A Relaxed Algorithm for Similarity Queries Performed with High-Dimensional Access Structures

Andreas Henrich

University of Bayreuth
D-95440 Bayreuth, Germany
andreas.henrich@uni-bayreuth.de

Abstract. Similarity queries searching for the most similar objects in a database compared to a given sample object are an important requirement for multimedia databases. However, strict mathematical correctness is not essential in many applications of similarity queries. For example, if we are concerned with image retrieval based on color and texture similarity, slight mathematical inaccuracies will hardly be recognized by the human observer. Therefore we present a relaxed algorithm to perform similarity queries for multidimensional index structures. This algorithm assures only that a user defined portion of the result list containing n elements actually belongs to the n most similar objects — the remaining elements are subject to a best effort semantics. As we will demonstrate, this allows to improve the performance of similarity queries by about 25 % with only marginal inaccuracies in the result.

1 Motivation

For multimedia databases similarity queries are an important requirement. To this end, objects are represented by feature vectors. For example, an image can be represented by a color histogram, texture measures or shape measures. Then objects similar to a given sample object can be determined with a similarity search based on these feature vectors.

To support such similarity queries, various access structures have been developed for multi-dimensional feature vectors — e.g. the SS-tree [20], the VAMSplit R-tree [19], the TV-tree [14], the SR-tree [13] the X-tree [3], the R*-tree [1] or the LSD^h-tree [10]. All these access structures provide a similarity search operation. Unfortunately the performance of these access structures decreases drastically for high-dimensional feature vectors. It has been shown that this effect is problem inherent [8,9,2]. As a consequence, it will not be possible to develop an access structure performing well for all conceivable feature vectors.

Therefore, our approach is not to design yet another access structure performing better than the known structures under some specific circumstances but to present an approach which is applicable to all access structures mentioned above for a broad variety of applications. The basic idea of this approach is to trade a

bit of accuracy for a significantly improved performance. It is based on the fact that strict mathematical correctness is not essential in many application areas of similarity search operations. In many applications the similarity calculation based on feature vectors is only a model for the human perception of similarity. Here slight mathematical inaccuracies with the calculation of the most similar objects might not even be observable by the person stating the query.

Our approach can be seen as an adaptation of the algorithm of Buckley and Lewit [5] implementing a similarity search for text documents based on the vector space model. This algorithm employs inverted files processed one file after the other. The user can define a parameter $\alpha \in (0, 1]$ and the idea is to stop the algorithm as soon as $\lceil \alpha \cdot n \rceil$ elements of the required n result elements surly belong to the n best matching text documents. We will describe this algorithm in some more depth in section 2.

In the present paper we adapt this approach to tree-based multi-dimensional access structures. Furthermore we present experimental results demonstrating that the approach yields significant performance improvements with hardly observable losses in accuracy.

Let us consider a simple example to sketch the idea of our approach. Assume the 2-dimensional data space given in figure 1. The data space is mapped onto 16 buckets holding the objects located in the corresponding regions of the data space. Each bucket has a capacity of at most three objects. In this situation we perform a query searching for the two objects ($n = 2$) most similar to the sample object q with feature vector $\boldsymbol{q} = (33, 65)$. Faced with this sample object we first consider bucket B_1 for which the associated region covers the query vector $(33, 65)$. When we consider the three objects stored in B_1, we have to take into account that other buckets can contain similar objects too. If we assume that the Euclidean distance is used as the similarity measure, the bucket which could hold the most similar objects is B_2. From the region associated with B_2 we can derive that objects stored in B_2 must have at least a distance of 8 to $\boldsymbol{q}$. As a consequence, we can be sure that object o_1 with a distance of 6 is the most similar object. For o_2, which has a distance of 10, we cannot be sure that it is the second best fitting object, because B_2 could contain more similar objects. If the user has chosen a value of 0.5 for α, our approach would nevertheless return the list with the objects o_1 and o_2 as the result and avoid additional bucket accesses, because 50 % of the result are definitely correct. Of course the other 50 % of the result could be wrong — but in this case they will usually be "not much worse" than the correct answer.

In the above example we avoided an additional bucket access at the price of a potentially slightly incorrect result. Of course there are applications were such potentially incorrect results are objectionable. On the other hand, in applications such as image retrieval based on color or texture similarity slight inaccuracies are unproblematic.

It has to be mentioned, that the efficient processing of queries searching the n best matching objects is also known as "Top N" query processing in the field of query optimization. Interesting approaches in this direction are presented in [6,

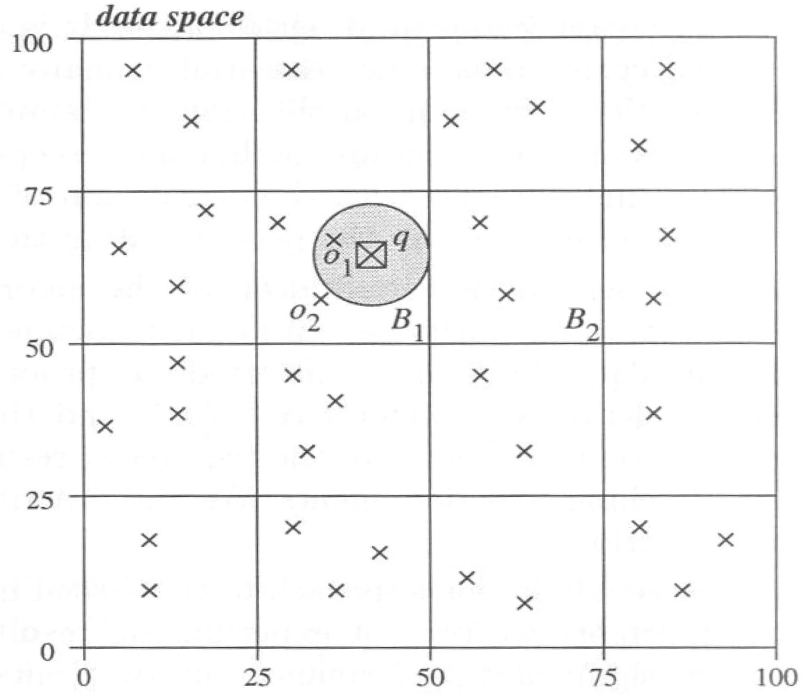

Fig. 1. Search for objects similar to the object q with feature vector $\boldsymbol{q} = (33, 65)$

