, 21 >$ and $< 0, 5, 11 >$ in the x and y dimensions, respectively. Together, these partition points divide space into 2^b cells, where there are $2^3 = 8$ cells in the example.

This method for establishing cells is similar to those used for grid files [NHS84], or for partitioned-hashing schemes [Ull88]. These latter approaches determine cell sizes such that the number of objects occupying each cell roughly fills a data page. Here, however, the heuristic is very different. Of the eight cells in the example, four are empty, and three contain only a single point. In fact, Section 4 considers approximations consisting of around 200 bits, with around 2^{200} cells. In such cases, the vast majority of cells are of course empty.

An approximation a_i for each vector $\vec{v}_i$ is generated as follows. Let the 2^{b_j} regions in dimension j be numbered $0, \ldots, 2^{b_j} - 1$. Let $r_{i,j}$ be the number of the region into which $v_{i,j}$ falls. We define a point to fall into a region only if it is greater than or equal to the lower bound for the region, and strictly less than the upper bound for the region. Therefore, $v_{i,j}$ falls into the region numbered $r_{i,j}$ if and only if:

$$p_j[r_{i,j}] \leq v_{i,j} < p_j[r_{i,j} + 1] \tag{3}$$

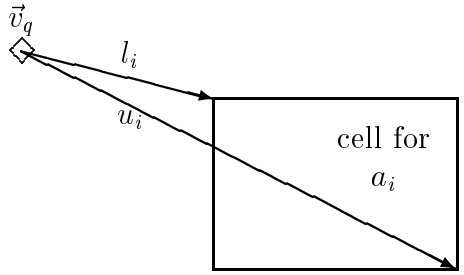


Figure 3: Lower and upper bounds for $L_p(\vec{v}_q, \vec{v}_i)$ given a_i guaranteeing $l_i \leq L_p(\vec{v}_q, \vec{v}_i) \leq u_i$

Consider the point $\vec{v}_i = (13, 6)$ from Figure 2. The 13 falls into region numbered 2, and the 6 falls into region numbered 1. Therefore we have $r_{i,1} = 2$, and $r_{i,2} = 1$. The approximation a_i is simply the concatenation of the binary b_j-bit bit patterns for each such region in turn. Hence, the approximation for point $(13, 6)$ is '101', the concatenation '10' for $r_{i,1}$, with '1' for $r_{i,2}$. For all the points illustrated in Figure 2, the approximations are as follows:

$$(1,3) \rightarrow 000; \quad (2,3) \rightarrow 000; \quad (4,10) \rightarrow 011; \quad (13,6) \rightarrow 101; \quad (18,1) \rightarrow 110;$$

The complete VA-File is simply an array of all these approximations: '000000011101110'. Notice that, unlike data-partitioning approaches such as grid files [NHS84], R-Trees [Gut84] and their variants, data need not be partitioned and clustered in a VA-File. This is because, as Section 2.2 explains, data-partitioning alone cannot overcome the dimensional curse.

3.2 Lower and Upper Bounds on Distances

We now show how an approximation can be used to derive bounds on the distance between a query point and a vector. Assume a query $\vec{v}_q$ and a distance function L_p, for some p. An approximation a_i determines a lower bound l_i, and an upper bound u_i such that:

$$l_i \leq L_p(\vec{v}_q, \vec{v}_i) \leq u_i \tag{4}$$

This is sketched in Figure 3. Intuitively, a_i contains sufficient information to determine the cell in which $\vec{v}_i$ lies. The lower bound l_i is simply the shortest distance from the query to a point in that cell. Similarly, the upper bound u_i is the longest distance to a point in that cell.

Formally, l_i and u_i are derived as follows. As with data vectors, a query $\vec{v}_q$ consist of components $v_{q,j}$, and these fall into regions numbered $r_{q,j}$. Also, given a_i, components $r_{i,j}$ are easily extracted.

Based on this information, the bounds l_i and u_i are defined by the equations:

$$l_i = \left(\sum_{j=1}^{d} {l_{i,j}}^p\right)^{\frac{1}{p}} \quad \text{where } l_{i,j} = \begin{cases} v_{q,j} - p_j[r_{i,j}+1] & r_{i,j} < r_{q,j} \\ 0 & r_{i,j} = r_{q,j} \\ p_j[r_{i,j}] - v_{q,j} & r_{i,j} > r_{q,j} \end{cases}$$

$$u_i = \left(\sum_{j=1}^{d} {u_{i,j}}^p\right)^{\frac{1}{p}} \text{ where } u_{i,j} = \begin{cases} v_{q,j} - p_j[r_{i,j}] & r_{i,j} < r_{q,j} \\ \max(v_{q,j} - p_j[r_{i,j}],\ p_j[r_{i,j}+1] - v_{q,j}) & r_{i,j} = r_{q,j} \\ p_j[r_{i,j}+1] - v_{q,j} & r_{i,j} > r_{q,j} \end{cases} \tag{5}$$

These definitions follow from Equations 1 and 3.

