Adapting a Spatial Access Structure for Document Representations in Vector Space

Andreas Henrich
Universität Siegen, Praktische Informatik, D-57068 Siegen, Germany
henrich@informatik.uni-siegen.de
http://www.informatik.uni-siegen.de/pi/user/henrich

Abstract

In the field of information-retrieval the vector space model has been proposed. In this model queries and documents are represented as *term vectors* where each coefficient represents the relevance of a given term with respect to the document or query. A typical task in this context is to search for the documents most similar to a given query vector.

On the other hand, algorithms to perform nearest neighbor and distance scan queries have been proposed for various types of spatial access structures. Unfortunately, these access structures assume implicitly that the number of dimensions is relatively small — which is not the case for document representation vectors.

In this paper we discuss the adaptation of spatial access structures for document representation vectors. We describe how some peculiarities of document representation vectors can be exploited to overcome the problems with higher dimensions to a certain extend. We exploit these peculiarities introducing a new cluster split technique and a sophisticated algorithm to calculate an upper bound for the similarity of the documents located in a subtree of the access structure.

1 Introduction

1.1 Vector Space Model

In general IR is concerned with the management of large sets of documents. The most common request in this context is to search for the most relevant documents with respect to a given information need.

To deal with queries of this type, the vector space model has been introduced [Sal89, SWY75]. This model assumes that an available term set (also called vocabulary) is used to identify both maintained documents and information requests. Both queries and documents can then be 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 the coefficients a_{ik} and q_{jk} represent the relevance of term k with respect to document D_i or query Q_j, respectively. 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. In the following, we use the formulas presented in [SB88] which have been proved to be competitive e.g. in [Har95].

Let N denote the number of documents, n_k denote the number of documents containing term k, and tf_{ik} denote the *term frequency* of term k in document D_i. Then the components of a preliminary document representation vector $D_i' = (a_{i1}', a_{i2}', \ldots, a_{it}')$ can be calculated as follows:

$$a_{ik}' = tf_{ik} \cdot \log \frac{N}{n_k} \tag{1}$$

The document representation vector D_i itself is a normalized version of D_i', i.e. $D_i = \frac{D_i'}{\|D_i'\|}$.

If the information need is given as a text document for which similar documents are searched, the following formula can be used to calculate a preliminary query representation vector $Q_j' = (q_{j1}', q_{j2}', \ldots, q_{jt}')$:

$$q_{jk}' = \begin{cases} \left(0.5 + \frac{0.5 \cdot tf_{jk}}{\max\limits_{1 \le l \le t} tf_{jl}}\right) \cdot \log \frac{N}{n_k} & \text{if } tf_{jk} > 0 \\ 0 & \text{if } tf_{jk} = 0 \end{cases} \tag{2}$$

Q_j itself is a normalized version of Q_j', i.e. $Q_j = \frac{Q_j'}{\|Q_j'\|}$.

The similarity between a query and a document can be calculated using the conventional vector product:

$$sim(Q_j, D_i) = \sum_{k=1}^{t} q_{jk} \cdot a_{ik} \tag{3}$$

In this context searching the most relevant documents with respect to a given query Q_j means to search for the documents with high values for $sim(Q_j, D_i)$. More precisely we are looking for the first m documents in a

list where the documents are sorted with respect to the values of $sim(Q_j, D_i)$ in descending order.

Usually such queries are performed physically using algorithms based on inverted lists. An inverted list is maintained for each term in the vocabulary. The list for term k ($1 \leq k \leq t$) contains one entry for each document i with $a_{ik} > 0$. A similarity query is performed scanning the lists for terms with $q_{jk} > 0$ in the order of decreasing values of q_{jk}. An initially empty auxiliary data structure is used to maintain entries for those documents, which might be in the result of the query. When processing an entry (a_{ik}) in the inverted list for term k two cases have to be distinguished: (1) There is no entry for document i in the auxiliary data structure: In this case a reference for document i is inserted into the auxiliary data structure together with a lower bound for its similarity ($a_{ik} \cdot q_{jk}$). (2) There is an entry for document i in the auxiliary data structure: In this case the lower bound for its similarity is increased by $a_{ik} \cdot q_{jk}$.

Furthermore an upper bound for the remaining similarity can be calculated from the q_{jk} values for the terms for which the inverted lists have not yet been considered. Examples for algorithms of this type are e.g. presented in [BL85] and [Luc88].

However, these algorithms are not without problems. On the one hand, the storage requirements are high because there is one list per term and one list entry for each $a_{ik} > 0$. On the other hand, the number of document references, which have to be maintained in the auxiliary data structure is high, especially if q_{jk} is greater than zero for many terms.

