Comparing the performance of two CBIRS indexing schemes

Wolfgang Müller, Günter Robbert, Andreas Henrich
Department for Applied Computer Science, Universität Bayreuth

ABSTRACT

Content based image retrieval (CBIR) as it is known today has to deal with a number of challenges. Quickly summarized, the main challenges are firstly, to bridge the semantic gap between high–level concepts and low–level features using feedback, secondly to provide performance under adverse conditions. High-dimensional spaces, as well as a demanding machine learning task make the right way of indexing an important issue.

When indexing multimedia data, most groups opt for extraction of high-dimensional feature vectors from the data, followed by dimensionality reduction like PCA (Principal Components Analysis) or LSI (Latent Semantic Indexing). The resulting vectors are indexed using spatial indexing structures such as kd-trees[11] or R-trees,[1] for example.

Other projects, such as MARS[19] and Viper[23] propose the adaptation of text indexing techniques, notably the inverted file. Here, the Viper system is the most direct adaptation of text retrieval techniques to quantized vectors. However, while the Viper query engine provides decent performance together with impressive user-feedback behavior, as well as the possibility for easy integration of long-term learning algorithms, and support for potentially infinite feature vectors, there has been no comparison of vector-based methods and inverted-file-based methods under similar conditions.

In this publication, we compare a CBIR query engine that uses inverted files (Bothrops, a rewrite of the Viper query engine based on a relational database), and a CBIR query engine based on LSD (Local Split Decision) trees for spatial indexing using the same feature sets.

The Benchathlon initiative works on providing a set of images and ground truth for simulating image queries by example and corresponding user feedback. When performing the Benchathlon benchmark on a CBIR system (the System Under Test, SUT), a benchmarking harness connects over internet to the SUT, performing a number of queries using an agreed-upon protocol, the multimedia retrieval markup language (MRML). Using this benchmark one can measure the quality of retrieval, as well as the overall (speed) performance of the benchmarked system.

Our Benchmarks will draw on the Benchathlon's work for documenting the retrieval performance of both inverted file-based and LSD tree based techniques. However, in addition to these results, we will present statistics, that can be obtained only inside the system under test. These statistics will include the number of complex mathematical operations, as well as the amount of data that has to be read from disk during operation of a query.

1. INTRODUCTION

For building a useful Content-Based Image Retrieval System (CBIRS), its designers have to find a convincing tradeoff between *speed* and *quality*, as well as *flexibility*.

While the need for quality is obvious, speed becomes necessary and non-trivial as we need to provide high-quality solutions in interactive time[14]: using a CBIRS is still a largely exploratory process. This constraint limits the complexity of operations that can be performed during search, and it also limits the complexity of operations that can be given to each image during the feature extraction and indexing step.

The current CBIR research mainstream (as described for example in[21]) solves the problem by a three step process.

1. Extract one or several *feature vectors* from each image,
2. reduce (if necessary) the dimensionality of each vector stored using methods like Karhunen-Loeve Transform[10] or Latent Semantic Indexing,[4] and

Internet Imaging IV, Simone Santini, Raimondo Schettini, Editors, Proceedings of SPIE-IS&T Electronic Imaging, SPIE Vol. 5018 (2003) © 2003 SPIE-IS&T · 0277-786X/03/$15.00

3. index the resulting data using spatial indexing structures like vp-trees,[27] R-trees[1] to name a few.

However, there are systems that do not follow this mainstream. In[25] Weber *et al.* did a study that is little known in CBIR circles, although it has quite an impact: according to their study, starting at a dimensionality of more than 9, spatial indexing structures do not give any advantage over a full table scan, *i.e.* using no indexing structure at all. As they assume they cannot improve performance beyond linearity in the number of indexed images, they work on improving the constant factor in the complexity $c \cdot n$. They obtain a small c using approximations: *vector approximation* files (VA-files). We will make this important data structure part of a forthcoming comparison.

A research direction that we want to investigate here is the adaptation of information retrieval (IR) techniques. In IR, the inverted file is the best known indexing structure: Inverted Files contain a list of lists: each sub-list i is associated to a feature (*i.e.* word) φ_i. It contains identifiers for all documents that contain φ_i. Weighting methods in IR are based on reading feature lists that correspond to each feature that is part of the query (see[24] for a theoretical justification). Given that a query is usually much smaller than the list of *possible* features that could be part of the query, there is a huge gain of efficiency. However, seen from up close, the case is similar to that of VA-files: we have time complexity roughly *linear*, $c \cdot n$ in the number of documents n in the collection, and the efficiency gain consists in obtaining a small constant factor c.

MARS and *Viper*[19, 23] are both based on inverted-file search. Their main difference lies in the adaptation of an indexing method for *discrete* data to real-valued data. The authors of both systems emphasize the retrieval performance for feedback queries, *i.e.* queries by example where the user submits multiple positive and negative examples.