Returning to the example, assume the distance metric L_1, and consider $\vec{v}_i$ to be the example point $(13, 6)$, and $\vec{v}_q$ to be the query $(20, 3)$. Hence $r_{q,1} = 3$, and $r_{q,2} = 0$. Then $l_i = 4 + 2 = 6$, which is the distance from $(20, 3)$ to the nearest point $(16, 5)$ in the cell identified by a_i. Similarly, $u_i = 11 + 8 = 19$, which is the distance to the furthest point $(9, 11)$ in the same cell.

It is straight-forward to see that l_i and u_i are indeed lower and upper bounds for $L_p(\vec{v}_q, \vec{v}_i)$. This follows from $l_{i,j} \leq |v_{q,j} - v_{i,j}| \leq u_{i,j}$, which in turn follows from Equation 3.

3.3 Two Search Algorithms

The result of a query must be the n vectors closest to $\vec{v}_q$. This section describes two different search algorithms for finding those vectors.

3.3.1 A 'Simple-Search' Algorithm (VA-SSA)

The simple-search algorithm, known as *VA-SSA*, is described in pseudo code in Figure 4. Arrays `ans` and `dst` are maintained of the nearest n vectors encountered so far, and their distances to $\vec{v}_q$, respectively. These are sorted in order of increasing distance. The approximations are scanned linearly. A vector is a candidate whenever less than n vectors have been encountered, or whenever the lower bound l_i is less than the distance of the n-th vector currently in the answer set. The actual distance based on L_p is evaluated only for these candidate vectors, hence only these candidates are visited. Thus, the VA-File is used as a simple filter.

The major advantages of this algorithm are simplicity, a low memory overhead, and that vectors are visited sequentially (i.e. there are no long, random seeks). On the other hand, the performance of simple-search depends upon the ordering of the vectors and approximations. Despite this potential difficulty, Section 4 shows that this algorithm performs well in practice.

3.3.2 A 'Near-Optimal' Search Algorithm (VA-NOA)

Assume an optimal algorithm named 'miracle' which minimizes the number of vectors visited. The second algorithm described here is equivalent to 'miracle' for all but a few highly-improbable cases. The cost of near optimality is additional complexity and memory overhead. The near-optimal algorithm, known as *VA-NOA*, is sketched in Figure 5, and has two phases. The first phase consists of an information-gathering scan of the approximations. The second phase visits vectors based on the results of the first phase, thereby determining the final answer set.

```
VAR ans, dst: ARRAY of INT;  a: ARRAY of Approx;  v: ARRAY of Vector;

FUNC InitCandidate(): INT;
VAR k: INT;
   FOR k := 1 TO n DO
      dst[k] := MAXINT;
   END;
   RETURN MAXINT;
END InitCandidate;

FUNC Candidate(d, i: INT): INT;
   IF d < dst[n] THEN
      dst[n] := d; ans[n] := i;
      SortOnDst(ans, dst, n);
   END;
   RETURN dst[n];
END Candidate;

PROC VA-SSA(v_q: Vector);
VAR i, d: INT;
   d := InitCandidate();
   FOR i := 1 TO v DO
      l_i := GetBounds(a_i, v_q);
      if l_i < d THEN
         d := Candidate(L_p(v_i, v_q), i);
   END;
END VA-SSA;
```

Figure 4: The Simple-Search Algorithm: *VA-SSA*

During the first phase, the approximations are scanned, and the bounds l_i and u_i are computed for each vector. Further, the following filtering step is then applied. Assume `d` is the n-th largest upper bound encountered so far. If an approximation is encountered such that l_i exceeds `d`, then v_i can be eliminated since n better candidates have already been found. In our practical experiments, between 85% and 96% of the vectors were eliminated during this first filtering step. Let N_1 denote the number of candidates remaining after this first phase. N_1 proved to be sub-linear in the database size. The space overhead of this algorithm is proportional to N_1.

During the second phase, vectors are visited in increasing order of l_i to determine the final answer set. Not all the remaining candidates must be visited. Rather, this phase ends when a minimum bound is encountered which exceeds or equals the n-th distance in the answer set. The data structure supporting this is a heap-based priority queue [Sed83]. Let N_2 denote the number of steps performed during this phase. Then the total number of additional steps for this algorithm is roughly $N_2 \log_2 N_1$. In particular, only the N_2 elements necessary to complete the search are sorted.

