Efficient and Effective Image Copyright Enforcement

Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, Laurent Amsaleg

Efficient and Effective Image Copyright Enforcement

Herwig Lejsek
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
herwig@ru.is

Friðrik Heiðar Ásmundsson
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
fridrik01@ru.is

Björn Þór Jónsson
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
bjorn@ru.is

Laurent Amsaleg
IRISA–CNRS
Campus de Beaulieu, 35042 Rennes, France
laurent.amsaleg@irisa.fr

Abstract

With the proliferation of high-speed internet access, piracy of multimedia data has developed into a major problem and media distributors, such as photo agencies, are making strong efforts to protect their digital property. Some recent work on image processing has therefore focused on content-based methods to detect image copyright violations, and a "local descriptor" method, which extracts several characteristic points of an image and describes through high-dimensional vectors, has been shown to be quite effective, albeit very inefficient.

We have applied a recent approximate query processing method, the OMEDRANK algorithm, to the image copyright protection method above and shown that it does not result in more efficient query processing without sacrificing the quality of the results. Therefore we have proposed a new index structure, the PvS-index, which segments the descriptor collection based on projections to random lines and utilizes all the nice properties of the OMEDRANK algorithm. In a detailed performance study using a collection of over 20 million image descriptors, we show that using OMEDRANK on top of the PvS-index results in extremely efficient and effective query processing.

1 Introduction

Today, many photo agencies expose their collections on the web for the purpose of selling access to the images. They typically create web pages of thumbnails, from which it is possible to purchase high-resolution images that can be used for professional publications. Enforcing intellectual property rights and fighting against copyright violations is particularly important for these agencies as these images are a key source of revenue.

With the proliferation of high-speed internet access, however, piracy of multimedia data has developed into a major problem. Of course, copyright violations may not always be malicious, as publishers may forget to pay for the rights of one particular image lost in the flow of

pictures they manipulate each day. The most problematic cases, and the ones that induce the largest losses, occur when "pirates" steal the images that are available on the Web and then make money by illegally reselling those images.

Although significant security measures are typically employed, there is always the possibility of skilled hackers thrusting their way into the Web sites and stealing images. It is therefore necessary to *detect* copyright violations. Today, the photo agencies assign specific people to examine pictures published in important newspapers, magazines, and web sites, and try to spot pictures that might originate from their collections. These people depend on their memory and knowledge of the "style" of the photographers they work with. When they identify such an image, they verify whether its publication rights have been paid; if this is not the case, they may issue an enforcement request that may end up in court.

Because it is entirely manual and because an ever-increasing number of images must be controlled, the current copyright enforcement process is very tedious. Building an automatic system enforcing these property rights would reduce the cost of enforcement and improve the violation detection rate. This paper addresses the indexing and search algorithms that must lie at the heart of such an automatic system and proposes a very efficient and effective solution to the problem of automatic copyright enforcement.

1.1 Copyright Enforcement Requirements

An automatic copyright enforcement system should process a stream of incoming images, that are found on the web or scanned and entered by users. The system should search its catalogs for matches and raise alarms when it believes it has found an image for which the publisher is not registered as having purchased that image. These alarms must then be processed by human experts to check whether there is a real violation of copyrights. In order to streamline this process, the system must meet the following four requirements.

First, it has to be *robust* to image modifications, as it is quite common for publishers to touch images; either for publication reasons (typically cropping, scaling and slight color modifications) or to make their piracy less obvious (e.g., line or column removal, small image distortions or other low-level signal desynchronisation).

Second, the system has also to be *effective*. For detection of large scale piracy, it is not mandatory to have 100% detection rate; instead the system should return as few false alarms as possible to save the experts' time, without missing many violations.

Third, the system must be *dynamic*. The most recent images, which are often news-related, have high commercial value and are likely to attract robbers. The system must therefore handle frequent and large updates to the ever-growing image collections.

Finally, it must obviously be very *efficient*, as the time needed to check a particular image must be small. The system must also be scalable as photo agencies have very large image collections to protect and there is a large set of potentially problematic images around.

1.2 Watermarking

Watermarking has been proposed as a solution to the problem of detecting copyright violations. It does, however, have problems meeting the requirements above. For example, water-

mark detection for today's advanced watermarking schemes is generally a very time consuming process.

Watermarking is not a robust enough technique, as it is possible in many cases to wash out watermarks. First, each proposed scheme is so far robust to only one type of image modifications, e.g., methods that are robust to image distortions typically fail on signal desynchronisations. Also, because photographers may sell the same image to many photo agencies, the same image may be found with many different watermarks and even unmodified, and a comparison of these copies will reveal the watermark. Furthermore, when the same type of watermark is inserted in many images, statistical analysis may be used to detect and then remove the watermarks.

Addressing all of these issues would require very advanced schemes that do not exist today. Therefore watermarking by itself is not a sufficient solution to the problem of detecting copyright violations.

1.3 Relying on Visual Similarity

The alternative method is to use content-based image retrieval techniques to detect stolen images and to raise alarms based on their visual similarities. This method has the advantage of being based solely on the image contents, thereby avoiding making any changes to the image itself. In such a system, the photo agency must maintain a local database of the images it possesses to compare outside images to. If an image was indeed stolen and used to create a pirated copy, the system has to identify the original image it believes was used to make the pirated copy.

Traditional image description schemes such as color histograms or corellograms fail to meet the above robustness requirement (due to severe cropping in particular) and their recognition power does not scale well to large collections [12]. A better approach, proposed in [3], is to use a fine-grained image recognition scheme based on the local descriptors devised by Florack et al. [6] for gray-scale images and extended to color images in [1]. With this description scheme, each image yields many descriptors (several hundreds for high-quality images), where each descriptor describes a small "local" area of the image. To retrieve the images that are similar to a query image, a Euclidean k-nearest neighbor query is run for each local descriptor computed on the query image. Each nearest neighbor "votes" for the image it is associated with. Then, the most similar images are found by ranking the images according to their number of votes.

This scheme has been shown to be robust to many different types of image modifications [1, 3]. It is quite insensitive to resizing, color variation, cropping, rotation, jpeg-compression, mirroring, various illumination changes, partial occlusions, etc. The scheme has also been shown to be very effective, as it has a high detection rate and results in very few false alarms [3].