This brings us to the main motivation of this paper: neither MARS nor *Viper* indexing structures have been tested against non-inverted file systems, *i.e.* to our best knowledge we are not aware of a performance test that compared retrieval performance of IR-specific distance measures applied to a given real-valued vector feature set. Neither are we aware of a *relative* efficiency test with respect to a given spatial indexing structure.

Benchmarking image retrieval systems is a hard problem, mainly because sufficient ground truth data are hard to obtain. Collecting such relevance data for flexible use is one of the harder tasks of the Benchathlon project.[2] As this shortage of ground truth has not been cured yet, parts of this paper are about how to combine real ground truth data with synthesized ground truth data to obtain useful results. In particular, we use real ground truth (obtained for a small collection) for measuring retrieval performance (*i.e.* quality of retrieval results), and synthesized ground truth (for a much larger collection) for measuring retrieval effectiveness (how does the system adapt to a user) in the context of partial evaluation. We feel that –in their combination– these measures allow a useful evaluation of an image retrieval system.

The paper is laid out as follows: in section 2 we will give a more thorough description of inverted files and their use in CBIR. In section 3 we will describe shortly the inner workings of LSD trees, before proceeding in section 4 on how we want to compare LSD-trees with inverted files.

2. INVERTED FILES IN CBIR

As said above, Content-based image retrieval systems store images by extracting features from images, and then make them searchable using an appropriate indexing structure, using a three step process: 1) extract one or several *feature vectors* from each image, 2) reduce (if necessary) the dimensionality of each vector stored using methods like Karhunen-Loeve Transform[10] or Latent Semantic Indexing,[4] and 3) index the resulting data using spatial indexing structures like vp-trees,[27] R-trees[1] to name a few.

Usually the steps two and three become hard, as the dimensionality of an image is very high ($\#pixels \times 3\frac{RGB\text{-}Values}{pixel}$ dimensions, to be precise) which has to be reduced to a much smaller dimensionality using feature extraction. Yet the dimensionality ($\mathcal{O}(100)$) of such a feature set is at least an order of magnitude above the number of dimensions suitable for spatial indexing ($\mathcal{O}(10)$). As is known in the LSI literature, finding a suitable level of dimensionality reduction is a problem in itself.

Both MARS and *Viper* were motivated by these difficulties. In Information Retrieval (IR), texts consisting of thousands of different words (*i.e.* features) are indexed in inverted files. In contrast to classical CBIR systems, in IR the number of words is typically not determined in advance. So, the data structures used in IR have to accomodate changes in the number of dimensionality when the collection changes. The other strong point for the use of IR algorithms is the fact that relevance feedback has been a part of IR research for a long time. In relevance feedback, the system learns from the feedback about the relevance of query result in order to expand and improve the query.

The classical IR model that served as inspiration for *Viper* is remarkably simple: for each feature (we will use the terms *word*, *term* and *feature* interchangeably), one maintains a list of all the documents that contain the said feature, storing also the *term frequency (tf)*, *i.e.* the number of times the feature occurs in the document. For each feature, we store the *document frequency (df)*, *i.e.* the probability that a document drawn at random from the collection contains a given feature.

On querying, the score for a given document D given the query Q is determined by calculating a weighted (by f) sum over the features φ contained both in the document and in the query.

$$S(D|Q) = \sum_{\varphi \in D \cap Q} f(\varphi|D, Q),$$

For the weighting function f, we use the well-known tf.idf weighting function:

$$f(\varphi|D, Q) = tf(\varphi, Q) \times \log \frac{1}{df(\varphi, Collection)}$$

As we only need to consider scores of documents that contain a given φ, we have only to read the feature lists contained in the inverted file for each $\varphi \in Q$ (that, by construction, contain all the documents D with $\varphi \in D$). The number of inverted file lists we have to evaluate is typically much smaller than the number of inverted file lists present in the indexed collection. This is the reason for the time savings gained by the use of inverted files. Fig. 1 gives more insight in the complexity reduction possible by the use of inverted files.

```
for D in Collection:                    for D in Collection:
    score(D)=0                              score(D)=0
    for phi in Q:                       for phi in Q:
       if phi in D:                         for D in InvertedFileList(phi):
          score(D)=score(D)+f(D,phi)            score(D)=score(D)+f(D,phi)
```

Figure 1. Algorithm for a straightforward full scan of documents (*left*) and for inverted-file based querying of a collection (*right*).

The left "naïve" loop needs to perform an intersection of D and Q for *all* Documents (images) of the collection. However, each image is only looked at once (small number of lookups). The complexity is approximately:

$$c_{naive}(Q, Collection) = \mathcal{O}(c_{disk} \sum_{D \in Collection} |D|)$$

When querying the inverted file (right), the complexity becomes in the approximately

$$\mathcal{O}(c_{disk} \sum_{\varphi \in Q} |InvertedFileList(\varphi)|) \approx \underbrace{\frac{|Q| \cdot \overline{(|InvertedFileList(\varphi)|)}}{\overline{(|D|)} \cdot |Collection|}}_{(*)} \cdot c_{naive}(Q, Collection) \underbrace{\cdot (1 + \frac{c_{lookup}}{c_{disk}})}_{(**)}$$

With c_{lookup} being the cost of a hash lookup in memory, and c_{disk} the cost of reading a tuple from disk, and $\overline{(x)}$ denoting the average over all x. The factor $\cdot(1 + \frac{c_{lookup}}{c_{disk}})$ accounts for the fact that during inverted file retrieval, whenever an element of an inverted file list is visited the corresponding document has to be looked up in a hash table for increasing its score. Usually $\frac{c_{lookup}}{c_{disk}}$ is small, and we can assume the time passed on retrieval as almost proportional to the amount of data read, *i.e.* the number of inverted file entries read.