This algorithm is near optimal in the sense that only under degenerate conditions it is possible to have a search algorithm which visits fewer vectors.[1] Assume some 'miracle' algorithm which visits first

[1]The degenerate, non-optimal conditions are the following. Assume $\vec{v}_i$ is the final member of the answer set found by *VA-NOA*. Further, let `d` be $L_p(\vec{v}_i, \vec{v}_q)$. Then non-optimality occurs only if: 1) $l_i =$ `d`, 2) the minimum bound of the previously-visited vector is also `d`, 3) no vector is visited by *VA-NOA* after $\vec{v}_i$, and 4) no vector in the answer set has a

```
PROC VA-NOA(v_q: Vector);
VAR i, d, l_i, h_i: INT; Heap: HEAP; Init(Heap);

(* PHASE - ONE *)
   d := InitCandidate();
   FOR i := 1 TO v DO
      l_i, u_i := GetBounds(a_i, v_q);
      IF l_i <= d THEN
         d := Candidate(u_i, i);
         InsertHeap(l_i, i);
      END;
   END;

(* PHASE - TWO *)
   d := InitCandidate();
   l_i, i := PopHeap(Heap);
   WHILE l_i < d DO
      d := Candidate(L_p(v_i, v_q), i);
      l_i, i := PopHeap(Heap);
   END;
END VA-NOA;
```

Figure 5: The Near-Optimal Search Algorithm: *VA-NOA*

the n vectors of the answer set, and then the remaining vectors for which l_i is strictly less than the distance of the n-th nearest vector. This miracle algorithm is optimal in the sense that no algorithm can guarantee correctness without visiting at least these vectors. Now, consider instances of both this miracle algorithm and the near-optimal algorithm above. Assume v_i is the last member of the answer set found by the near-optimal algorithm. Then l_i is a minimum bound on $L_p(\vec{v}_i, \vec{v}_q)$. Furthermore, since the near-optimal algorithm visits vectors in increasing order of the lower bound, the lower bounds of all previously-visited vectors must be less than or equal to l_i. Therefore, these are also visited by the miracle algorithm (but see footnote). When the near-optimal algorithm visits v_i, the n nearest vectors are found, although the termination condition might not yet be reached. However, all vectors $v_{i'}$ visited subsequently have the property that $l_{i'}$ is strictly less than the distance to the n-th nearest vector. Therefore, these vectors too are visited by both algorithms. In fact, with the exception of the degenerate conditions mentioned above, the sets of vectors visited by both algorithms are the same.

3.3.3 Optimizations

A number of computational optimizations are possible which, although not significant in terms of complexity, have a considerable impact in practice. Notably, Equations 1 and 5 require p-th roots to be taken. Since only the ordering matters in practice, and this is preserved when taking the p-th root, these roots need not actually be computed. Also, computations of these terms involve iteration over all dimensions. These iterations can be terminated early if a distance exceeds the maximum of interest. These optimizations were included in the implementations of both the VA-File and the *Scan* methods discussed in the next section.

A further important optimization is possible for the VA-File methods. The L_p metrics are based on repeatedly summing the power of a difference (see Equations 1 and 5). With approximations, only a

distance to the query larger than d. These conditions are exceedingly improbable in practice.

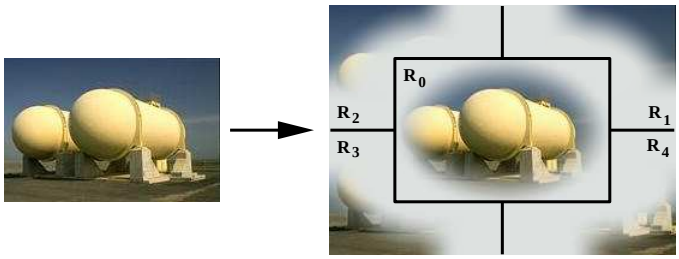


Figure 6: Fuzzy regions: Center, top–left, top–right, bottom–left, bottom–right

limited number of distances can arise; specifically, those between a query point and the partition points. Therefore, these distances and their powers can be pre-computed for a query, prior to evaluating the query itself. This optimization is important since it implies that the number of complex arithmetical operations is no longer proportional to the database size. Further, the space overhead of pre-computing these values never exceeded two memory pages in the experiments reported. This optimization too was included in the results reported in the next section.