1.2 Spatial Access Methods

At least the first drawback of the inverted lists approach does not occur with spatial access structures, which support multi-dimensional similarity searching in the sense of spatial proximity. Here exactly one – of course multidimensional – entry has to be maintained per document. Hence, it seems to suggest itself to use a multidimensional access structure which has originally been developed for spatial applications to maintain term vectors and to support similarity searching on these vectors. However, since various efficient access structures for large sets of spatial objects have been developed in recent years, the question is, which structure should be used as starting point for an adaptation to term vectors.

Like the one-dimensional B-tree most spatial access structures store the objects in buckets of fixed size. When a bucket B overflows, because a new object has to be inserted into B and the bucket capacity is already exhausted, the objects located in B are distributed over two new buckets according to a given split strategy and an entry representing the split decision is inserted into the directory of the access structure. Roughly spoken, three types of spatial access structures can be distinguished:

The first group are hash-based structures. These structures avoid a large directory applying hash techniques. Typical representatives of this group are the grid file [NHS84] or the buddy-tree [SK90]. Unfortunately these structures are relatively rigid in the way buckets have to be split in case of an overflow.

The second type of spatial access structures can be called the R-tree family [Gut84, SRF87, BKSS90]. These access structures provide complete freedom when splitting a bucket, because the split decision in maintained in the directory storing a bounding rectangle for each bucket (or directory page) together with the reference to the bucket (or directory page). However, for the t-dimensional vector space, these bounding rectangles would become t-dimensional bounding intervals which would require disproportional much storage space.

The third group are structures using a generalized k-d-tree as directory [Rob81, OMSD87, Fre87, LS89, HSW89, KO91]. With these structures a bucket split is represented as a split line – i.e. a pair consisting of the split dimension and the split position – in the directory. This seems to be a good compromise for our aspired application area, because the storage space per directory entry is independent of the number of dimensions and there is nevertheless a great freedom when splitting a bucket because the split line can be chosen independent of all other splits in the access structure.

Algorithms to perform nearest neighbor queries — or to be more general, distance scan queries — have been proposed for various types of access structures [Hen94, HS95, RV95]. Given a two- or three-dimensional query point, the proposed algorithms yield the m closest objects in the access structure in sorted order.

Although the basic situation is the same for spatial objects and document representation vectors, there remain some important differences which prevent the straightforward application of spatial access structures for document representation vectors. The first and most important difference is the number of dimensions. Whereas spatial applications are usually concerned with two- or three-dimensional objects, vectors with 1000 and more dimensions are common for the vector space model. Another difference is the similarity measure. Whereas the Euclidean distance is usual in spatial applications a typical similarity measure for document retrieval applying the vector space model is the scalar product.

To our best knowledge, there are only few approaches presented in the database literature to deal with higher dimensions – namely the TV-tree [LJF94] and the SS-tree [WJ96]. These access structures are designed for so-called feature vectors. However, the number of dimensions addressed in these papers (< 100) is by

far less than the number of dimensions induced by the application of the vector space model. Furthermore the presented access structures do not address the special situation with document representation vectors.

In contrast, in this paper we try to overcome the problems with high dimensions exploiting some typical properties of document representation vectors: (1) In document representation vectors most coefficients are zero. (2) Due to normalization all document representation vectors are located on the surface of the t-dimensional unit sphere.

The rest of the paper is organized as follows: Section 2 describes the concepts of a k-d-tree based access structure which are essential for this paper. Section 3 presents our basic approach to overcome at least part of the problems with high dimensions. We call it the *cluster split approach.* Section 4 presents the corresponding split strategy and section 5 presents the adaptations for the distance scan algorithm. Section 6 gives some experimental results and section 7 concludes the paper.

2 Basic concepts of the LSD-Tree

Although the techniques proposed in this paper can be applied to all multidimensional access structures which use a k-d-tree based directory, we will use the LSD-tree[1] as an example for such an access structure in the following in order to be as precise as possible.

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

Initially, the whole data space corresponds to one bucket. After a certain number of insertions the initial bucket has been filled, and an attempt to insert an additional object causes the need for a bucket split. To this purpose, a *split line* is determined and the objects on one side of the split line are stored in one bucket, while those on the other side are stored in another bucket. After some further insertions, the capacity of another bucket will be exceeded. In this case, a split line in the corresponding bucket region is determined, thus splitting this region into two subregions. This process is repeated each time the capacity of a bucket is exceeded.