7,4]. However, these approaches address the problem from the perspective of th query optimizer in a — usually relational — database system. Here the acces structure is more or less a "black box" with a well defined behaviour. In contrast our approach addresses the problem from the perspective of the access structur and tries to improve the performance of similarity queries based on the interna processes of the similarity query algorithm.

The rest of the paper is organized as follows: In section 2 we present the ide behind the Buckley-Lewit-algorithm which inspired our approach. Thereafter w elaborate our approach for the LSD^h-tree. Although the approach is applicabl for all tree-based multi-dimensional access structures we choose to present it fo a concrete access structure in order to be as precise as possible. Therefore w present the LSD^h-tree shortly in section 3. In section 4 we present the modifie algorithm for similarity queries in detail. Thereafter experimental results ar given in section 5. Finally section 6 concludes the paper.

2 The Buckley-Lewit-Algorithm

The Buckley-Lewit-algorithm [5] implements the vector space model (VSM) [15] searching for text documents relevant with respect to a given information request. The VSM assumes that an available *term set* (called *vocabulary*) is used to identify both maintained documents and information requests. This term set contains the terms which might be useful to describe a given information need.

In the VSM queries and documents are represented as *term vectors* of the form $D_i = (a_{i1}, a_{i2}, \ldots, a_{it})$ and $Q_j = (q_{j1}, q_{j2}, \ldots, q_{jt})$ where t is the number of terms in the vocabulary and where the coefficients a_{ik} and q_{jk} represent the

relevance of document D_i or query Q_j, respectively, with respect to term k. In the literature various term-weighting formulas have been proposed to calculate the a_{ik} and q_{jk} automatically for a given text document or query text, respectively (see e.g. [16]). For the Buckley-Lewit-algorithm it is only important to note that the document representation vectors are usually normalized — consequently $a_{ik} \leq 1$ holds for all components.

The similarity between a query and a document is calculated using the conventional vector product, i.e. $sim(Q_j, D_i) = \sum_{k=1}^{t} q_{jk} \cdot a_{ik}$. In this context searching the most relevant documents with respect to a given query Q_j means to search for the documents with the highest values for $sim(Q_j, D_i)$.

Usually such queries are performed employing inverted files. Here an inverted file is maintained for each term in the vocabulary. The file for term k ($1 \leq k \leq t$) contains one entry for each document i with $a_{ik} > 0$. Based on this access structure the algorithm of Buckley and Lewit performs a similarity query scanning the lists for terms with $q_{jk} > 0$. More precisely the algorithm proceeds as follows:

The algorithm employs two auxiliary data structures: (1) A *TopDocs* array with $n+1$ cells maintaining the $n+1$ best matching documents according to the actual state of the algorithm. In *TopDocs* the elements are sorted in descending order with respect to their similarity to Q_j. (2) A Set *ADS* maintaining entries for the documents for which at least one component of an inverted file has been considered during the algorithm. For these documents the set *ADS* maintains the actual subtotal of $\sum_{k=1}^{t} q_{jk} \cdot a_{ik}$ resulting from the inverted file elements considered up to now.

Based on these auxiliary data structures the inverted files are processed one after the other, starting with the file for the term k_1 with the highest q_{jk_1} value.

Then all entries of this inverted file for term k_1 are processed. To this end, for each document i for which there is an entry in this list (i.e. $a_{ik_1} > 0$), a corresponding entry is inserted into *ADS* and *TopDocs* is updated accordingly. Thereafter the inverted file for the term k_2 with the second highest value q_{jk_2} is processed in the same way. The only difference is that documents considered with this second inverted file might also have been considered with the first file. In this case the actual subtotal for the similarity maintained in ADS and *TopDocs* (i.e. $q_{jk_1} \cdot a_{ik_1}$) has to be increased to $q_{jk_1} \cdot a_{ik_1} + q_{jk_2} \cdot a_{ik_2}$. In addition, the ordering of the elements in *TopDocs* has to be updated accordingly. Thereafter the inverted files for the remaining query terms (i.e. terms with $q_{jk} > 0$) are processed in decreasing order of the q_{jk} values.

Let T_p denote the set with the indices of the query terms for which the corresponding inverted files have been considered at a given time during the processing of the algorithm and let $T_{\bar{p}}$ denote the set with the indices of the query terms for which the corresponding inverted files have not yet been considered. Further assume that the n most relevant documents are required. In this case we could stop the algorithm as soon as $\sum_{k \in T_{\bar{p}}} q_{jk} \cdot 1$ is smaller than the smallest difference between two entries in *TopDocs*. This is because 1 is an upper bound for the a_{ik} and in this situation the positions of the upper n entries in *TopDocs*

cannot change due to the additional consideration of the inverted files for the query terms in $T_{\bar{p}}$.