Efficiency, on the other hand, is a problem in this scheme. Although the nearest neighbor query for each query descriptor may be run somewhat efficiently, using multidimensional indexing techniques, the fact that there are hundreds of such query descriptors results in very inefficient execution. It was shown in [1] that advanced multi-dimensional indices, such as the VA-file [13], fail to beat a sequential scan of the whole descriptor collection. As the search process is CPU bound, scanning large collections may take hours or even days, depending on the hardware and the size of the collection.

In [2], a faster retrieval scheme was proposed,

based on pre-clustering the data and running approximate nearest neighbor queries. This scheme, which is similar in spirit to the CLINDEX approach [9], was shown to improve query response time by over 90% without any loss of effectiveness when applied to the problem of copyright enforcement. For very large collections, however, response time savings of 90% over a sequential scan will still result in excessive query evaluation times. The query processing is still CPU bound, as most of the time is spent on distance calculations. Additionally, the process of pre-clustering the data was very time-consuming, taking a full week for a collection of 9544 images (little over 2.5 million descriptors). Clearly this approach will not scale to collections of realistic sizes.

In summary, local descriptors fulfill the requirements for robustness and effectiveness. What is needed, however, is an efficient query processing algorithm that exhibits very small response times despite the many consecutive query descriptors that need to be considered to check the copyright of one particular image.

1.4 Contributions of the Paper

Recently, Fagin et al. proposed in [5] a framework for very efficiently evaluating single descriptor nearest-neighbor queries over high-dimensional collections. This framework is based on projecting the descriptors onto a limited set of random lines. Each random line is used to give a ranking of the database descriptors with respect to the query descriptor. These rankings are then efficiently aggregated to produce a fairly good approximation of the actual Euclidean k-nearest neighbors. The fastest algorithm to aggregate the rankings was called OMEDRANK.

The OMEDRANK algorithm has several nice properties: it is based on a cheap aggregation of rankings instead of a complex distance function; it uses standard B^+-trees to index the data, therefore handling updates gracefully; and it allows for a clever dimensionality reduction, by varying the number of random lines that are indexed. While OMEDRANK performed very well in the experiments of [5], which used rather small collections of global descriptors, it was not clear that it would perform well for large collections of local descriptors. The goal of our work was therefore to study the performance of OMEDRANK with local descriptors, and to find ways to make the performance acceptable for large image collections.

This paper makes three major contributions. First, we analyze the behavior of OMEDRANK in our application and show that it performs poorly despite its nice properties. In a nutshell, while OMEDRANK reads only a small percentage of the whole database for every query descriptor, this turns out to be a significant amount of data when the database is very large and when there are many query descriptors. While the processing time of OMEDRANK can be improved through dimensionality reduction, the quality of the results suffers instead.

Second, we propose a modification to the index creation strategy of OMEDRANK. Our indexing strategy is based on repeatedly segmenting the descriptor collection into overlapping segments and projecting the data in the segments onto new random lines. While the index size grows due to the overlapping segments, our indexing strategy significantly reduces the amount of data that needs to be read for each query descriptor; in fact, we have designed the index to require only 3 disk reads per query descriptor, which results in very efficient query evaluation.

Third, we present a detailed performance

study using a large collection of real descriptor data. Our results show that our proposed indexing scheme results in query execution that is both efficient and effective. A system based on our technique would thus meet all four requirements for efficient and effective image copyright enforcement.

The remainder of this paper is organized as follows. Section 2 reviews the OMEDRANK algorithm and its properties. Section 3 describes the application of the OMEDRANK algorithm to local descriptors, and the associated problems. Section 4 presents our new indexing approach, the PvS-index, and Section 5 describes our performance experiments. Finally, Section 6 gives our conclusions and directions for future work.

2 The OMEDRANK Algorithm

This section gives the practical details on OMEDRANK that are necessary for understanding the remainder of the paper. This overview is very brief; for further information on the theoretical underpinnings and optimality results of OMEDRANK, see [5, 8].

The discussion in this section assumes a single-descriptor k-nearest neighbor search on a collection of n descriptors in d dimensions. At the end of the section we review some of the nice properties of the algorithm, which make it the logical choice for our application, and in Section 3 we describe the extension of OMEDRANK to multi-descriptor searches.

2.1 Indexing

The OMEDRANK algorithm requires a preprocessing step to build several indices that are subsequently searched. First a set of d' random lines is chosen (typically $d' \leq d$) and for each of these lines, a B^+-tree index is created and initialized. Each descriptor s is then projected onto each of the random lines. For random line j, a pair (id_s, v_s^j) is created, where id_s is the identifier of the descriptor and v_s^j is the value of the descriptor along the random line j. This pair is then inserted into the B^+-tree index for line j, which is ordered by the v_s^j values. When all the descriptors in the collection have been processed in this manner, the OMEDRANK index is ready and consists of d' B^+-trees, each containing n pairs of (id_s, v_s^j) values.

2.2 Searching

The query descriptor q is projected onto each of the d' random lines, giving a value of v_q^j for random line j. Each B^+-tree index is then probed with the appropriate value to find a starting point for the query. Next, two cursors are started for each index, respectively reading successively lower and higher values. The cursors are used in a round-robin fashion, to simultaneously traverse all d' B^+-tree indices and retrieve the descriptor identifiers id_s. The algorithm keeps track of how often each descriptor identifier is encountered, while these cursors are moved. When a particular descriptor identifier has been seen in more than $\frac{1}{2}d'$ indices, it is returned as the nearest neighbor. Processing then continues, until k descriptor identifiers have been returned.

2.3 Properties of OMEDRANK

The OMEDRANK algorithm implements the *median rank* distance function. In [5], this distance function is shown to approximate Euclidean distance, but also to have several nice properties of its own. The OMEDRANK algorithm itself also has the following desirable fea-

tures that makes it a strong candidate for query evaluation with local descriptors:

1. The algorithm allows for trading off efficiency for effectiveness, with some probabilistic guarantees on the quality of the result. As demonstrated in [3], the local descriptor search can tolerate approximate results, due to the inherent redundancy resulting from the sheer number of query descriptors involved.

2. By projecting the descriptors to random lines in the data space, the approach allows for a very clever dimensionality reduction. This may become even more important in the future, as researchers are already working on 128-dimensional local descriptors [10].