For example, consider the two-dimensional data space in figure 1. The first split was made according to coordinate 50 in the first dimension, giving buckets 1 and 2. Then bucket 2 was split according to position 60 of the second dimension, and so on.

The split lines are maintained in a directory which is a generalized k-d-tree [Ben75]. For each split, a new

[1] *LSD* stands for *local split decision* in this context

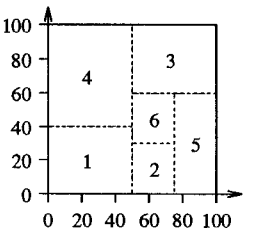

Figure 1: Possible data space partition for an LSD-tree

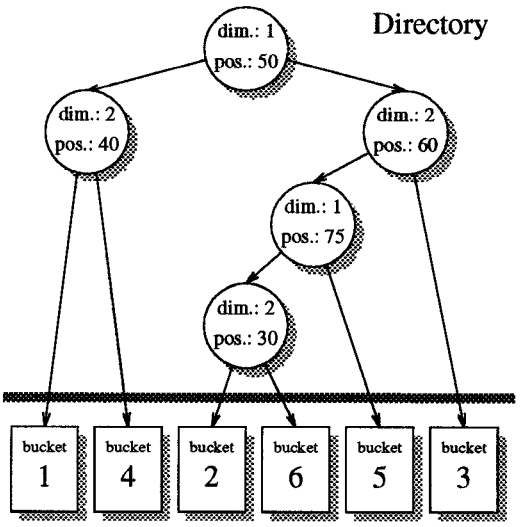

Figure 2: The LSD-tree associated with the data space partition of figure 1

node containing the position and the dimension of the split is inserted into the directory tree.

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 [HSW89].

To describe the processing of a **distance scan query**, we have to introduce the term of a *data region.* The *data region* of a bucket is its bucket region, and the *data region* of a directory node is defined as the union of the two data regions of its sons. As a consequence, a data region in an LSD-tree is always an axis-parallel rectangle.

A distance scan yields the objects stored in the access structure sorted with respect to their distance to a given query point p. In principle the distance scan algorithm works as follows (see [Hen94] for more details):

We start at the root of the directory and search for the bucket for which the bucket region contains the starting

point p of the distance scan. Every time during this search when we follow the left son, we insert the right son into an auxiliary data structure *NPQ* (= node priority queue), where the element with the smallest distance between p and its data region has highest priority, and every time we follow the right son, we insert the left son into *NPQ*.

Then the objects in the found bucket are inserted into another auxiliary data structure *OPQ* (= object priority queue), where the object with the smallest distance to p has highest priority. Thereafter the objects with a distance to p less than or equal to the minimal distance between p and the data region of the first element in *NPQ* are taken from *OPQ*.

Now the directory node or bucket with highest priority is taken from *NPQ*. If it happens to be a directory node, a new search is started choosing always the son on the side of the split line facing p. The other son is inserted into *NPQ*. The bucket determined in this way is processed inserting the objects stored in this bucket into *OPQ* and removing the objects with a distance to p less than or equal to the minimal distance between p and the first element in *NPQ* from *OPQ*. Thereafter the process is continued in the same way until either a given upper bound for the number of the retrieved objects is reached or *NPQ* and *OPQ* are empty.

Analytical considerations presented in detail in [Hen96] which assume that a bucket has a capacity of b objects and that we are concerned with t-dimensional objects show that

$$\kappa = \frac{b \cdot \ln 2}{2} \left\lfloor \left(\left(\frac{2a \cdot \Gamma\left(\frac{t}{2}+1\right)}{\pi^{\frac{t}{2}} \cdot b \cdot \ln 2} \right)^{\frac{1}{t}} + 1 \right)^{t} - \left(\max\left(0, \left(\frac{2a \cdot \Gamma\left(\frac{t}{2}+1\right)}{\pi^{\frac{t}{2}} \cdot b \cdot \ln 2} \right)^{\frac{1}{t}} - 1 \right) \right)^{t} \right\rfloor$$

is a lower bound for the number of objects in *OPQ* after scanning a objects under the assumption of a uniform distribution of the objects over the data space. Hence we have to consider $\kappa + a$ objects if we want to get the a closest objects for a given point p. If we assume $b = 10$ and $a = 50$, we get $\kappa = 41.1$ for $t = 3$, $\kappa = 6418.1$ for $t = 10$ and $\kappa = 576211.3$ for $t = 15$. This stresses the performance problems with high dimensions mentioned in the introduction.