The factor $(*)$ tells us what are the primary goals in optimizing a feature set for indexing in inverted files: in order to optimize the efficiency gain with respect to a full scan of all the documents in the collection, the inverted file lists need to be short, *i.e.* the expected document frequency should be low for each feature. Secondly, each document should be represented only by few features out of the possible features ($\Phi_{Collection}$):

$$|Q| \ll \Phi_{Collection}$$

Viper's features were chosen with this goal in mind, in the following section we will describe *Viper*'s color features.

2.1. *Viper*'s color features

The idea pursued in *Viper* is to partition the image into a quad-tree hierarchy of blocks ($4 + 16 + 64 + 256 = 340$ blocks). The colors are quantized in $18 \times 3 \times 3 = 162$ HSV bins plus 4 gray values. Blocks and colors are labelled, and for each block the most frequent color is stored as an integer. Thus each image has 340 color features. Each block is represented only by one feature out of 162 possibilities, so on average over all possible features and all documents a document will share a given feature with less than one percent of the whole image collection.

As these features are highly selective, *Viper* offers also histogram-based retrieval where instead of a *tf.idf* weighting *idf*-weighted histogram intersection is used, that is we use the weighting function

$$f(\varphi|D, Q) = \min(tf(\varphi, Q), tf(\varphi, D)) \cdot \log \frac{1}{df(\varphi, Collection)}$$

where $df(\varphi, Collection)$ is defined as the fraction of documents that have at least *one pixel* that of the color associated with φ.

2.2. Pruning: gaining speed by partial evaluation

The above describes complete evaluation. However, the retrieval complexity can be further reduced by evaluating *just a part* of the query.[22] There are two approaches towards pruning:

1. Minimizing the factor $(*)$, *i.e.* minimizing the number of tuples that have to be read by the system while processing the queries. In H. Müller *et al.*'s publication,[22] the number of tuples read is reduced by sorting query terms by their weight, and reading only the inverted file lists for the highest weighted terms.
2. Minimizing the factor $(**)$, *i.e.* minimizing the number of documents for which a score is maintained. H. Müller *et al.* reduce the number of score board items in a stepwise process during the query.

3. LOCAL SPLIT DECISION TREES (LSD$^{\mathrm{H}}$-TREES)

In the following we will first describe the LSD^h-tree, as a spatial low-dimensional access structure. This comprises the basic concepts (section 3.1) and the algorithm for nearest neighbor queries (section 3.2). Thereafter we present the special split strategy we use for high-dimensional data (section 3.3). For the details of this access structure we refer to.[8]

3.1. Basic Idea of the Spatial LSD-tree

As usual for access structures which support spatial access to point objects the LSD-tree divides the data space into pairwise disjoint data cells. With every data cell a bucket of fixed size is associated, which stores all objects contained in the cell. In this context a data cell is often called *bucket region.*

Let us consider the creation of a two-dimensional LSD-tree. We assume that a bucket can hold two points. Initially, the whole data space corresponds to one bucket, say bucket 1. After the insertions of two objects, (60,30) and (20,80) bucket 1 has been filled, and an attempt to insert a third object (15,25) causes the need for a bucket split. To this purpose, a *split line* is determined. For example the split can be performed in dimension 1 at position 40. The objects on the left side of the split line remain in bucket 1, while those on the right side are stored in a new bucket, bucket 2. The split is represented by a directory node containing the split dimension and the split position. Thereafter the new object (15,25) can be stored in bucket 1. With the next insertion (30,45), we again achieve an overflow in bucket 1. Again the bucket is split into two, and the split decision is represented in the directory tree by a new node. This process is repeated each time the capacity of a bucket is exceeded. Usually, the directory grows up to a point where it cannot be kept in main memory any longer. In this case, subtrees of the directory are stored on secondary memory, whereas the part of the directory near the root remains in main memory. For the details of the paging algorithm we refer to.[7]

3.2. Nearest Neighbor Queries

The following algorithm efficiently solves the m-nearest neighbor problem where we look for the m points in the structure located closest to a given query point. Since the details of this algorithm can be found in[5] we restrict ourselves to a terse description here.

To describe the processing of a m-nearest neighbor query, we have to introduce the term of a *data region*: The *data region* of a bucket is its bucket region, and the *data region* of a directory node is defined as the union of the two data regions of its sons.