To reduce the number of the inverted files which have to be considered Buckley and Lewit propose to stop the algorithm as soon as $\sum_{k \in T_{\bar{p}}} q_{jk} \cdot 1 \leq TopDocs[\lceil \alpha \cdot n \rceil] - TopDocs[n+1]$. This condition assures that a portion of size α of the n documents in the top positions of the array *TopDocs* surly belongs to the n best matching documents. For the other documents among the n documents at the top positions of the array *TopDocs* we do not know whether they really belong to the n best matching documents. Assume for example $\alpha = 0.25$ and $n = 10$. In this case $TopDocs[\lceil \alpha \cdot n \rceil] - TopDocs[n+1]$ simplifies to $TopDocs[3] - TopDocs[11]$ and represents the difference with respect to the similarity known so far for the 3^{rd} and the 11^{th} document in *TopDocs*. Since this different is at least as high as the upper bound for the similarity not yet considered, at least the top 25 % of the delivered result really belong to the correct result.

In experimental tests, Buckley and Lewit demonstrated that this inaccuracy does only induce marginal deterioration with respect to the retrieval quality — expressed in terms of recall and precision as usual in information retrieval. At the same time the number of page accesses can be reduced significantly (up to 30 %).

In section 5 we will show that the same performance savings are achievable when applying a similar approach to multi-dimensional access structures performing similarity queries. The adaptation of the algorithm to such access structures will be presented in section 4. Beforehand we give a short overview of the LSD^h-tree in order to describe the algorithm as precise as possible.

3 The LSD^h-Tree

In the following we sketch the basic concepts of the LSD^h-tree (section 3.1) and the exact algorithm for similarity queries (section 3.2).

3.1 Basic Idea of the LSD^h-Tree

As usual for multi-dimensional access structures the LSD^h-tree[1] 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*.

Figure 2 illustrates the creation of an LSD^h-tree. In this example we assume that a bucket can hold two objects. Initially, the whole data space corresponds to one bucket. After two insertions the initial bucket has been filled, and an attempt to insert a third object causes the need for a bucket split. To this purpose, a

[1] LSD stands for "Local Split Decision" in this context and the supplement "h" denotes a variant of the original LSD-tree [12] adapted for the use with high-dimensional feature vectors [10].

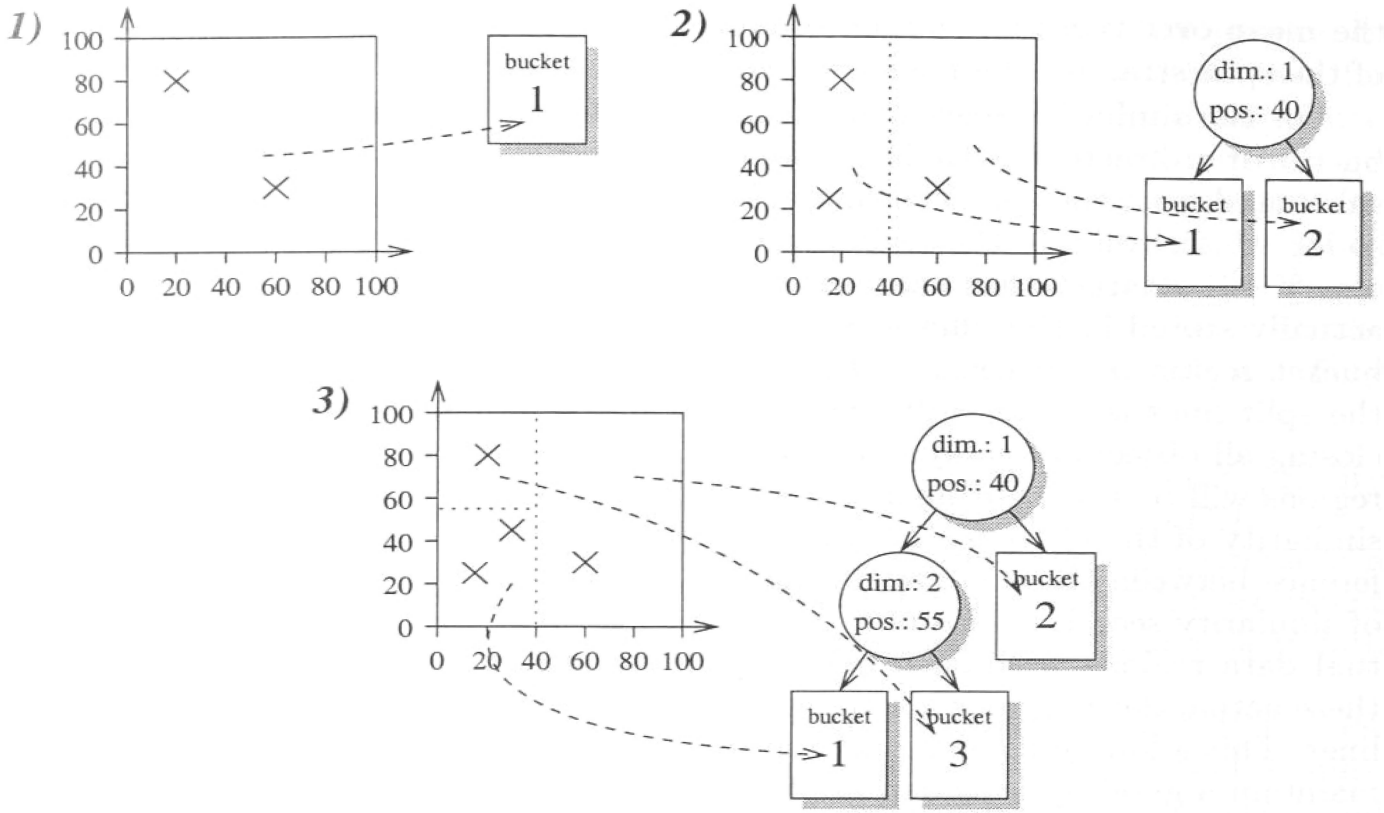

Fig. 2. The creation of an LSD-tree

split line is determined by a so-called split strategy based on the objects in the overrun bucket or on some statistical assumptions [11]. In our example the split is performed in dimension 1 at position 40. The objects on the left side of the split line remain in the old bucket, while those on the right side are stored in a new bucket. The split is represented by a directory node containing the split dimension and the split position. Thereafter the new object can be stored in bucket 1. With the next insertion 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 [12].