3 Basic Idea of Cluster Splitting

Describing the insertion of objects into an LSD-tree at the beginning of section 2, we mentioned in passing that whenever an attempt to insert an additional object into a bucket causes the need for a bucket split, a *split line* has to be determined. In fact, the choice of an appropriate strategy to compute the split lines is crucial for the performance of the access structure. Such a split strategy has to compute the split dimension and the split position.

The natural aim when determining a split line is to achieve an even distribution of the objects over the two new buckets. Unfortunately this cannot be achieved for a k-d-tree maintaining document representation vectors. The reason is that each coefficient a_{ik} determining the relevance of document i with respect to term k is greater than zero only if term k occurs in document i, and most terms occur only in a very small percentage of the documents. E.g. results presented by Crouch in [Cro90] show that terms occurring in more than 10 % of the documents should not be included in the vocabulary, because they are too general.

Hence, in each dimension of the data space at least 90 % of the coefficients are zero, and the best – in the sense of tree balancing – we can achieve with a split in this situation is a 90:10 distribution of the objects over the new buckets. As a consequence, the resulting directory is extremely unbalanced. The shape of the left directory tree shown in figure 3 depicts this effect for a set of 875 documents maintained in an access structure with a bucket capacity of approximately 17 objects. The number t of terms in the vocabulary is 335.

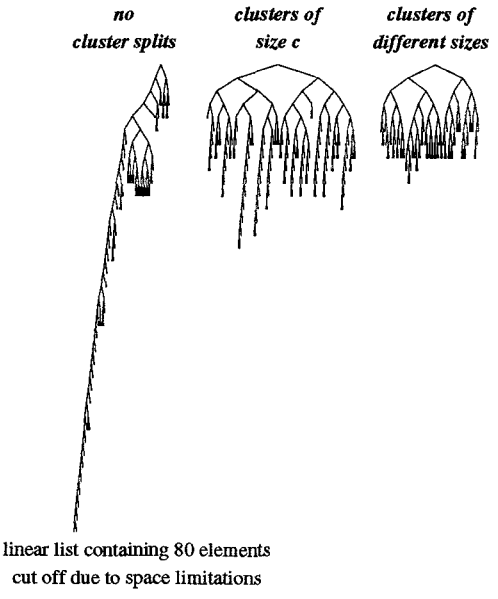

Figure 3: Shape of directory trees achieved with and without cluster splits

To overcome these problems, we propose what we call a *cluster split*. The idea of a cluster split is based on two observations: (1) Since a_{ik} is zero for most documents, the best we can do when splitting in dimension k for the first time, is to separate those documents with $a_{ik} = 0$ from those documents with $a_{ik} > 0$. (2) Since the probability for $a_{is_{dim}}$ to be greater than zero with the first split in dimension s_{dim} is less than 10 %, how can

we achieve an even distribution over the new buckets? The answer is simple, we consider not only dimension s_{dim}, but the dimensions $s_{dim}, s_{dim}+1, \ldots, s_{dim}+(c-1)$ and store an object in the left bucket if

$$\forall k\,(s_{dim} \leq k < \min(s_{dim}+c, t)) : a_{ik} = 0$$

and in the right bucket if

$$\exists k\,(s_{dim} \leq k < \min(s_{dim}+c, t)) : a_{ik} > 0\,.$$

To calculate an optimal value for the cluster size c, we calculate the probability $P(a_{ik} > 0)$ for a coefficient to be greater than zero:

$$P(a_{ik} > 0) = \frac{1}{t}\sum_{k=1}^{t} \frac{n_k}{N}$$

Then the cluster size c should be chosen such that the probability for the coefficients to be zero in c given dimensions is 50 %, i.e. such that $(1 - P(a_{ik} > 0))^c = 0.5$. Taking into consideration that c must be a natural number, this equation can be transformed into

$$c = \left\lfloor \frac{\ln 0.5}{\ln(1 - P(a_{ik} > 0))} \right\rfloor$$

Now we can choose between two types of splits: (1) We can perform a "normal" split in one dimension using the mean over the object coordinates as the split position. (2) We can perform a cluster split.

For our small example data set with 875 documents and a vocabulary with 335 terms, the optimal cluster size c is 84. Applying a cluster split whenever it yields a more even distribution of the objects over the new buckets than the "best" possible normal split yields the directory tree depicted in the middle of figure 3 for our example document set. Obviously, the first splits – which are all cluster splits – yield a good distribution, whereas the splits below level five have the same problems as before and tend to build linear lists.