3. Since the algorithm is based on the aggregation of ranking rather than complex distance calculations, it has the potential to use the CPU more efficiently. Putting more emphasis on disk operations has the advantage of allowing for performance improvements, for example via RAID solutions, without changing the core algorithm.[1]

4. The algorithm uses several standard B^+-trees, rather than a single specialized index. This has three main advantages. First, very efficient implementations of B^+-trees exist. Second, B^+-trees handle updates gracefully. Third, as performance of indices degrades due to updates, each index can be individually and efficiently reorganized.

[1] When using OMEDRANK for a large collection of data, the processing is actually bound by the memory latency. It is only when OMEDRANK is used with PvS-indices that these benefits materialize.

As far as we know, this is the only available algorithm that has all these nice properties, making it the logical choice for our application.

3 OMEDRANK and Local Descriptors

In this section, we first briefly review the way local descriptors are typically used for query processing. We then detail the changes required to adapt OMEDRANK to query processing over local descriptors. Finally, we present experimental results, which show that OMEDRANK performs poorly with local descriptors.

3.1 Local Descriptors

The creation of local descriptors proceeds in three steps: First, specific points in the image, called interest points, are selected based on the shape of the image signal at these points, according to the approach proposed by Harris [7]. Second, the signal around each interest point is characterized by its convolution with a Gaussian function and its derivatives up to the third order. Finally, these derivatives are mixed to enforce invariance properties and to make the descriptors robust to several image changes. For details, see [1].

The current version of the descriptor creation yields 24-dimensional descriptors. The number of descriptors per image can vary significantly, depending on the size, resolution, quality and contents of the images. For typical images, several hundreds of descriptors may be created; for large, high quality images, even more than a thousand descriptors. Computing descriptors over all the images is done off-line. To know which image a descriptor has been computed

from, image identifiers are stored together with the descriptors.

The similarity retrieval proceeds as follows. Interest points are first identified in the query image and the corresponding local query descriptors are computed. The query descriptors are then used to query the descriptor database. For each descriptor of the query image, the system returns the 30 most similar descriptors found in the database, using Euclidean distance for the measure of similarity. The image identifiers of these descriptors indicate the associated image from the collection, and it is straight-forward to count the number of occurrences of each image identifier during the whole retrieval process. Once all the query descriptors have been used to probe the database, the occurrence counters allow the system to rank the candidate images by decreasing similarity, with the image with the most votes considered most similar, the image with the second most votes the second most similar, and so on. For further details, see [4].

If the database does not contain any image that is similar to the query image, then the votes of all the images returned are roughly similar and of small values. In contrast, if one or more images are indeed similar, then they have many more votes.

3.2 Adapting OMEDRANK

The indexing step for OMEDRANK is slightly more complicated because in our application many descriptors are associated with the same image identifier. OMEDRANK, however, must count the number of times each descriptor identifier is encountered while the cursors are moved. Therefore, with local descriptors, image identifiers originally associated to descriptors must be replaced by *descriptor identifiers*. A secondary data structure mapping descriptor identifiers to image identifiers is stored on disk. This data structure is heap-based, and is small and efficient. Aside from this mapping, the indexing step proceeds exactly as before, except that with local descriptors the index becomes much larger as more descriptors are inserted.

During the search, each query descriptor is used in turn to search for the k-nearest neighbors. As before, each neighboring descriptor gives one vote to an image; the corresponding image identifier is then found through the secondary data structure described above. Once all descriptors have been processed, the images are ranked based on the number of votes, again as described above.

3.3 Performance of OMEDRANK

To study the performance of OMEDRANK, we used a collection of almost 30 thousand, real-life, high resolution images obtained from a photo agency, resulting in more than 20 million 24-dimensional descriptors. More details on the collection and the workload are described in Section 5.

For queries, we have selected random images from the collection and used the protocol described in [3] to apply some standard "pirate" manipulations to images, resulting in several different images variants subsequently used as queries. The search queries the image database with the pirated images and tries to find the original images at the top of the ranked list of images that is returned (with rank 1).

We have used the StirMark benchmarking software (version 4.0) to generate the image variants used [11]. StirMark has originally been designed to attack the watermarks inserted into images (i.e., wash out, partially destroy or make illeg-

ible). It applies many modifications to images that have in general only a small visual impact, yet, it deeply touches the signal of the images. Its limited visual impact and the large number of variants it creates is the rationale for having used this software to emulate piracy.

Given an image, StirMark creates more than a hundred different variations, using cropping, compression, rotation, rescaling, signal manipulations, random geometric distortions etc. Previous studies have shown the local descriptors to handle most of these variations very well, aside from a few variations that significantly reduce the visual quality of the image [3].

In this paper, we focus mostly on five representative StirMark variants. The first variant is CROP 75, which selects 75% of the area of the image; as it applies no other transformation to the image, it is handled easily by the local descriptors. ROT 90 rotates the image by 90 degrees; as the descriptors are designed to handle rotations well, it is also handled easily. JPEG 80 compresses the image data by 20% and ROT 15 rotates the image by 15 degrees; both are handled quite well by the local descriptors. The final variant is JPEG 15, which compresses the image data by 85%, resulting in a very rough approximation of the original image. This variant is not handled well by the local descriptors, which have not been designed to handle such severe compression levels. The results are averaged over 18 different original images.

Turning to the performance results, Figure 1 shows the response time for the five variants of the query images. The x-axis shows the number of indices used by OMEDRANK to answer the query. For each variant, the time taken by the sequential scan is also shown as a straight line (with a matching pattern) for comparison. Note, that the differences in query processing

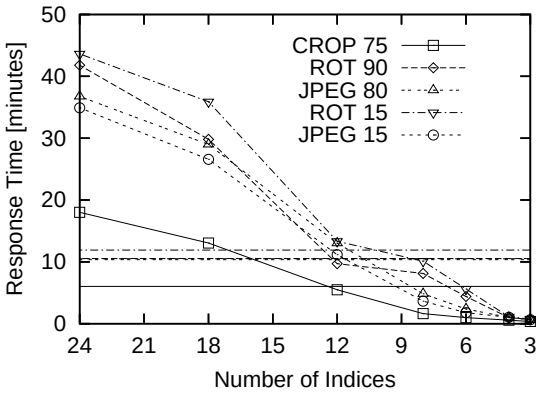

Figure 1: Response time of OMEDRANK and sequential search

time between the five variants arise due to the fact that each variant has a different number of query descriptors (ranging from 238 descriptors for CROP 75 to 483 descriptors for ROT 15).