An important part of the insertion algorithm of the LSD^h-tree is the split strategy which determines the split position and the split dimension in case of a bucket split. For high-dimensional feature vectors a good choice is a strategy employing the dimension with the highest variance for the object coordinates in the bucket to be split. For this dimension the split position can be calculated as

the mean over these object coordinates. For a detailed discussion of the choice of the split strategy the reader is refered to [11,10] [2].

For the similarity search the bucket region or data region associated with a bucket or a directory node is important. With the LSD^h-tree this region can be calculated from the split lines maintained in the nodes of the directory traversed so far. Unfortunately the region calculated this way is usually larger than necessary — i.e. larger than the minimum bounding rectangle enclosing all objects actually stored in the bucket or in the corresponding subtree. For example the bucket region of bucket 1 in the third step depicted in figure 2 derived from the split lines is $[0, 40] \times [0, 55]$, whereas the minimum bounding rectangle enclosing all objects actually stored in the bucket is $[15, 30] \times [25, 45]$. Since these regions will be used during a similarity search to derive an upper bound for the similarity of the objects stored in the corresponding bucket or subtree, the difference between these rectangles has a significant influence on the performance of similarity search operations. Therefore the LSD^h-tree maintains so-called actual data regions with each reference in the directory. To spare storage space these actual-data regions are coded relative to the regions derived from the split lines. This allows for a compact representation of the actual data regions which maintain a good approximation of the minimum bounding rectangles [10].

3.2 Exact Similarity Scan Algorithm

The algorithm to perform similarity queries can be best described as a similarity scan for the maintained objects. The algorithm delivers all objects maintained in the access structure sorted in descending order with respect to their similarity to the given query object o_{query}. This process can be stopped after each delivered object. Usually it will be stopped as soon as a given number of objects (say n) has been received or as soon as the similarity of the delivered objects falls below a given threshold.

In the following we assume that there is a function $\mathcal{S}(o_1, o_2)$ calculating the similarity between the objects o_1 and o_2. Furthermore we assume that high values of $\mathcal{S}$ stand for a high degree of similarity [3]. In addition we need a function $\mathcal{S}_{max}(o, w)$ which determines an upper bound for the similartiy between object o and all objects potentially stored in the bucket or directory subtree w of an LSD^h-tree without actually accessing these objects. Consequently $\mathcal{S}_{max}$ has to be calculated based on the corresponding data region or actual data region.

[2] Please note that the split process in high dimensional index structures must not be mistaken with semantic clustering approaches as presented in [17]. In contrast to these approaches the split process in a high dimensional index structure is triggered by a bucket overflow and aims for a 50:50-distribution of the objects in the bucket to be split. Of course the objects in both resulting buckets should be as homogeneous as possible, but the main objective is a 50:50-distribution of the objects.

[3] For metrics for which low values represent high similarities – such as the Euclidean distance – we can either apply a corresponding conversion (like $\frac{1}{1+D(o_1,o_2)}$) or adopt the algorithm accordingly.

Applying these functions the algorithm is based on two priority queues. In the first queue called NPQ (= node priority queue) entries are maintained for subtrees and/or buckets of the LSD^h-tree which have not yet been considered during the similarity scan. In this priority queue NPQ the bucket or directory subtree w with the highest potential similarity — i.e. with the highest value for $\mathcal{S}_{max}(o_{\text{query}}, w)$ — has highest priority. In the second priority queue called OPQ (= object priority queue) objects taken from the buckets of the LSD^h-tree are maintained. Here the object o with the highest similarity — i.e. with the highest value for $\mathcal{S}(o_{\text{query}}, o)$ — has highest priority. Based on these auxiliary data structures the algorithm proceeds as follows:

1. To initialize the algorithm NPQ and OPQ are created as empty priority queues. Then the root node of the LSD^h-tree is inserted into NPQ.
2. Then the first element is taken from NPQ. If this element represents a bucket, all objects stored in this bucket are inserted into OPQ. If the element represents a directory node, both sons are inserted into the auxiliary data structure NPQ.
3. Now the algorithm can deliver elements from OPQ as long as OPQ is not empty and their similarity to o_{query} is at least as high as the potential similarity $\mathcal{S}_{max}(o_{\text{query}}, w)$ of the first element in NPQ. The elements taken from OPQ are sorted correctly because parts of the LSD^h-tree which have not yet been considered cannot contain objects with a higher similarity.
   At first glance this 3^{rd} step of the algorithm might seem necessary only if a bucket has been processed in step 2. However, due to the use of actual data regions, the replacement of a directory node with its both sons in NPQ can reduce the value $\mathcal{S}_{max}(o_{\text{query}}, w)$ for the first element in NPQ. In this case additional elements from OPQ could become deliverable.
4. When the first element in OPQ has a similarity value $\mathcal{S}(o_{\text{query}}, o)$ smaller than the potential similarity $\mathcal{S}_{max}(o_{\text{query}}, w)$ of the first element in NPQ, additional parts of the LSD^h-tree have to be considered before delivering the next element, because other parts of the tree might contain more similar objects. To this end, we continue with step 2 of the algorithm.