In principle the m-nearest neighbor algorithm works as follows:

We start at the root of the directory and search for the bucket for which the bucket region contains the starting point p of the query. Every time during this search when we follow the left son, we insert the right son into an auxiliary data structure NPQ (= node priority queue), where the element with the smallest distance between p and its data region has highest priority, and every time we follow the right son, we insert the left son into NPQ. Then the objects in the found bucket are inserted into another auxiliary data structure OPQ (= object priority queue), where the object with the smallest distance to p has highest priority. Thereafter the objects with a distance to p less than or equal to the minimal distance between p and the data region of the first element in NPQ are taken from OPQ. These objects are already sorted correctly with respect to their distance to p. They represent the closest neighbors. Now the directory node or bucket with highest priority is taken from NPQ. If it happens to be a directory node, a new search is started choosing always the son on the side of the split line facing p. The other son is inserted into NPQ. The bucket determined in this way is processed inserting the objects stored in this bucket into OPQ and removing the objects with a distance to p less than or equal to the minimal distance between p and the first element in NPQ from OPQ. Thereafter the process is continued in the same way until either m objects are taken from OPQ or NPQ and OPQ are empty — which means that m is larger than the number of objects in the access structure.

3.3. Split Strategy

An important part of the insertion algorithm of the LSD-tree is the split strategy which determines the split position and the split dimension in case of a bucket split. In[6] we distinguished two types of split strategies: *Data dependent split strategies* depend on the objects stored in the bucket to be split. An example is to use the average of all object coordinates with respect to the split dimension. *Distribution dependent split strategies* choose the split dimension and the split position independently of the actual objects stored in the bucket to be split. An example based on the assumption of a uniform distribution is to split a cell into two cells of equal areas.

Since data dependent split strategies calculate the split line based on the objects actually stored in the access structure, on the one hand, they yield a good adaptation of the data space partition. On the other hand, the data

space partition depends on the order of insertion and degenerates especially if the objects are inserted in sorted order. However, in accordance with the results presented in[6] we observed that the risk for a degeneration of the directory tree caused by pre-sorted data decreases for higher dimensions. On the other hand, high dimensional data is usually more skewly distributed. Therefore we propose to use the average over the coordinate values of the objects in the bucket to be split for the determination of the split position when concerned with feature vectors. With respect to the split dimension we have to consider the fact that feature vectors usually contain features with only few potential feature values. Therefore we propose the following split strategy:

Assume we have to maintain t-dimensional feature vectors and the dimensions are numbered from 0 to $t-1$. Let d_{old} be the split dimension used in the node referencing the bucket to be split. Set i to 1. If there are different feature values for dimension $(d_{old}+i) \bmod t$ in the bucket to be split, use dimension $(d_{old}+i) \bmod t$ as the split dimension. Otherwise increase i by 1 until there are different feature values for dimension $(d_{old}+i) \bmod t$ in the bucket to be split. This strategy assures that a split dimension is used only if there are different feature values in this dimension in the bucket which has to be split.

For very high dimensional data the described split strategy may prefer lower dimensions. Therefore it can be extended in a way that the dimension with the highest variance for the object coordinates in the bucket to be split is used as the split dimension — this directly corresponds to the strategy applied with the VAMSplit R-tree.[26]

4. WHAT WE MEASURE

Within the previous two sections, we have elaborated on the strengths and weaknesses of the data structures involved. Within this section, we will elaborate on what we want to learn within our performance experiments*.

A list of interesting problems includes:

- Inverted files
  - Quantizing a continuous-valued feature set
    - Efficiency improvements and result quality degradation due to quantization
    - Influence of $tf \cdot idf$ weighting on the result quality
  - Full-precision histograms
    - Efficiency improvements and result quality degradation due to partial evaluation
- LSD-trees
  - Influence of the curse of dimensionality for a large real world data set.
  - Efficiency improvements and result quality degradation when doing only partial evaluation of next-neighbor lists.
  - Query result improvements on relevance feedback
  - Influence of the distance measure on the efficiency of the LSD-tree structure

Within this publication, we concentrate on comparing performance of an LSD-tree that indexes dimensionality-reduced vectors in an euclidean space to pruned histogram intersection using inverted files. An evaluation of LSD-trees with partial evaluation of next-neighbor lists has been presented in.[9]

*Within the following enumeration, we will discern between *(query) performance, i.e.* the quality of results and the *(query) complexity, i.e.* some weighted number of operations needed for obtaining the query result

4.1. Feature sets used

For our experiments, we decided to use a simple, real-world feature set which is simple to produce in many variations. Color histograms meet these requirements. Krishnamachari *et al.*[12] give a thorough description of MPEG-7 color features. They describe the reduction of color histograms down to a 64-bit bitvector, to be compared based on the Hamming distance.

In their work, Krishnamachari *et al.* extract 64-bin histograms from images. Each histogram is interpreted as an 64-dimensional real-valued vector that is subjected to a Haar wavelet transform. The resulting 63 coefficients are quantized in a *binary* fashion: positive coefficents are represented by a 1 in the bit vector, negative or zero coefficients are represented by a 0.