Overall, Figure 1 shows that when many indices are used, the query processing of OMEDRANK is significantly slower than the sequential scan. The break-even point is roughly at 12 indices; with fewer indices OMEDRANK is faster than a sequential scan. The performance of OMEDRANK is largely dependent on the *probe depth* of the search, or the number of descriptor identifiers read from each index for each query descriptor. Each such descriptor identifier must be entered into a hash table, which keeps track of all descriptor identifiers seen during the search. When using many indices almost all descriptors of the collection are seen in at least one of the indices and the hash table becomes very large. Each hash table update therefore suffers an access to main memory, making the search process bound primarily by the memory latency. By using only three indices, however, OMEDRANK is able to run in less than 10% of the time re-

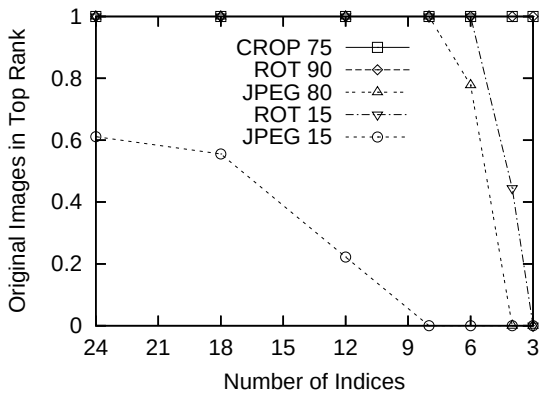

Figure 2: Rank of original image for OMEDRANK

quired for a sequential scan, taking 22-45 seconds to complete the query evaluation.

Figure 2, on the other hand, represents the quality of the OMEDRANK search. The figure shows the proportion of query images that return the original image in the top rank. As the figure shows, the original images are generally lost below 6 indices for the medium variants. For the difficult JPEG 15 variant, the image is only found in about half the cases, even with 24 indices. Note that the sequential scan (not shown) always returns the original image in the first position in all variants except the JPEG 15 variant. So, while the performance of OMEDRANK is acceptable using few indices, the effectiveness suffers.

3.4 Discussion

The key problem of the OMEDRANK indexing approach is that when projecting multiple dimensions onto a single line, many unrelated descriptors may be projected in between near neighbors, or "into-the-way" of the search. This is illustrated in the following example.

Example 1 *Consider Figure 3, which shows twelve data points (labeled a through l) in two dimensions. Assume now that descriptor a in Figure 3 belongs to a pirated image, and that the image has been modified, such that the query descriptor corresponding to a is now a'. We note that after the modification of the image, the original descriptor a is only the second nearest neighbor of a'. Assuming the local descriptor search process is returning three nearest neighbors, however, a would still give a vote to the image.*

Let us now apply the OMEDRANK algorithm to this example. Focusing first on the projection to line L_1, we observe that both points g and h are projected in between a and b, such that on the line L_1, descriptor a is now only the fourth nearest neighbor to a'. With a large collection, it is likely that many descriptors will be projected into-the-way of the search in this manner; this is demonstrated by the high probe depth of the OMEDRANK search.

Turning to the line L_2, we observe that the same problem arises, but this time descriptors f, e, i and k are projected into the way of the search. The reader is invited to confirm that when running OMEDRANK with this dataset and these lines, the order of nearest descriptors would be b, e, h, i, g, f, l, a, c, k, d and j. The original descriptor is hence only the 8th nearest neighbor, and the vote should be considered lost in this small example.

As the performance results of the this section have shown, the OMEDRANK algorithm is quite efficient when used with only three indices, but the quality of the results is unacceptable. This fact motivated us to investigate how we could retain the efficient performance of OMEDRANK while improving the results.

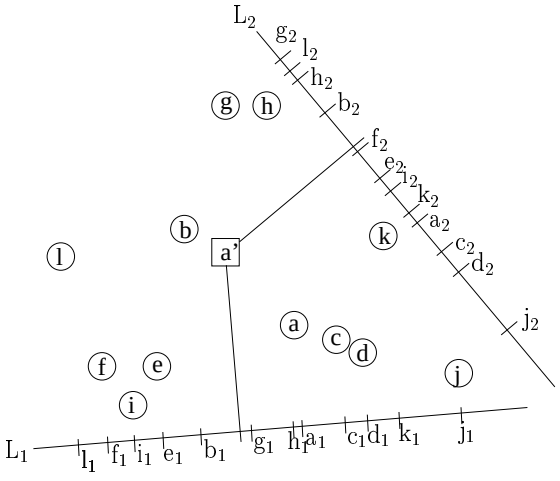

Figure 3: Projection example

4 The PvS-Index

In this section, we propose a new indexing strategy, called the PvS-index, which is based on a combination of projections and segmentations. By replacing each B^+-tree index used by OMEDRANK with a PvS-index, we are able to obtain results of good quality while guaranteeing excellent performance.

A recurring theme in multi-dimensional indexing structures is some form of segmentation of the data space. The first key idea of the PvS-index is indeed to segment the data space, but in combination with random projections, as the random projections can in many cases alleviate data distribution problems and result in better performance. The second key idea of the PvS-index is to introduce redundancy into the index, by creating overlapping segments. Although the redundancy increases the storage requirements of the index, it allows the OMEDRANK algorithm to restrict its search to a single segment of each index per query descriptor. Because disk space is cheap, while disk performance is more costly, it is a good trade-off.

This section proceeds as follows. We first introduce our basic approach of segmenting and projecting through a simple example. We then describe the overlapping segments and proceed to define the PvS-index, which consists of a single "segment tree" storing pointers to small B^+-trees, which in turn may be used by OMEDRANK. Finally, we present our implementation. Due to space limitations, the description here is brief; more details will be presented in a future technical report.

4.1 Segmentation and Re-Projection

The B^+-tree indices used for OMEDRANK were formed by a single projection of each descriptor onto a random line. It can be shown that descriptors that are far apart on the projected line, are also far apart in the original space, while descriptors that are close on the projected line, may or may not be close in reality.

The main problem with OMEDRANK arises exactly when descriptors that are far apart in reality, appear to be close on the projected line. The PvS-index addresses this problem by 1) segmenting the projected line to separate descriptors that are known to be far apart, and 2) reprojecting the data of each segment onto a *new random line* to put distance between descriptors that appeared to be close on the first projected line, but were far apart in reality. The following example, which describes a single PvS-index, demonstrates this approach.