In order to state this algorithm in pseudo-code, we define the operations for the priority queues OPQ and NPQ. We assume that the priority queues can manage point objects as well as axis-parallel rectangles (i.e. data regions):

$q :=$ **CreatePQ**(o_{query}) creates a priority queue q in which the highest priority is given by the highest similarity to the object o_{query} (defined by $\mathcal{S}$ for point objects and by $\mathcal{S}_{max}$ for data regions).
PQInsert(q, x) inserts the object or subtree x into q.
$x :=$ **PQFirst**(q) assigns the object or subtree with highest priority in q to x.
PQDeleteFirst(q) deletes the object or subtree with highest priority from q.
IsEmptyPQ(q) checks if q is empty.

Figure 3 summarizes the explained similarity scan algorithm. The individual result objects are returned with the operation **Deliver**(o). The following aspects are noteworthy with respect to figure 3:

- The part of the algorithm denoted with (a) manages the elements taken from NPQ. For a directory node both sons are inserted into NPQ and for a bucket the objects stored in the bucket are inserted into OPQ.
- When we have processed an element from NPQ, we have to recalculate the highest potential similarity of an object not yet inserted into OPQ — the corresponding location in the program is marked with (b). To this end, we determine $\mathcal{S}_{max}$ for the first element in NPQ. If NPQ is empty all objects maintained in the LSD^h-tree have been inserted into OPQ and we can set the threshold value $\bar{\mathcal{S}}$ to zero.
- Now we can deliver elements from OPQ as long as their similarity is higher than the threshold value $\bar{\mathcal{S}}$ — the corresponding location in the program is marked with (c).

4 Relaxed Similarity Scan Algorithm

Now that we have introduced the algorithm of Buckley and Lewit in section 2 and the exact similarity scan algorithm in section 3.2 we can describe the adaptation of the idea of a relaxed similarity scan to multi-dimensional access structures. The key idea is that only a given portion — say α — of the result is required to be surely correct. More precisely this means that if we are searching for the n most similar objects we require only $\lceil \alpha \cdot n \rceil$ of the n returned objects to be amongst the mathematically correct n most similar objects. The remaining objects in the result are subject to a "best effort" semantics.

Let us denote the number of objects already delivered by the algorithm with $c-1$ — i.e. if we consider a next element taken from OPQ this would be element number c delivered by the algorithm. Using this notion we can describe the adaptations for the relaxed similarity scan: We have to replace the condition **IF** $\mathcal{S}(o, o_{\text{query}}) \geq \bar{\mathcal{S}}$ **THEN** of the exact similarity scan algorithm with a condition comparing $\bar{\mathcal{S}}$ with the element with the $\lceil \alpha \cdot c \rceil$ highest similarity of an object delivered so far.

This assures that at each time during the query process at least a portion of α of the delivered elements is correct whereas the other elements might not be contained in a correct answer of the same size.

To realize the relaxed algorithm we have to access the element with the $\lceil \alpha \cdot c \rceil$ highest similarity delivered up to now. To this end, we use an additional list called *TopSims*, which maintains the similarity values of the objects taken from OPQ sorted in descending order. On this list two operations are defined: ListInsert(TopSims,v) inserts the value v into the list and restores the ordering in the list. Of course, duplicate values are allowed in TopSims. ListDelete(TopSims,v) removes one entry with value v from the list.

Using this additional list which maintains as much numbers as elements are delivered, we can define our relaxed similarity scan in figure 4.

The differences compared to the exact algorithm are marked with a "▷". First there are the TopSims list and a variable c containing the actual number

```
FUNCTION SimScan (o_query, T);
{ delivers all objects stored in the LSD^h-tree with directory T, sorted in
  descending order according to their similarity to object o_query }
BEGIN
    CreatePQ(NPQ, o_query);
        { auxiliary data structure for the directory nodes and buckets to be
          examined later }
    CreatePQ(OPQ, o_query);
        { auxiliary data structure for the objects }
    w := root(T);  PQInsert(NPQ, w);
    REPEAT
        w := PQFirst(NPQ);  PQDeleteFirst(NPQ);
(a)     IF w is not a bucket THEN
            { insert both suns into NPQ }
            PQInsert(NPQ, w_r);  PQInsert(NPQ, w_l);
        ELSE
            { w is a bucket }
            FOR EACH o ∈ w DO PQInsert(OPQ, o) END;
        END;
        { calculate a new border S̄ for the similarity of objects not yet inserted
          into OPQ }
(b)     IF IsEmptyPQ(NPQ) THEN
            S̄ := 0;
        ELSE
            w_tmp := PQFirst(NPQ);  S̄ := S_max(w_tmp, o_query);
        END;
        done := false;
(c)     WHILE ¬done ∧ ¬IsEmptyPQ(OPQ) DO
            o := PQFirst(OPQ);
            IF S(o, o_query) ≥ S̄ THEN
                PQDeleteFirst(OPQ);  Deliver(o);
            ELSE
                done := true;
            END;
        END;
    UNTIL IsEmptyPQ(NPQ);
END SimScan;
```

Fig. 3. Algorithm for an exact similarity scan

of elements in TopSims. When an object is taken from OPQ the corresponding similarity value is inserted into TopSims on trial [4]. Thereafter the condition **IF** $\mathcal{S}(\text{TopSims}[\lceil c \cdot \alpha \rceil], o_{\text{query}}) \geq \bar{\mathcal{S}}$ **THEN** is used to check if o can be appended to the result queue. If this is not the case the corresponding value has to be