While this distance measure is most efficiently employed by storing the bit vectors for all images in a flat file that is read in a full table scan, the way the bitvectors are obtained serve as inspiration in this publication.

As base for our experiments we partitioned the HSV color space in $18 \times 3 \times 3 = 162$ bins. We then reduced the dimensionality using PCA to 8, 16, or 64 dimensions respectively. These data were indexed using an LSD tree. Similarly, we used bothrops for evaluating only the 5%, 20% highest-valued histogram components.

As both the LSD-tree and bothrops were integrated into a common framework that sacrifices some efficiency for the sake of genericity, we felt that performance measures using implementation-independent measures were most appropriate. In particular, we chose:

Number of bytes read from disk: As we want to investigate methods usable for large collections, we assume that a large part of the index resides on disk, and only small, and frequently-used parts reside in main memory. In particular, we assume that the inverted file and LSD-tree indexes reside on disk and measured how much data has to be read during each query process.

Hash accesses: Inverted file scoring needs to access a scoring hash table that contains all documents as well as their scores. LSD-trees need to access the auxiliary priority queues (OPQ and NPQ) when looking for near neighbours to a given element. During our measurements we counted these accesses.

Floating point multiplications: for comparing distances between images, as well as for scoring, floating point multiplications have to be performed. As previous experiments with *Viper* suggest, the number of such multiplications is non-negligible.

In addition to that, we measured the raw query time, used within bothrops and the LSD-tree, ignoring the time spent in the framework.

4.2. Query scenarios and data collections

Much of the usefulness of a quality/efficiency performance test of CBIR systems depends on a suitable choice of the collection (for testing the efficiency) as well as quality measures and ground truth.

4.2.1. Collection size:

While we regard it as understood that within this paper we are treating the case of an unconstrained photo collection, the question of collection size merits further discussion: Specialized spatial indexing structures aim at a search time that is logarithmic in the number of items (*i.e.* feature vectors) in the collection, an inverted file's search time will be proportional in the number of indexed feature vectors. For a fair test, we will have to choose a reasonably large data collection.

4.2.2. Ground truth:

A precondition for measuring the result quality obtainable using a CBIRS is some definition about the retrieval result we want to achieve. Definitions of ground truth differ widely, and are mostly implicit in the testing scenarios used in the literature. Some of them include:

Test queries by hand: some test users scan the test collection by hand, providing relevance ratings for each image of the collection with respect to a number of test query images.[14] Due to the fact that each test user scans all the collection, there are multiple, partial conflicting relevance ratings. A system will be best that is able to learn all of these ratings.

Known clusters from still images: stock photo collections usually come in clusters (or sub-collections) of fixed size (100 images). Many publications use these clusters as ground truth for performance measures, *e.g.* Cox *et al.* and Lee *et al.*.[3, 13] An advantage of this method is the abundance of relevance data, however it is not taken into account that there is some unintended overlap between clusters. In some rare cases, identical images are part of two clusters. More frequently, images could be semantically part of two or more clusters. For example, an image of *Pont Neuf* in Paris could well be part of the "Paris" and the "Bridges" section of the collection.

Known clusters from videos: images from one shot are considered as a cluster. Potential problems here are the overlap and the potentially *very* high similarity between images of one shot.

Annotation as user simulation: While in an ideal world, evaluating test queries by hand is probably the most exact method, it is less so because it is very costly, and thus cannot be applied on large collections and large numbers of queries: for n_q test queries, n_u ground truth gathering users and $|Collection|$ images in the collection, we need to perform $n_u \cdot n_q \cdot |Collection|$ checks for relevance. This number of checks quickly becomes prohibitive. For example, the relevance data used in Müller *et al.*'s publication[14] needed $14 \cdot 3 \cdot 2500 = 105000$ relevance checks. Still, this data set is considered tiny by many scientists. One way of countering this shortcoming is to annotate each image of the collection, in a way that the annotation can be used as a base for the *simulation* of ground truth (for example in[15]). While annotating an image takes more time than a simple relevance judgement, each image needs to be annotated only once, and from an annotated collection ground truth for $|Collection|$ example queries can be derived.

While we consider annotation a valid method for simulating user feedback, unfortunately at the time of writing, the annotation effort within the Benchathlon has barely begun. As we have access to the *Viper* $14 \cdot 3 \cdot 2500$ ground truth collection, and to the free, yet unannotated Benchathlon image collection,[2] we decided to derive our results from both collections.

We used the *Viper* ground truth collection to get a hint on the actual retrieval quality obtainable with the systems compared.

For efficiency-related experiments, we used the Benchathlon image collection. For measuring, how partial evaluation, as well as reduction of dimensionality influences the result, we evaluated the retrieval performance with respect to a full dimensionality full evaluation of the query.

4.2.3. Quality measures

Not wanting to preclude the choices of the ongoing Benchathlon effort and the work on a Benchathlon benchmarking harness, we decided to choose a widely known and widely used quality measure from the literature, the precision recall graph.