Example 2 *Consider Figure 4, which shows the same twelve data points as before. Assume the data points are first projected onto the line* $L_{1,1}$.

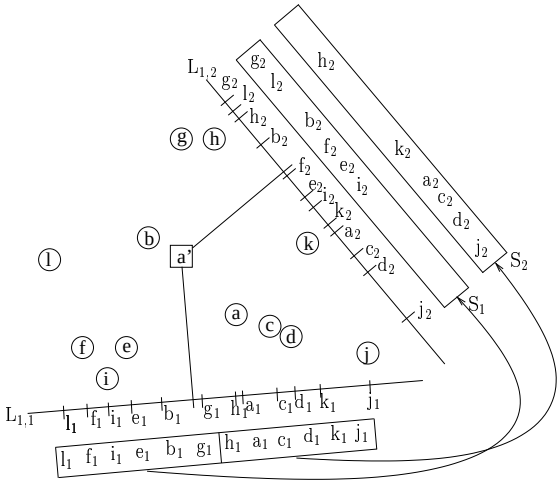

Figure 4: Segmentations and re-projections

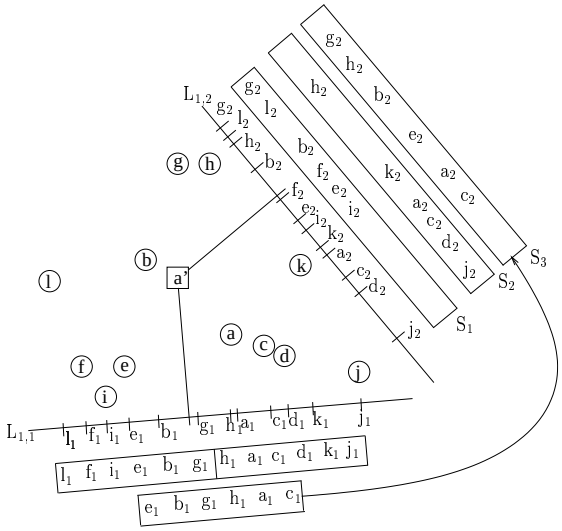

Figure 5: Overlapping segments

The projected line is then segmented, in this example into two segments, and re-projected onto the second line, $L_{1,2}$, yielding the two segments S_1 and S_2. When searching for query descriptor a' in these segments, the descriptor would land in segment S_1, between b_2 and f_2.

There are three things that must be noted about Example 2. First, in order to use the index in OMEDRANK search, further PvS-indices need to be created with projections onto different lines, therefore having very different segmentations. Second, with a large collection, the process of segmenting and re-projecting is repeated, until each segment is of an appropriate size for efficient query processing. If the number of segments created at level i is s_i, then the total number of segments in this approach is $\prod_i s_i$. Third, the segmentation process of Example 2 is not yet complete. As mentioned above, we have chosen to introduce redundancy into the index using overlapping segments; this approach is described next.

4.2 Overlapping Segments

Note that in many cases, the segmentation may push apart descriptors that are close in reality, such as a and b in Figure 4. In fact, the search for a' would not find the data point a in Figure 4. While one solution to this problem would be to search many segments, this would quickly lead to an exponential number of segments being searched and the search performance would suffer. Instead, we have taken a different approach with the PvS-index, which is to create overlapping segments at each level, in order to reduce the likelihood of near neighbors being separated.

Example 3 *Figure 5 illustrates how the overlapping segments work. In the figure, a new segment is created that overlaps half of each of the previous segments on line $L_{1,1}$; the data points of this segment are then re-projected onto $L_{1,2}$ to form S_3.*

When searching for query descriptor a', both

segments S_1 and S_3 are candidates, as a' matches both along the $L_{1,1}$ line. As only one segment can be searched, however, the search would choose between the two segments. Because the value of a' is closer to the middle of the new overlapping segment along the $L_{1,1}$ line, segment S_3 would be chosen, which contains all of the actual nearest neighbors of a'.

Let us examine more closely how to decide which overlapping segment to descend into during the search. The reason for overlapping the segments is to avoid searching near the border of a segment. Therefore, a value must be stored which is half-way between the borders of overlapping segments. For Figure 5, the half-way value for S_1 and S_3 would be the value b_1. All search descriptors that are projected to a value smaller than b_1 would search in S_1, while equal or higher values would search in S_3. Similarly, the value of a_1 would be chosen to decide between S_3 and S_2.

The overlapping segments, of course, lead to an enlargement of the index. At each level i, instead of s_i segments we now have $2s_i - 1$ segments, for $\prod_i (2s_i - 1)$ segments in total. Each additional level thus almost duplicates the size of the index. By requiring the OMEDRANK algorithm to search only one segment from each index, however, the penalty is only paid in construction and storage of the index, but not during the search. Again, as disks are increasing rapidly in storage capacity but only slowly in speed, this is a great trade-off.

4.3 The PvS-Index Structure

In order to support the decisions made during the search on which segments to descend into, the system maintains a tree structure called the *segment tree*. The segment tree is essentially a B-tree like structure that stores in intermediate nodes the half-way values between the segment boundaries at each level. At the leaf level, the segment tree stores pointers to the actual B^+-trees containing the final segments, which the OMEDRANK algorithm can search.

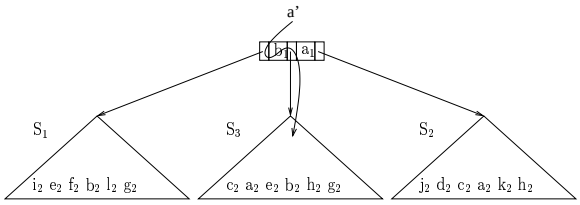

Figure 6: A PvS-index

Figure 6 shows the segment tree associated with the PvS-index of Figure 5, as well as the search path that the search for a' would traverse. During the search, the descriptor is projected onto the appropriate line and the projected value is compared with the half-way points. Once the appropriate sub-node is found, the descriptor is re-projected and the search process repeated. Once the appropriate leaf is found, the corresponding B^+-tree is passed to the OMEDRANK algorithm for processing.

For large collections, the segmenting and re-projecting can be applied repeatedly, until segments of a reasonable size are obtained. In our experiments, we have made each segment 128KB, which is exactly the I/O granularity of the Linux system we are using. It is straight-forward to visualize the extension of the segment tree of Figure 6 to multi-level PvS-indices.