4 Practical Evaluation

Our practical evaluation is based on cooperation with image scientists, the authors of [SO95]. The dataset they use is a moderate-sized image database consisting of 11'648 thumbnails. The similarity method for color distribution described by Stricker and Orengo [SO95] generates degree-9 feature vectors for each image. A newer approach treats five overlapping parts of images separately, as illustrated in Figure 6, and generates degree-45 feature vectors for each image (i.e. $9 \times 5 = 45$). These 11'648 degree-45 vectors formed the basis of our practical evaluation. The distance function is based on the L_2 metric (Euclidean distance). When not stated otherwise, the number of objects in the answer set is ten; that is, $n = 10$. For a number of the experiments we performed, data sets containing far more than 11'648 vectors were required. To obtain these larger database, the 11'648-vector data set was scaled synthetically. The distributions of the original data set were retained within each dimension, while any correlation between dimensions was discarded.

The following four search structures/algorithms were evaluated. The two variants of the VA-File described in Section 3.3 were evaluated: namely, *VA-SSA* for the VA-File with the simple-search algorithm, and *VA-NOA* for the near-optimal algorithm. As an example of a data-partitioning index method, an R*-Tree was evaluated. Finally, the yardstick for most of the experiments described was *Scan*, a well-tuned sequential scan of the vectors themselves.

4.1 Sequential Scan of the Data

The *Scan* algorithm is a simple sequential scan of the vectors themselves, maintaining a ranked list of the nearest n vectors encountered so far. An important advantage of this algorithm is that it is very simple. Moreover, disk technology is optimized for sequential access, and the best performance is

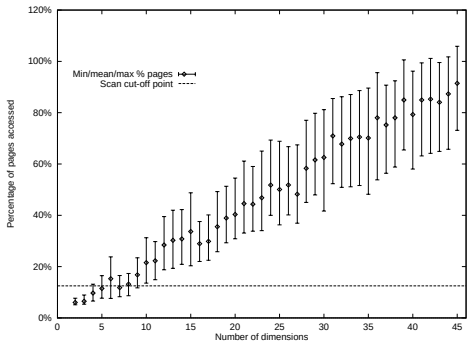


Figure 7: R*-Tree performance as dimensionality increases

typically achieved when the fetch quantum is considerably larger than a page. Because of these factors, a well-tuned *Scan* algorithm frequently out-performs more sophisticated algorithms, since the latter frequently generate random, page-oriented I/O. Hence, for these reasons and for simplicity, many commercial database systems exploit well-tuned *Scan* algorithms in practice. In our experiments, these factors consistently accounted for a factor of 8 improvement in performance (i.e. close to an order of magnitude) over vanilla file-system implementations. Therefore, *Scan* is used as the yardstick for the other methods discussed in this paper.

4.2 A Data-Partitioning Method: R*-Tree

We evaluated an R*-Tree as an example of a data-partitioning index method. The R*-Tree implementation we used incorporates a split policy based on minimal spherical volumes, and similarity search based on the ranking algorithm of Hjaltason and Samet [HS95]. Our experimentation demonstrated good results at low dimensionality, but increasingly poor results at larger dimensionalities, as shown in Figure 7. For each dimensionality, 300 experiments were performed. If p is the number of pages visited by *Scan*, and p' is the number of pages visited in the R*-Tree, then the metric presented is $\frac{p'}{p} \times 100$; that is, the number of pages visited as a percentage of those necessary for a linear scan of the data. This metric is reasonable since it incorporates the facts: 1) that the data volume grows with dimensionality, and 2) that the number of vectors per page decreases with dimensionality. The 'scan cut-off point' is the percentage beyond which the *Scan* algorithm performs better in practice.

Firstly, we note that the results in Figure 7 at degree 2 are consistent with those presented elsewhere for other R-Tree methods [Gut84, BAK96]. However, beyond the first few degrees, the R*-Tree degrades beyond the scan cut-off point, making *Scan* an attractive alternative. In particular, the overhead of the random I/O patterns for the R*-Tree are outweighed by the benefits of the sequential I/O patterns of a well-tuned *Scan* algorithm. Because of these factors, we chose the *Scan* algorithm as the yardstick for the remainder of this paper.