Precision/recall The result of a query is a sequence of images: $r_1, \ldots, r_{Collection}$ We define R_i, $i < |Collection|$ as the set of retrieved images, up to the ith result: $R(i) := \{r_1, \ldots, r_i\}$. Be REL_q the set of images relevant with respect to a query q. Then we can define precision as $precision(i) := \frac{|R(i) \cap REL_q|}{i}$ *i.e.* the fraction of documents that have been retrieved among the first i elements and that are relevant to the query.

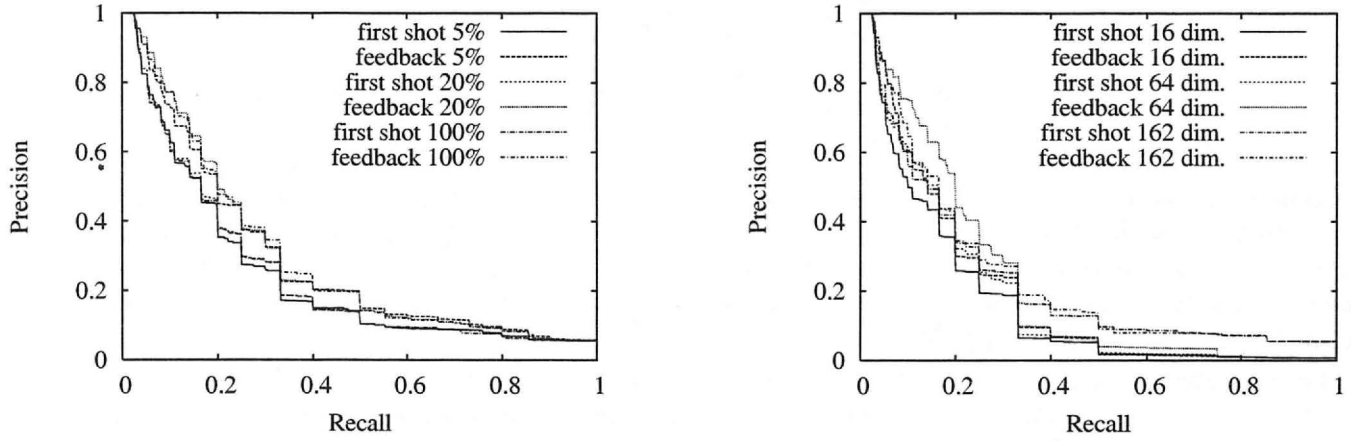

Figure 2. *Experiments with* bothrops *(left) and the LSD-tree (right), using* Viper *relevance data (2500 images, and 14 queries with user data for 3 users). In each image, we compare first-shot performance with feedback performance,* with complete (100% features/ 162 dimensions) and partial evaluation, respectively. Note that one can see two groups of curves in each graph, before (worse results) and after feedback (better results).

The recall is the fraction of documents that have been retrieved and are relevant, divided by the documents that are relevant and part of the collection. $recall(i) := \frac{R(i) \cap REL_q}{|REL_q|}$. A precision-recall graph consists of the dots $(recall(i), precision(i))$ for $i \in 1, \ldots, |Collection|$.

5. EXPERIMENTAL SETUP

5.1. The GNU Image Finding Tool

The GNU Image Finding tool[18] is a modular system for the test and use of CBIR query engines. New query engines can be added as plugins, as described in more detail in W. Müller's thesis. The plugin mechanism was used in order to make both LSD trees and inverted files accessible for use by an MRML-compliant benchmarking harness.

5.2. SnakeMeter: an MRML compliant benchmarking harness

As benchmarking harness, we used SnakeMeter, which has already been used for other publications.[15,18] SnakeMeter is a modular system that uses MRML[16,17] for communicating with the system under test. When benchmarking, queries are issued by SnakeMeter, and raw query results coming from the system under test are gathered in a relational database. SnakeMeter then generates records in a second step, a visit of the database.

6. RESULTS

As it was said before, our experiments were performed on two image collections. Firstly, the a 2500 image collection provided by the *Télévision Suisse Romande* (TSR2500), 10090 public domain images provided by the Benchathlon initiative. Fig. 2 shows precision-recall plots using the *Viper* image collection and relevance data. We can see that feedback improves precision on both systems. The color histogram intersection using inverted files obtains better results than the Euclidean distance based LSD-tree. As could be expected, performance degrades when evaluating fewer features.

6.1. TSR2500 collection

What is most interesting in the experiments with inverted files is that even if evaluating just the 5% most probable bins of a given histogram, such a pruned evaluation with feedback is more effective than a one-shot query evaluating 20% or 100% of the histogram. Given that the complexity of the process is proportional to the number of bins evaluated it is thus more efficient to evaluate few bins and give the user the opportunity for feedback, as can be seen from the performance data:

Fraction of features evaluated	5%	20%	100%
Hash accesses (=Multiplications)	30300	90500	413000
Bytes read	272000	992000	4861944
Raw query duration (s)	0.37	0.45	0.8

Furthermore, it is easily seen that bothrops has quite an amount of administrative overhead: while the main parameters that should govern query complexity vary linearly with the number of histogram bins evaluated, this complexity is not reflected by the time spent on a query. A possible, missing parameter would be the number of `seek()`s on disk. Unfortunately, this parameter is difficult to get at in a database setting, futhermore, this parameter is implementation dependent.