StirMark Variant	Category	Response Time	Query Descriptors	Original Votes	Competitor Votes	Ratio (Competitor/Original)
CROP 75	Easy	4.5 minutes	237.7	227.7	23.2	0.10
ROT 90	Easy	9.0 minutes	412.8	378.0	40.7	0.11
JPEG 80	Medium	8.8 minutes	409.7	221.8	35.3	0.16
ROT 15	Medium	10.2 minutes	483.4	174.8	43.7	0.25
JPEG 15	Hard	8.0 minutes	419.0	40.1	35.9	0.90

Table 1: Results of the sequential scan

4.4 Implementation

We have taken great care to implement the PvS-index creation very efficiently. The index creation starts by creating all the projected lines that are required for the index. The lines are chosen randomly, with the constraint that they must be nearly orthonormal ($\cos(angle) < 0.1$). Then each descriptor is projected onto all lines, and the projected values are stored in a file. The index creation process itself is a recursive process of sorting the current file on the appropriate projection, creating the segments (noting in the process the values for the segment tree) and writing into new files, each of which is subsequently sorted and reprojected, until the leaf level is reached. Note that the segments are processed depth-first for performance reasons, such that only one path through the index is stored at a time. Once the leaf level is reached, the final projected values and their descriptor identifiers are inserted into a B^+-tree which is then stored on disk.

The retrieval progresses as follows. For each query descriptor, the segment trees of the PvS-indices are searched to find the appropriate B^+-trees, which are then searched by OMEDRANK. Two performance improvements to the search are worth noting, however.

First, in order to improve the CPU cache performance of the OMEDRANK search, we read 16 descriptor identifiers from each cursor of each index at a time, rather than a single identifier. As a result, the CPU time is reduced by 50%. Although this is a further approximation to the median rank distance function, our results show it to be acceptable.

Second, we have implemented a very efficient hash-table for the OMEDRANK search. As noted before, there is a hash-table at the heart of the OMEDRANK algorithm, which keeps track of the descriptors seen so far. Every time a value is returned from one of the index scans, this table must be probed, so the efficiency of this hash-table is paramount to the efficiency of the search. Because our segments guarantee an upper bound on the amount of data that will be read, we were able to implement a very efficient hash-table, which has a hash function based only on bit-operations and can fit inside the L2-cache of the computer.

5 Performance Analysis

In this section, we present results from our detailed performance study. We first describe the experimental setup, including the descriptor collection and workloads. We then describe experiments to examine 1) the effect of the depth of the segment tree and 2) the effect of varying the

number of indices used in the search. The section concludes with a discussion of the results.

5.1 Experimental Setup

5.1.1 System Setup

The experiments were run on DELL PowerEdge 1850 machines, each equipped with two 3GHz Intel Pentium 4 processors, 2GB of DDR2-memory, 1MB CPU cache, and two 140GB 10Krpm SCSI disks. The machines were running Gentoo Linux (2.6.7 kernel) and the ReiserFS file system. In all experiments, only one of the disks is used.

We have accumulated many metrics to measure the efficiency and effectiveness of the search. The primary efficiency metric is the response time of the search, but we also examine CPU time and other metrics, as appropriate.[2] The primary effectiveness metrics we have studied are the rank of images and the number of votes, in particular for the original image, from which the query image was derived, and the "best competitor" or the non-original image with the most votes.

5.1.2 Workload

The image collection used in our experiments consists of 29,077 high quality press photos. The descriptor collection was produced by first resizing each image, such that the larger size is 512 pixels, and then calculating the local descriptors. The resulting descriptor collection contains 20,506,800 descriptors; as each descriptor requires 100 bytes, the size of the collection is just under 2GB.

We chose randomly 18 different "original" images. From each of these images, we used StirMark [11] to create "pirated" images, as described in Section 3. We have experimented with 14 different StirMark variants, which we have categorized as "easy", "medium" and "hard". In our presentation, however, we have chosen to focus on the five representative variants described in Section 3. For each variant, the results are averaged over the 18 different images.[3]

We compare the performance of the various configurations of the PvS-index to that of the sequential scan; the performance of the sequential scan is shown in Table 1. As the table shows, the number of descriptors in the query images ranges from 400-500, except for CROP 75, and the query processing time is largely dependent on the number of query descriptors.

Table 1 also shows that for the easy variants, over 90% of the descriptors are found within the 30 nearest neighbors, while the best competitor received votes for about one descriptor in 10. For the medium variants, the search is also effective, finding $\frac{1}{3}$ to $\frac{1}{2}$ of the descriptors. For the hard variant, the original image is sometimes found at the top and sometimes not.

5.2 Experiment 1: Segmentation Strategy

In this experiment we focus on the effect of the segmentation strategy. We decided, as mentioned above, to make each segment the size of a single disk read, which is 128KB on our system. In each such segment, we fit a single B^+-tree

[2] Note, that we have taken care not to overlap I/O and CPU processing for better understanding of the performance; we have also avoided any prefetching or other optimization of disk performance, which we leave for future work.

[3] Time constraints prevented us from running further experiments; we will run many more images for the final version of the paper.

containing the data of the segment. Since B^+-trees typically have utilization of 67%, we can fit roughly 10,000 (*descriptor identifier*, *value*) pairs into a segment. With our collection, that gives roughly 2000 non-overlapping segments in each PvS-index (below we discuss how many overlapping segments are created).

There are infinitely many ways to build a segment tree to obtain 2000 non-overlapping segments. Segmenting "aggressively" yields shallow trees with many segments at each node, while segmenting "gently" yields deep trees with few segments at each node. This is, in fact, the reason for the name PvS: Projection vs. Segmentation.

We have focused on a uniform segmentation strategy, where the each node of the segment tree contains roughly the same number of non-overlapping segments. With a segment tree of l levels, each node contains roughly $\sqrt[l]{2000}$ segments. We present the number of segments at each level, s_i, with an array of values. For example, with two levels, we have chosen $[45, 45]$, which yields 2,025 non-overlapping segments. With overlap, however, the segment tree contains 7,921 segments.

We have chosen to use only three PvS-indices in this experiment, which means that in total three indices are searched and two descriptor votes are required for a descriptor to be returned as a neighbor. This configuration was chosen as it gives good results in a very short time; we present results using more indices in the second experiment.