[4] Note that this "insertion on trail" is replaced by a case statement in the real implementation to avoid unnecessary sort operations. However this does only marginally influnece the overall performance of the algorithm.

```
FUNCTION SimScan^r (o_query, T, α) ;
{ delivers all objects stored in the LSD^h-tree with directory T, sortec
  descending order according to their similarity to object o_query; α ∈ ((
  defines the accuracy of the operation }
BEGIN
    CreatePQ(NPQ, o_query);
        { auxiliary data structure for the directory nodes and buckets }
    CreatePQ(OPQ, o_query);
        { auxiliary data structure for the objects }
▷   TopSims := EmptyList();
        { list for the similarity values of the objects taken from OPQ, ma
          taining these values in sorted order }
▷   c := 0;
        { number of of elements in the list TopSims }
    w := root(T);   PQInsert(NPQ, w);
    REPEAT
        w := PQFirst(NPQ);   PQDeleteFirst(NPQ);
        IF w is not a bucket THEN
            { insert both suns into NPQ }
            PQInsert(NPQ, w_r);   PQInsert(NPQ, w_l);
        ELSE
            { w is a bucket }
            FOR EACH o ∈ w DO PQInsert(OPQ, o) END;
        END;
        { calculate a new border S̄ for the similarity of objects not yet inser
          into OPQ }
        IF IsEmptyPQ(NPQ) THEN
            S̄ := 0;
        ELSE
            w_tmp := PQFirst(NPQ);   S̄ := S_max(w_tmp, o_query);
        END;
        done := false;
        { take objects from OPQ as long as the accuracy is good enough
        WHILE ¬done ∧ ¬IsEmptyPQ(OPQ) DO
            o := PQFirst(OPQ);
▷           ListInsert(TopSims, S(o_query, o)); c := c + 1;
▷           IF S(TopSims[⌈c · α⌉], o_query) ≥ S̄ THEN
                PQDeleteFirst(OPQ);   Deliver(o);
            ELSE
▷               ListDelete(TopSims, S(o_query, o)); c := c − 1;
                done := true;
            END;
        END;
    UNTIL IsEmptyPQ(NPQ);
END SimScan^r;
```

Fig. 4. Algorithm for a distance-scan

removed from the TopSims list and we can stop checking objects from OPQ, because beforehand additional elements from NPQ have to be considered.

Now that we have presented our relaxed algorithm for similarity queries, we can compare this approach to the algorithm of Buckley and Lewit. Due to the use of inverted files in the algorithm of Buckley and Lewit the representation of each object is distributed over the inverted files. As a consequence, the similarity values for the objects calculated after the consideration of a certain number of inverted files are not complete. For all objects only subtotals are known and the question is, whether additional coefficients of the objects which will be accessed in further files can influence the intermediate ranking. In contrast, as soon as we have accessed an entry for an object in a multidimensional access structure, we know its complete similarity value. The problem here is to determine, whether objects stored in not yet considered parts of the access structure might have higher similarity values. Nevertheless, the basic principle is analogous.

5 Experimental Results

To assess the performance improvements and the inaccuracy induced by our approach we used a test set with 69753 photos from the Corel Clipart Gallery and from the internet. For these photos feature vectors with 10 dimensions representing the color distribution according to the Munsell color model [18] have been generated. These feature vectors have been inserted into an LSD^h-tree with 2945 buckets, 116 directory pages and an internal directory with at most 1000 nodes. The bucket size has been 2048 bytes and the directory page size 1024 bytes. We performed test runs with 25 different sample images for which 10, 100 and 1000 similar images were searched. Figure 5 depicts the results of these test runs.

In figure 5 a) and b) the performance improvements are stated. Figure 5 a) gives the number of bucket and directory page accesses needed to perform the queries. For the interpretation of the figures it is useful to remind that the non-relaxed algorithm corresponds to $\alpha = 1$. It becomes obvious that for all values of n significant performance improvements can be achieved especially with $\alpha < 0.4$. For example with $\alpha = 0.3$ we spare 24% of the page accesses for $n = 10$, 27% of the page accesses for $n = 100$ and 24.3% of the page accesses for $n = 1000$. Another important performance measure is the number of elements in the auxiliary data structures OPQ and NPQ. Figure 5 b) demonstrates that similar savings are achievable in this respect.

In figure 5 c) and d) the resulting inaccuracy in the output of our algorithm is addressed. To this end, figure 5 c) states the portion of elements in the result which are not part of the correct result. For example with $\alpha = 0.1$ and $n = 100$ on average 11.04% of the 100 elements in the result do not occur in the correct result. In other terms, nearly 89% of the delivered elements indeed belong to the 100 most similar objects. Figure 5 d) gives the highest relative real rank position of an element ranked under the top n elements with our relaxed algorithm. For example with $\alpha = 0.1$ and $n = 10$ we get a relative real rank position of 32.8% which means that the least similar element under the delivered 10 elements