The performance results of the LSD tree show an efficient implementation of an efficient data structure. One can see that for the small 2500-image collection query times are between 0.01 and 0.07 seconds on our test system. We see that for obtaining 20 documents from the retrieval result, we have to look at almost the whole collection: the near-complete LSD-file is read (2197 to 2497 images of 2500 images) for all dimensions tested. Yet, when lowering the dimensionality of the vectors indexed, each stored vector becomes smaller, thus decreasing the number of bytes to be read for processing a query:

Dimensions	8	16	64	162
Hash accesses	2197	2351	2414	2497
Multiplications	18032	38591	159044	415036
Bytes read	144530	288231	1055792	2340815
Raw query duration (s)	0.010	0.015	0.028	0.062

6.2. Benchathlon

For these performance experiments, we used 10090 images from the benchathlon collection. Benchathlon images are in the public domain. While large parts of the benchathlon efforts include defining how these images are to be annotated and performing the annotation itself, these images are (at the time of writing) yet unannotated.

In our measurements we queried each system under test 21 times for 20 images, comparing the images retrieved to the first 20 images retrieved without dimensionality reduction or pruning, respectively.

The results for the LSD tree are:

Dimensions	8	16	64	162
Percent retrieved	31	39	50	100
Hash accesses	8468	8438	9577	9535
Multiplications	70000	139000	635000	1595000
Bytes Read	552000	986000	4350000	9283000
Raw query duration (s)	0.06	0.07	0.52	0.66

Concerning efficiency: just now, around 10k images, the LSD-tree starts becoming useful, saving us from reading 15% of the images with respect to a full table scan when using 8 dimensions. We can also see from the raw query duration that the LSD-tree becomes efficient partly due to caching that is performed by the operating system.

Regarding the result quality itself, we see that reducing the dimensionality of the vectors indexed alters strongly the retrieval results. Actually, this is much easier to see than on the precision-recall plots.

Percent of features evaluated	5	20	100
Percent retrieved	70	92	100
Hash accesses	121000	363000	1670000
Bytes Read	1090000	4000000	19600000
Raw query duration	4.9	5.1	5.8

While not reaching the efficiency of LSD-trees, partial histogram intersection using inverted files responds better to partial evaluation than LSD-trees given dimensionality reduction using PCA and Euclidian distance.

7. CONCLUSION

There are two groups of conclusions we can draw from these experiments.

Firstly on the experiments themselves: our experiments have shown that well-implemented high-dimensional access structures are fast even with dimensionality $\gg 10$, *i.e.* under conditions where they have to perform a full table scan. However, experiments also show that our LSD-tree implementation needs to be adapted to distance measures other than the Euclidean distance. Inverted files tend to read much data, yet their performance degrades very gracefully when doing only partial evaluation, even when using feature sets that are not specially adapted for inverted files. We plan to investigate further into pruning techniques for fast evaluation of color distance measures with inverted files.

Secondly, regarding methods of benchmarking: Our experiments suggest that the Benchathlon could benefit from using two collections of differring size in order to account for both retrieval effectiveness and speed. This becomes important, as spatial indexing structures aim at a complexity that is sub-linear in the number of documents. In high-dimensional spaces, such complexity can only be achieved in large collections.