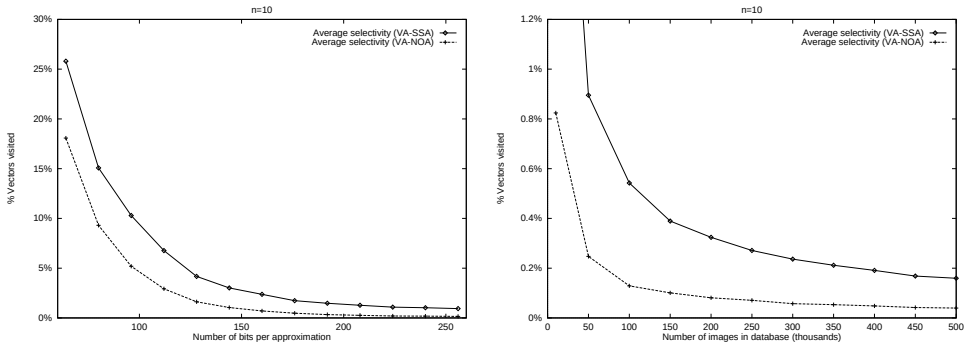


Figure 8: Selectivity as a function: (left) of b, the number of bits per approximation; and (right) of v, the number of vectors in the database.

4.3 Selectivity Experiments

Our first experiments measured the number of times $L_p(\vec{v}_q, \vec{v}_i)$ is computed—hence the number of vectors which are visited—as a percentage of the total number of vectors. Two sets of experiments were performed. Selectivity was measured firstly as a function of the number of bits per approximation (Figure 8, left), and secondly as a function of the database size (Figure 8, right). The experiments were performed for both the simple-search (*VA-SSA*) and near-optimal (*VA-NOA*) algorithms, and the queries were 100 randomly-chosen vectors from the data set itself.

Figure 8 (left) shows improved selectivity as the number of bits in approximations increases. This experiment used the basic set of 11'648 vectors. For 192-bits per signature, the *VA-SSA* method visits less than 2% of the vectors, while the *VA-NOA* method achieves a selectivity of less than 1%. The space overhead of the VA-File itself varies linearly from 22.7 KBytes to 364 KBytes; that is, from 1.11% to 17.78% of the size of the vector data.

Figure 8 (right) shows improved selectivity as the database size increases with the number of bits per approximation (b) fixed at 192. Considering a database of 400'000 vectors, *VA-SSA* visits one vector in every 500 (i.e. 0.2%), while *VA-NOA* visits one vector in every 2'000 (i.e. 0.05%). Although both algorithms are fundamentally linear in the database size, sub-linearity is observed in practice. In the case of *VA-SSA*, the simple-search algorithm, this is because the distance to the n-th nearest vector decreases as the search proceeds. Hence, longer searches in larger databases exhibit improved selectivity. In the case of *VA-NOA*, sub-linearity arises since, assuming the distribution is unchanged, the distance to the n-th nearest vector is typically smaller in larger databases. Therefore, the percentage of vectors for which l_i is less than that distance is also smaller. Consequently, the number of vectors which are visited grows sub-linearly with the database size.

4.4 Wall-Clock (Elapsed Time) Experiments

While the selectivity results above are encouraging, they do not fully reflect reality. For large databases, the factor influencing performance is the percentage of pages visited, not the percentage of vectors

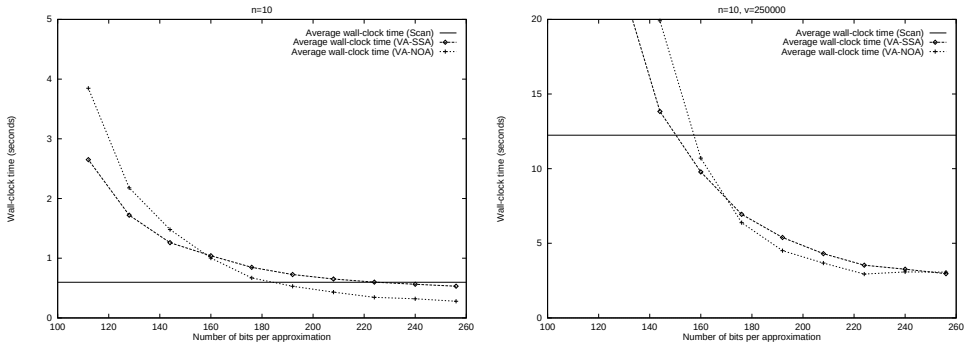
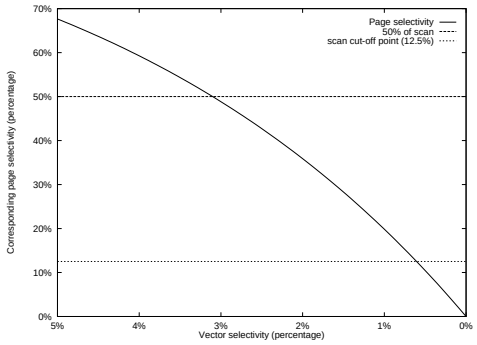