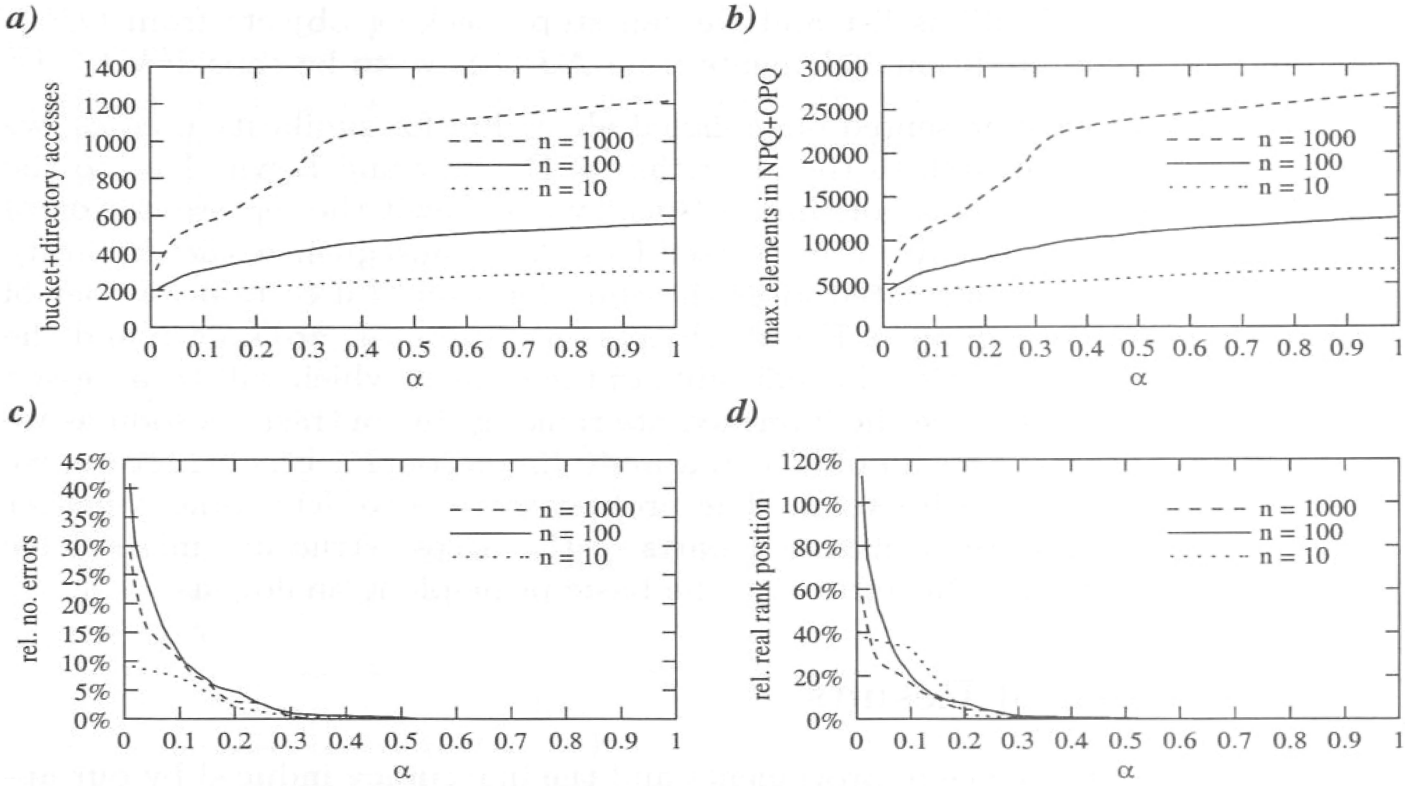

Fig. 5. Experimental results for a data set with 69753 images

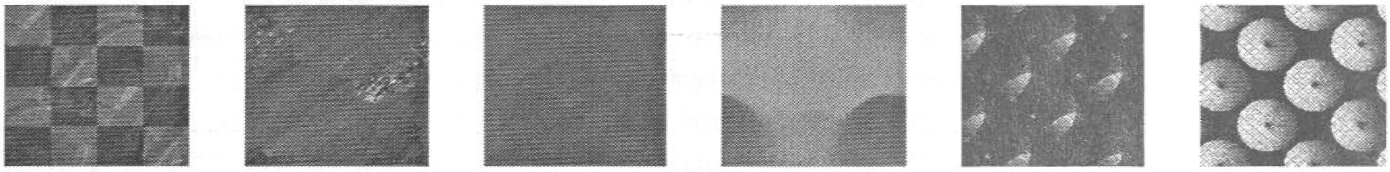

Fig. 6. Six randomly chosen images from our texture collection

actually has rank 13.28 in the correct ranking (on the average). Figure 5 c) and d) impressively show that especially for values $\alpha \geq 0.3$ the inaccuracy in the result is negligible. Hence these experiments show that we can achieve a performance increase of about 25% with only marginal inaccuracies in the result.

To demonstrate that slight inaccuracies in the result ranking are justifiable in many application areas, let us consider a texture similarity example (unfortunately the above example with color similarity is not well suited to a paper printed in black and white). To this end, we inserted 750 images typically used for web page backgrounds into an LSD^h-tree. Figure 6 shows six randomly chosen images from this collection to give an impression of the images. The results achieved for a similarity search on this collection with $n = 9$ and $\alpha = 1.0$, resp. $\alpha = 0.3$, are given in figure 7. The query image was the image ranked first in both results. It turns out that the mathematical inaccuracy induced by our relaxed algorithm is hardly notable for a human.