5.2.1 Index Creation

Table 2 shows the performance of the index building phase. As the table shows, the index creation performs very well, but the indices nearly double in size with each additional level of the segment tree, as explained in Section 4. For the experiments of this section, three indices were built for each configuration. The largest configuration for this experiment, which consists of three five-level PvS-indices trees, thus required about 13.5GB of disk space, and took an hour and fifteen minutes to build.

Segmentation Strategy	Size (GB)	Build Time per Index	Overlapping Segments
[45, 45]	1.00	6 minutes	7921
[13, 13, 12]	1.75	9 minutes	14375
[7, 7, 7, 6]	2.95	15 minutes	24167
[5, 5, 5, 4, 4]	4.50	25 minutes	35721

Table 2: PvS-index size and creation time

5.2.2 Efficiency

Figure 7 shows the total time to process the queries with different variants. The x-axis represents the height of the segment tree. As the figure shows, query processing always takes less than 16 seconds with PvS.[4] Referring back to table 1, we see that the corresponding time was over 10 minutes for the sequential scan. Note, that the performance of PvS is independent of the size of the collection, while the time of the sequential scan grows linearly.

While the bottleneck of the sequential scan is the high number of CPU-intensive distance calculations, the PvS-index suffers from the low performance of random disk operations. In fact,

[4]The fact that the search time is reduced for shallower segment trees is primarily due to buffer management effects. Unfortunately, clearing buffers in Linux is very time-consuming, so we were not able to counter the buffering effect. With the largest indices, however, so many segments are read that the buffer manager becomes ineffective.

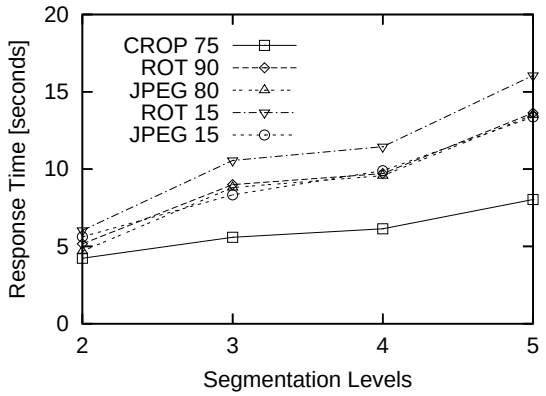

Figure 7: Response time of PvS (Exp. 1)

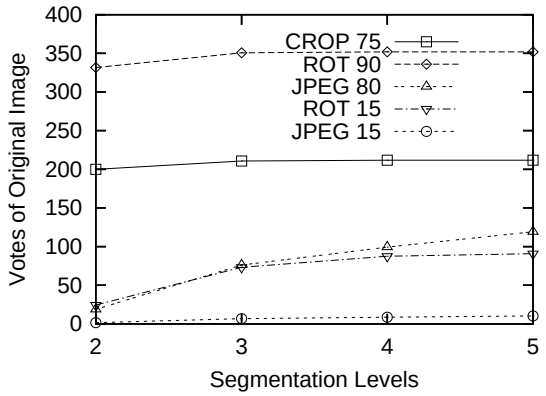

Figure 8: Votes of original image for PvS (Exp. 1)

the share of the CPU-time for PvS (not shown) ranges only from 3% to 10% of the total evaluation time, depending on the probe depth in the segments.

5.2.3 Effectiveness

Turning to the effectiveness, we have observed that for the easy and medium variants, the original images are always the first image returned (except for the JPEG 80 variant with only two levels), while the JPEG 15 variant is frequently lost. Figure 8, on the other hand, shows the number of votes for the original image. Again, this figure must be compared to the sequential scan results of Table 1. As the figure shows, the easy and hard variants are largely unaffected. The easy ones are still found with many votes, but the hard variant receives few votes.

The number of votes received by the original images greatly improves for the medium variants ROT 15 and JPEG 80 when going from two levels of segmentation to three levels. With only two levels, the data space is not segmented and reprojected enough to remove a significant fraction of the descriptors that get "into-the-way" of the search, reducing the likelihood with which the descriptor identifier of the original image gets actually stored in the segments that are searched. With three levels, however, the segmentation is improved due to the extra projection and many more votes are returned. Note that further segmentation does not improve the quality of the results significantly, although more votes are returned. For this collection three levels of segmentations are therefore enough to distinguish the original images from other images.

Overall, comparing these numbers to the results in Table 1 we observe that PvS loses several votes compared to the sequential scan. This effect can be explained by the fact that even for the deeper segment trees, some points are still projected into the way of the search, especially in densely populated areas of the 24-dimensional descriptor space.

When we look at the number of votes of the best competitor in the final result (not shown), we observe that the PvS-index always yields fewer than 10 votes on average for non-matching images, which is significantly lower than for the sequential scan. The vote ratio of the best

competitor to the original image (not shown) is therefore much better for PvS. For the easy and medium variants, with the higher segment trees, the best competitor receives fewer than 10% of the votes of the original image, while comparable numbers for the sequential scan range from 10% to 25%. PvS is therefore in fact better able to distinguish the original image from non-matching images.

For detecting copyright protection it is important to raise as few false alarms as possible. Noting that the number of votes of the best competitor represents the highest score of a non-matching image, our results indicate that PvS will very rarely return false positives.

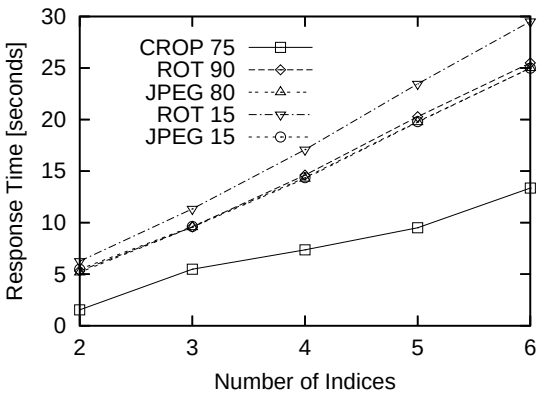

Figure 9: Response time of PvS with four-level segment trees (Exp. 2)

5.2.4 Discussion

Based on the results of this experiment, we can conclude that using the PvS-indices with the OMEDRANK algorithm results in very efficient and effective query processing. Although some votes are lost in the process, the results are still of high quality; in the future we will also consider attacking this loss by gathering more than 30 nearest neighbors. From our results we conclude that using three- to five-level PvS-indices is recommended; the choice depends mostly on the disk space available.