REFERENCES

1. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf.*, pages 322–331, Atlantic City, N.J., USA, 1990.
2. The Benchathlon Network.
   URL: `http://www.benchathlon.net`.
3. Ingemar J. Cox, Joumana Ghosn, Matt L. Miller, Thomas V. Papathomas, and Peter N. Yianilos. Hidden annotation in content based image retrieval. In *IEEE Workshop on Content-based Access of Image and Video Libraries (CBAIVL'97)*, pages 76–81, June 1997.
4. Scott Deerwester, Susan T. Dumais, George W. Furnas, T. K. Landauer, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.
5. A. Henrich. A distance-scan algorithm for spatial access structures. In *Proc. 2nd ACM Workshop on Advances in Geographic Information Systems*, pages 136–143, Gaithersburg, Md., USA, 1994.
6. A. Henrich and H.-W. Six. How to split buckets in spatial data structures. In *Proc. Intl. Conf. on Geographic Database Management Systems*, Esprit Basic Research Series DG XIII, pages 212–244, Capri, 1991.
7. A. Henrich, H.-W. Six, and P. Widmayer. The LSD-tree: spatial access to multidimensional point and non point objects. In *Proc. 15th Intl. Conf. on VLDB*, pages 45–53, Amsterdam, 1989.
8. Andreas Henrich. The lsd^{h}-tree: An access structure for feature vectors. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 362–369. IEEE Computer Society, 1998.
9. Andreas Henrich. A Relaxed Algorithm Similarity Queries. In *in Proceedings of the 2nd International Workshop on Multimedia Data Document Engineering (MDDE'2002), March 24,*, number 2490 in LNCS. Springer, March 2002.
10. K. Karhunen. Zur Spektraltheorie stochastischer Prozesse. *Ann. Acad. Sci. Fennicae*, 37, 1946.
11. M.J. van Kreveld and M.H. Overmars. Divided k-d trees. *Algorithmica*, 6:840–858, 1991.
12. Santhana Krishnamachari, Akio Yamada, Mohamed Abdel-Mottaleb, and Eiji Kasutani. Multimedia Content Filtering, Browsing and Matching Using MPEG-7 Compact Color Descriptors. In Robert Laurini, editor, *Proceedings of VISUAL 2000*, number 1929 in Lecture Notes in Computer Science, Lyon, France, November 2–4 2000. Springer-Verlag.
13. Catherine S. Lee, Wei-Ying Ma, and HongJiang Zhang. Information Embedding Based on User's Relevance Feedback for Image Retrieval. In Panchanathan et al..[20] (SPIE Symposium on Voice, Video and Data Communications).
14. Henning Müller, David McG. Squire, Wolfgang Müller, and Thierry Pun. Efficient access methods for content-based image retrieval with inverted files. In Panchanathan et al..[20] (SPIE Symposium on Voice, Video and Data Communications).

15. Wolfgang Müller, Stéphane Marchand-Maillet, Henning Müller, David McG. Squire, and Thierry Pun. Evaluating Image Browsers Using Structured Annotation. *Journal of the American Society for Information Science*, 52(11):961–968, 2001.
16. Wolfgang Müller, Henning Müller, Stéphane Marchand-Maillet, Thierry Pun, David McG. Squire, Zoran Pečenović, Christoph Giess, and Arjen P. de Vries. MRML: A Communication Protocol for Content-Based Image Retrieval. In Robert Laurini, editor, *Fourth International Conference On Visual Information Systems (VISUAL 2000)*, Lecture Notes in Computer Science, Lyon, France, November 2–4 2000. Springer-Verlag.
17. Wolfgang Müller, Zoran Pečenović, Arjen P. de Vries, David McG. Squire, Henning Müller, and Thierry Pun. MRML: Towards an extensible standard for multimedia querying and benchmarking – Draft proposal. Technical Report 99.04, Computer Vision Group, Computing Centre, University of Geneva, rue Général Dufour, 24, CH-1211 Genève, Switzerland, October 1999.
18. Wolfgang T. E. Müller. *Design and implementation of a flexible Content Based Image Retrieval framework: the GNU Image Finding Tool.* PhD thesis, University of Geneva, 24, rue du Général Dufour, CH-1211 Genève, 2001.
19. Michael Ortega, Yong Rui, Kaushik Chakrabarti, Sharad Mehrotra, and Thomas S. Huang. Supporting similarity queries in MARS. In *Proceedings of The Fifth ACM International Multimedia Conference (ACM Multimedia 97)*, pages 403–413, Seattle, WA, USA, November 9–13 1997.
20. Sethuraman Panchanathan, Shih-Fu Chang, and C.-C. Jay Kuo, editors. *Multimedia Storage and Archiving Systems IV (VV02)*, volume 3846 of *SPIE Proceedings*, Boston, Massachusetts, USA, September 20–22 1999. (SPIE Symposium on Voice, Video and Data Communications).
21. Arnold W.M. Smeulders, Marcel Worring, Simone Santini, A. Gupta, and Ramesh Jain. Content based retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(12):1349–1380, 2000.
22. David McG. Squire, Henning Müller, and Wolfgang Müller. Improving response time by search pruning in a content-based image retrieval system, using inverted file techniques. In *IEEE Workshop on Content-based Access of Image and Video Libraries (CBAIVL'99)*, pages 45–49, Fort Collins, Colorado, USA, June 22 1999.
23. David McG. Squire, Wolfgang Müller, Henning Müller, and Jilali Raki. Content-based query of image databases, inspirations from text retrieval: inverted files, frequency-based weights and relevance feedback. In *The 11th Scandinavian Conference on Image Analysis (SCIA'99)*, pages 143–149, Kangerlussuaq, Greenland, June 7–11 1999.
24. C. J. van Rijsbergen. *Information Retrieval.* Butterworths, London, 2nd edition, 1979.
25. Roger Weber, Hans-J. Schek, and Stephen 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 (VLDB)*, New York, USA, August 1998.
26. D.A. White and R. Jain. Similarity indexing: Algorithms and performance. In *Proc. Storage and Retrieval for Image and Video Databases IV (SPIE)*, volume 2670, pages 62–73, San Diego, CA, USA, 1996.
27. Peter Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms*, pages 311–321, Austin, Texas, USA, January25-27 1993.