Finally it should be noted that even for this small texture example with only 750 maintained objects an 17 buckets performance savings similar to the results

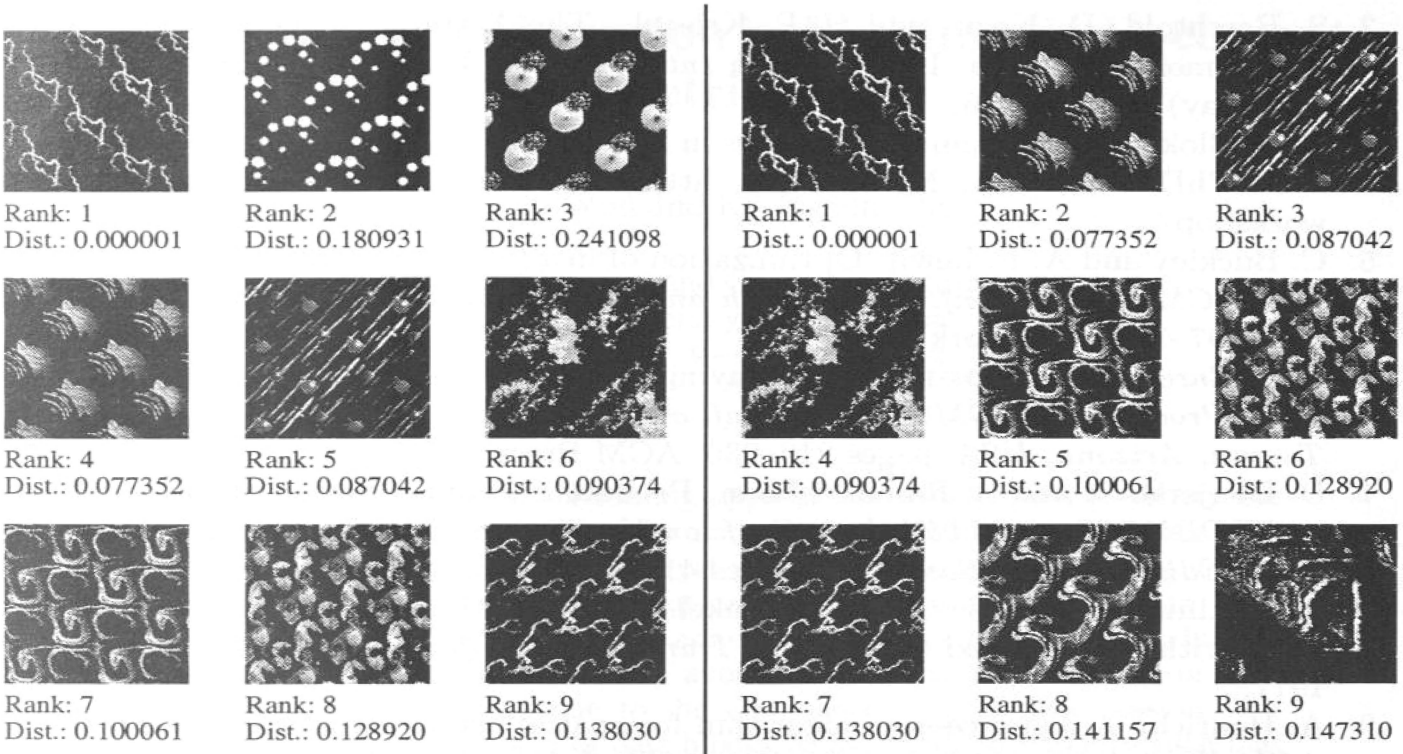

Fig. 7. Results achieved with $\alpha = 0.3$ and $\alpha = 1.0$

given for the color similarity example have been achieved. For example to find the nine most similar images of our above example with $\alpha = 1$ four bucket accesses were needed whereas only three bucket accesses were needed with $\alpha = 0.3$.

6 Conclusion

We have presented a relaxed version of the algorithm for similarity queries which tries to improve the performance significantly and at the same time accepts some moderate inaccuracies in the result. The experimental results show that the relation between the performance improvements and the induced inaccuracies seems rather sensible for many applications, such as image retrieval or other types of similarity queries.

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. S. Berchtold, C. Böhm, D. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. 16th ACM Symposium on Principles of Database Systems*, pages 78–86, Tucson, Arizona, 1997.

3. S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22th Intl. Conf. on VLDB*, pages 28–39, Mumbai (Bombay), India, 1996.
4. H. E. Blok. Top N optimization issues in MM databases. Proceedings of the EDBT 2000 PhD Workshop, Mar. 2000. http://www.edbt2000.uni-konstanz.de/phd-workshop/.
5. C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proc. 8th Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 97–110, New York, 1985.
6. M. J. Carey and D. Kossmann. On Saying "Enough Already!" in SQL. In *SIGMOD 1997, Proc. ACM SIGMOD Intl. Conf. on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 219–230. ACM Press, 1997.
7. D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB'99, Proc. of 25th Intl. Conf. on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 411–422. Morgan Kaufmann, 1999.
8. J. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, 3:209–226, September 1977.
9. 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.
10. A. Henrich. The LSD^h-tree: An access structure for feature vectors. In *Proc. 14th Intl. Conf. on Data Engineering, Orlando, Florida, USA*, pages 362–369, 1998.
11. 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.
12. 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.
13. N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Conf.*, pages 369–380, Tucson, Arizona, USA, 1997.
14. K.-I. Lin, H. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, Oct. 1994.
15. G. Salton. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*. Addison-Wesley, Reading, Mass., USA, 1989.
16. G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.
17. G. Sheikholeslami, W. Chang, and A. Zhang. Semantic clustering and querying on heterogeneous features for visual data. In *Proc. 6th ACM Intl. Conf. on Multimedia (Multimedia-98)*, pages 3–12, N.Y., 1998. ACM Press.
18. J. Sturges and T. Whitfield. Locating basic colours in the munsell space. *Color Research and Application*, 20:364–376, 1995.
19. D. 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.
20. D. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th Intl. Conf. on Data Engineering*, pages 516–523, New Orleans, La., USA, 1996.