Figure 9: Wall-clock time as a function of b, for (left) 11'648 vectors, and (right) 250'000 vectors

visited. Since vectors share pages, the former is somewhat higher than the latter. Assuming degree-45 vectors, 22 vectors share each 4096-byte page. If p is the probability of visiting a vector, then $1-(1-p)^{22}$ is the probability of visiting a page. Thus, page selectivity as a function of vector selectivity is given by the graph:



For example, a selectivity of 3% at the vector granularity achieves a selectivity of only 50% at the page granularity. A vector selectivity of around 0.6% must be achieved in order to reach the scan cut-off point. For these reasons, we performed a further series of tests based on wall-clock (or elapsed time) measurements for the two VA-File methods, and a competitive implementation of *Scan*. Two data sets were considered: one consisting of the basic set of 11'648 vectors (Figure 9, left), and one consisting of a larger set of 250'000 vectors (Figure 9, right).

All the experiments reported in this section are based on storing data directly on a raw partition of a local disk. This is a realistic assumption for database environments, and eliminates operating-system and network interference. The fetch quantum for the *Scan* algorithm was 400 KBytes. The test machine was a 64MB SUN SPARCstation 4.

The results are illustrated in Figure 9, with the number of bits per approximation being varied up to 256. The experiment was repeated 100 times for each value of b, using randomly-chosen vectors as queries. In both graphs, the horizontal line represents the average wall-clock time for a well-tuned *Scan* algorithm. Notice that *Scan* processes data at close to 4MBytes per second, which is the observed maximum transfer rate of the disk at hand. Thus, it is not possible to have a better *Scan* algorithm

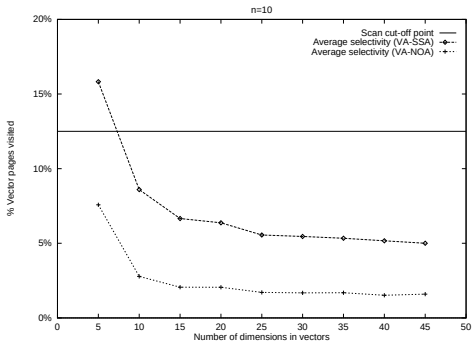


Figure 10: Page selectivity as a function of d, the number of dimensions

with the hardware available.

Firstly, consider the results in Figure 9 (left) for the small data set. Previously (in Figure 8, left) the algorithm *VA-NOA* first achieves a vector selectivity of 0.6% at roughly 224 bits per dimension. This is the selectivity which is required to perform better than the *Scan* algorithm. Correspondingly, *VA-NOA* begins to out-perform *Scan* at this point. Surprisingly, the algorithm *VA-SSA* performs considerably better than expected in practice. This is largely because the I/O patterns generated by *VA-SSA* are sequential, whereas those generated by *VA-NOA* are random. The number of pages visited by *VA-SSA* is in fact higher than the number of pages visited by *VA-NOA*.

The wall-clock times given in Figure 9 (left) are encouraging, but not overwhelming. The results in Figure 9 (right), on the other hand, are very promising. For large databases—which are the interesting ones—the VA-File offers a significant performance improvement over the *Scan* algorithm. In particular, roughly a factor of 4 improvement in response time is observed. Again, the performance of the simple-scan algorithm is surprisingly good, equaling that of the more sophisticated near-optimal algorithm in practice. This, again, is largely because of the sequential access patterns of *VA-SSA*.

4.5 The Dimensional Curse: Performance with Dimensionality

The final aspect of performance which we measured is as a function of dimensionality. As discussed in Section 2.2 and shown in Section 4.2, data-partitioning index methods perform poorly with increased dimensionality. Especially, similarity search suffers from growing dimensionality as more paths or cells have to be observed.

The vector-selectivity measure is inappropriate here since the number of vectors sharing pages decreases as dimensionality increases. The wall-clock measure is also inappropriate here since the total data volume grows with dimensionality. For these reasons, the measure we chose for this experiment is the same as that used for the R*-Tree in Section 4.2. Specifically, we measured the number of pages visited as a percentage of the number of pages necessary for the *Scan* algorithm. Further, the number of bits in approximations was varied to be proportional to the volume of the data set, thus ensuring that the ratio $\frac{\text{index-size}}{\text{database-size}}$ is constant.

The result is shown in Figure 10. The major result of this paper is that the performance of the

VA-File method does not degrade, and even improves with increased dimensionality. This result can be compared directly with that used in Figure 7 for existing, data-partitioning index methods. For low-dimensional spaces, data-partitioning methods are to be preferred. However, for high-dimensional spaces, the VA-File offers significant advantages over either data-partitioning or scan-based methods. Thus, the VA-File approach overcomes the dimensional curse.