5.3 Experiment 2: Number of indices

In the second experiment we wanted to determine how many PvS-indices we should create for best results. We focused on PvS-indices with 3 to 5 levels of segmentation, as they gave the best results, and varied the number of indices from 2 to 6. Since the median rank distance function requires a descriptor to be found in *more than* half of the indices to be considered a match, the vote requirements are 2/2 2/3, 3/4, 3/5 and 4/6.

Figure 9 shows the response time of the search with a four-level PvS-index. As the figure shows, the response time increases linearly, due to a linear increase in the number of disk reads. Similar results are seen for three- and five-level PvS-indices (not shown).

Turning to the effectiveness results, we observed that the results for the "easy" and "hard" variants were essentially unaffected by the increased number of indices. The "medium" variants were affected, however, and we present results for the JPEG 80 variant.

Figure 10 shows the number of votes of the original image for three-, four-, and five-level PvS-indices. As the figure shows, the score of the images is always maximized with five PvS-indices. For the three-level PvS-indices, the curve shows a slight "M"-shape. This can be explained by examining the proportion of the indices that the descriptor needs to be seen in, which is 1, 0.67, 0.75, 0.60, and 0.67 for 2–6 lines, respectively. The lower the proportion, the more votes are generally found. This is also the reason

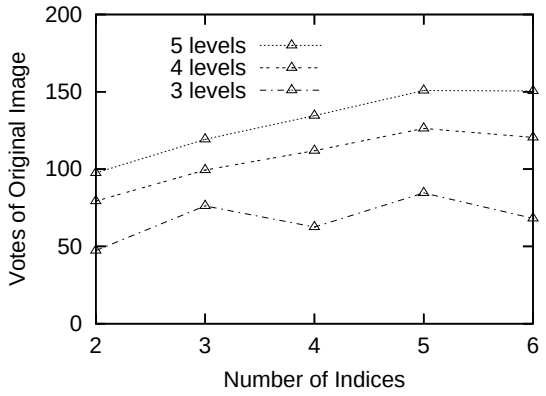

Figure 10: Votes of the JPEG 80 variant (Exp. 2)

why 5 indices are always better than 6 indices. For four- or five-level PvS-indices, however, fewer indices perform worse, albeit always better than using a three-level index.

To conclude this experiment, the results show that while using five indices gives slightly better results than using three indices, the loss of quality for using only three indices is marginal and is offset by faster query processing.

5.4 Discussion

Because searching with PvS-indices only requires one disk read per query descriptor per index, we guarantee an upper bound on the query processing time, regardless of the size of the image collection. We therefore strongly believe that our scheme is able to scale to very large collections of images.

Guaranteeing effectiveness when searching a very large collection is clearly crucial. Experiment 1 gave evidences that a collection of descriptors has to be segmented enough to provide high-quality results. Creating deep segment trees only consumes disk space and does not induce any additional processing cost at search time—we believe this is a very database friendly trade-off.

In addition, and independently from the depth of PvS-indices, the quality of the search might be improved by increasing the number of such PvS-indices. This can counter-balance disk-space consumption: if space is a major concern, and if CPU-power is not an issue, then creating more, shallower PvS-indices is the way to go.

6 Conclusion and Future Work

In this paper we have studied the application of the OMEDRANK algorithm to the image copyright protection problem using local descriptors. We have proposed a new indexing strategy, the PvS-index, which utilizes all the nice properties of the OMEDRANK algorithm. We have shown that using PvS-indices on modest hardware results in excellent query processing performance. Because of the redundancy of the local descriptor approach, the approximate nature of the PvS-index results in efficient and effective image copyright protection.

There are several avenues for future work, including examining the effect of returning more than 30 nearest neighbors to each query descriptor. We are planning to study buffer management and prefetching policies for the PvS-indices, as well as the effect of advanced disk architectures. Since using the PvS-index is I/O bound and guarantees constant evaluation time for each query descriptor, we will also study the effectiveness of the PvS-index with very large descriptor collections.

Acknowledgments

This work is part of the *Eff*^{2} project on *Efficient and Effective Image Retrieval*. The *Eff*^{2} project is a cooperation between researchers at Reykjavík University, Iceland, and the IRISA laboratory in Rennes, France, and is partially supported by Rannís Technical Research Grant 030290004 and ÉGIDE Jules Verne Travel Grant 4-2003. See http://datalab.ru.is/eff2.

References

[1] L. Amsaleg and P. Gros. Content-based retrieval using local descriptors: Problems and issues from a database perspective. *Pattern Analysis and Applications*, 4(2/3), 2001.

[2] S.-A. Berrani, L. Amsaleg, and P. Gros. Approximate searches: k-neighbors + precision. In *Proc. of ACM CIKM Conference, New Orleans, LA*, 2003.

[3] S.-A. Berrani, L. Amsaleg, and P. Gros. Robust content-based image searches for copyright protection. In *Proc. of ACM MMDB Workshop, New Orleans, LA*, 2003.

[4] S.H. Einarsson, R.Ý. Grétarsdóttir, B.Þ. Jónsson, and L. Amsaleg. The *Eff*^{2} image retrieval system prototype. In *Proc. of IASTED DBA Conf.*, Innsbruck, Austria, 2005.

[5] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proc. of the ACM SIGMOD Conference, San Diego, CA*, 2003.

[6] L.M.J. Florack, B.M. ter Haar Romeny, J.J. Koenderink, and M.A. Viergever. General intensity transformation and differential invariants. *Journal of Mathematical Imaging and Vision*, 4(2), 1994.

[7] C. Harris and M. Stephens. A combined corner and edge detector. In *Proc. 4th Alvey Vision Conference*, 1988.

[8] H. Lejsek and F.H. Ásmundsson. The application of the MEDRANK algorithm to content-based image retrieval using local descriptors. BS project report, Reykjavík University, 2004.

[9] C. Li, E.Y. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4), 2002.

[10] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. Journal of Computer Vision*, 60(2), 2004.

[11] F.A.P. Petitcolas et al. A public automated web-based evaluation service for watermarking schemes: StirMark benchmark. In *Proc. of Electronic Imaging, Security and Watermarking of Multimedia Contents III, San Jose, CA*, 2001.

[12] M. Stricker and M. Swain. The capacity of color histogram indexing. In *Proc. of CVPR Conf. Seattle, WA*, 1994.

[13] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of VLDB Conference, New York, NY*, 1998.