5 Discussion

The simple, flat structure of the VA-File offers a number of important advantages, independently from the performance issues discussed in the previous section. It is possible to integrate search in VA-File structures over multiple vector spaces, and also over signature files for non-metric spaces. Such searches are necessary to support conjunctive queries, which are frequent in multimedia databases. It is straight-forward to include weights in Equation 5, thereby accommodating weighted search and, as a special case, partial match. Consequently, improved queries based on the feedback of users are possible and may lead to a better answer set. Data-partitioning index methods as R*-trees lack completely of this facility. Moreover, parallelization, distribution and transaction management are simplified by the flat VA-File structure.

In general, update operations for the array structure of a VA-File is unproblematic, although performance might degrade if too much fragmentation is allowed to occur. A detail which requires specific attention, however, concerns the outer partition points $p_j[0]$ and $p_j[2^{b_j}]$ in each dimension j. These are the upper and lower bounds on that dimension. Consequently, the invariant of Equation 3 is invalidated if a new vector is inserted with a component outwith these bounds. Fortunately, no re-computation of the VA-File is necessary in this case. Rather, it suffices to update the partition points such that they enclose also the new points.

A more serious issue concerns the maintenance of the partition points. In particular, partition points are computed statically. This implies that the method is appropriate for static databases, or for dynamic databases for which data's distribution is relatively static. These properties hold for many multimedia databases, however they cannot always be taken for granted. If partition points become too out-of-date, it can become necessary to derive new partitions, and regenerate the VA-File accordingly. The simplest approach to this is to sort all the components in each dimension, and select the partition points from the sorted list. Although accurate, this can become expensive, and required around 15 minutes for our experiments with large databases. This approach is infeasible for large, highly-dynamic databases. Alternatively, new partition points might be computed based on stochastic sampling of the data, or on a known data distribution. In comparison to the cost of establishing partition points, the cost of regenerating the VA-File itself is negligible.

Many open questions remain. With respect to the structure itself, we would like to better understand how the number of bits should be selected, and also (as in multi-key hashing) how those bits should be allocated to dimensions. Preliminary experiments in this respect have not shown significant improvements, and hence are omitted here. Also, although data-partitioning methods typically perform poorly at high dimensionality, the VA-File might still benefit from clustering. For example, one could consider using a data-partitioning method simply to improve clustering of the primary vector

data, and then building the VA-File over the clustered data. This might work well for stable data sets. The benefit of this approach accrue from the smaller number of pages loaded due to the higher possibility, that two visited vectors lie on the same page. Further, as with signature methods, improved (hierarchical) organizations of the VA-File itself can be considered. Especially, the combination of, say, M-tree [CZR96] and VA-File, where the M-tree is used for the hierarchical organization and the VA-File establishes the search in leaf nodes, reduces the amount of data to be observed by the search algorithm and defines a good partition of data for parallelization and distribution. Further, we would like to better understand how feedback of users and weighted search could improve the quality of the answer set. Other practical areas for future work include integration with our prototype database system CONCERT [BRS96], experimentation with distribution and parallelization, and integration with (distributed) signature methods for similarity search [Sch96].

6 Conclusion

We have motivated and described the VA-File, an approximation-based index method for high-dimensional vector spaces. Such search spaces underlie many similarity measures in multimedia database. The main contribution of this method is that the VA-File overcomes the dimensional curse, which plagues existing data-partitioning index methods. To the best of our knowledge, this structure has not been described previously.

We have provided a performance evaluation for the VA-File method. Experimental results based on both selectivity and wall-clock experiments are promising. For large databases, the VA-File outperforms a well-tuned scan (local, raw disk partition) by a factor of up to four. We have also shown that performance does not degrade, and even improves slightly with dimensionality. Further advantages accrue from the simple, flat structure of the VA-File. It is possible to integrate search in VA-File structures over multiple vector spaces, and also over signature files for non-metric spaces. Such searches are necessary to support conjunctive queries, which are frequent in multimedia databases. Further, both the VA-File and signature methods are relatively straight-forward to parallelize and distribute.

Acknowledgments. This work has been partially funded in the framework of the European ESPRIT project HERMES (project no. 9141) by the Swiss *Bundesamt für Bildung und Wissenschaft* (BBW, grant no. 93.0135), and partially by the ETH cooperative project on *Integrated Image Analysis and Retrieval*. Special thanks are due to Alexander Dimai and Markus Stricker of the Image Science Group at ETH for their help in preparing the vector data which formed the basis of our practical evaluation. Figure 6 is also courtesy of Alex. The vector data was derived from an image database provided by Virginia E. Ogle of the Chabot project. The R*-Tree implementation we used was Daniel Green's implementation, which is freely available over WWW.

References

[BAK96] Stefan Berchtold, Daniel A.Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. In *vldb*, pages 28–39, 1996.

[BF79] Jon Louis Bentley and Jorome H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, USA, 1990.

[Bri95] Sergey Brin. Near-neighbor search in large metric spaces. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 574–584, Zurich, Switzerland, 1995.

[BRS96] Stephen Blott, Lukas Relly, and Hans-Jörg Schek. An open abstract-object storage system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 330–340, Montreal, Canada, June 1996.

[Chi94] Tzciker Chiueh. Content-based image retrieval. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 582–593, Santiago, Chile, September 1994. Morgan Kaufmann.

[CHP+95] M.J. Carey, L.M. Haas, P.M.Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the Workshop Research Issues in Data Engineering–Distributed Object Management (RIDE-DOM95)*, Taipei, Taiwan, March 1995.

[CZR96] P. Ciaccia, P. Zezula, and F. Rabitti. A data structure for similarity search in multimedia databases. In *9th ERCIM Database Research Group Workshop on Multimedia Databases*, Darmstadt, Germany, March 1996.

[ELS95] G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A modified hB-tree supporting concurrency, recovery and node consolidation. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 551–561, Zurich, Switzerland, September 1995.

[Fag96] Ronald Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 216–226, Montreal, Canada, June 1996.

[Fal85] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, March 1985.

[FC87] Christos Faloutsos and Stavros Christodoulakis. Description and performance analysis of signature file methods for office filing. *ACM Transactions on Office Information Systems*, 5(3):237–257, July 1987.

[FEF+94] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, July 1994.

[FL95] Christos Faloutsos and King-Ip Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 163–174, San Jose, California, 22–25 May 1995. *SIGMOD Record* 24(2), June 1995.

[Fli95] M. Flickner *et al.* Query by image and video context: The QBIC system. *IEEE Computer*, pages 23–32, September 1995.

[Fuk90] Keinosuke Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, 2nd edition, 1990.

[Gut84] A. Guttmann. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1984.

[HS95] Gísli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Proceedings of the Fourth International Symposium on Advances in Spatial Database Systems (SSD95)*, number 951 in Lecture Notes in Computer Science, pages 83–95, Portland, Maine, August 1995. Springer Verlag.

[Jag91] H.V. Jagadish. A retrieval technique for similar shapes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 208–217, May 1991.

[KF92] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. Technical Report CS-TR-2820, University of Maryland Institute for Advanced Computer Studies Dept. of Computer Science, Univ. of Maryland, College Park, MD, January 6 1992.

[LJF94] King-Ip Lin, H.V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *The VLDB Journal: The International Journal on Very Large Data Bases*, 3(4):517–542, October 1994.

[Lom90] David B. Lomet. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[MG95] Rajiv Mehrotra and James Gary. Feature-index-based similar shape retrieval. In Stefano Spaccapietra *et al*, editor, *Proceedings of the Third IFIP 2.6 Working Conference on Visual Database Systems (VDB-3)*, Lausanne, Switzerland, March 1995.

[NBE+93] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, and P. Yanker. The QBIC project: Querying images by content using color, texture and shape. In *Storage and Retrieval for Image and Video Databases (SPIE)*, San Jose, CA, February 1993.

[NHS84] Jürg Nievergelt, Hans Hinterberger, and K.C. Sevcik. The Grid File: An adaptive, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[PF96] Euripides G. M. Petrakis and Christos Faloutsos. Fast retrieval by image content in an image database. *IEEE Transactions on Knowledge and Data Engineering*, 1996. To appear.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[Pra91] William K. Pratt. *Digital Image Processing.* John Wiley and Sons, Inc., New York NY, USA, 2nd edition, 1991.

[Sch96] Christian Schraner. Signaturmanager für Ähnlichkeitssuche. Diplomarbeit (Masters Thesis, in German), Institute of Information Systems, ETH, Zurich, August 1996. English title: Signature Manager for Similarity Search.

[Sed83] Robert Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, 1983.

[SO95] Markus Stricker and Markus Orengo. Similarity of color images. In *Storage and Retrieval for Image and Video Databases (SPIE)*, San Jose, CA, 1995.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloustos. The R+ -tree: A dynamic index for multi-dimensional objects. In *vldb*, pages 507–518, Brighton, England, 1987.

[Str80] Gilbert Strang. *Linear Algebra and its Applications*. Academic Press, 2nd edition, 1980.

[Ull88] Jeffrey D. Ullman. *Pronciples of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.