The shape of the resulting directory tree shows that the problems with uneven splits start as soon as all possible cluster splits are performed. To avoid these problems, cluster splits with clusters of different sizes can be introduced. To this end, we store the applied cluster size together with the split dimension and the split position in the directory and the split strategy determines all three values. The details of this split strategy are presented in the following section. The directory tree achieved for our example data set using this split strategy is shown on the right side of figure 3.

4 Split Strategy

The basic principle of the split strategy is as follows: Assume a bucket capacity of b objects. Let B denote the set containing the b objects stored in the bucket to be split and the object which has to be inserted – i.e. the object which caused the bucket split. Furthermore let $o[k]$ denote the coefficient of an object $o \in B$ in dimension k.

The split strategy considers the following cluster sizes c_i for $i \in \{0, 1, \ldots, 5\}$:

$$c_i = \begin{cases} c & \text{if } i = 0 \\ \lceil \frac{c}{3 \cdot i} \rceil & \text{if } i > 0 \end{cases}$$

We calculate the potential split positions for non-cluster splits $s_{pos}^{(d,\diamond)}$ and for cluster splits $s_{pos}^{(d,c_i)}$ according to the following formula:

$$\begin{array}{lcll} s_{pos}^{(d,\diamond)} & = & \frac{1}{b+1} \cdot \sum_{o \in B} o[d] & \\ s_{pos}^{(d,c_i)} & = & 0 & \text{for } i \in \{0, 1, \ldots, 5\} \end{array}$$

For non-cluster splits $s_{pos}^{(d,\diamond)}$ is calculated for each dimension $d \in \{1, \ldots, t\}$ and for cluster splits $s_{pos}^{(d,c_i)}$ is calculated for each dimension $d \in \{k \mid (\exists j \in \mathbb{N} : k = 1 + jc_i) \wedge (k \leq t)\}$.

To determine the split dimension and the cluster size which should be applied, we have to consider the aim that a split should separate the objects in B in the most homogeneous way.

Let $B_l^{(d,\gamma)}$ with $\gamma \in \{\diamond, c_0, c_1, \ldots, c_5\}$ denote the objects which would be stored in the *left* bucket after a non-cluster split ($\gamma = \diamond$) or a split with cluster size γ in dimension d. Analogously, let $B_r^{(d,\gamma)}$ denote the objects which would be stored in the *right* bucket after a non-cluster or cluster split in dimension d.

Then we can define homogeneity as the most even distribution of the objects over the resulting buckets. One measure in this respect is $\mathcal{H}^{(d,\gamma)} = \min(|B_r^{(d,\gamma)}|, |B_l^{(d,\gamma)}|)$. High values for $\mathcal{H}^{(d,\gamma)}$ stand for high homogeneity. For each potential split position $s_{pos}^{(d,\gamma)}$ the value $\mathcal{H}^{(d,\gamma)}$ is calculated. Then the dimension d and the cluster size γ with the highest value for $\mathcal{H}^{(d,\gamma)}$ are chosen. If there are candidates with equally high values for $\mathcal{H}^{(d,\gamma)}$, larger clusters are preferred and if there are candidates with equally high values for $\mathcal{H}^{(d,\gamma)}$ with the same cluster size, a random choice is made.

5 Distance Scan Adaptation

An adaptation of the distance scan algorithm is necessary because different similarity measures are used in spatial applications and in IR. This affects the priority of the entries in the auxiliary data structures.

For the spatial situation the point with the smallest Euclidean distance to the query point has highest priority in *OPQ*. With document representation vectors, the vector with the highest similarity according to formula (3) has highest priority in *OPQ*. Whereas this change does not cause any problem, the situation with the auxiliary data structure *NPQ* is a little bit more difficult.

In *NPQ* the directory node or bucket which might contain the vector with the highest similarity to the query vector should have highest priority.

Assume the query vector $Q_j = (q_{j1}, q_{j2}, \ldots, q_{jt})$ with $\sum_{k=1}^{t} q_{jk}^2 = 1$ and the data region $R(w) = [w_{\lambda_1}, w_{v_1}] \times [w_{\lambda_2}, w_{v_2}] \times \ldots \times [w_{\lambda_t}, w_{v_t}]$ of a directory node or bucket w [2]. In this case the priority of w in *NPQ* should be

$$SIM\,(Q_j, R(w)) = \max\{sim(Q_j, D_i) \mid D_i \in R(w)\}$$

To calculate the exact upper bound for $SIM\,(Q_j, D(w))$ we have to find a solution $X = (x_1, x_2, \ldots, x_t)$ for the following nonlinear optimization problem:

$$\forall k \in \{1, \ldots, t\}: \; w_{\lambda_k} \leq x_k \leq w_{v_k}$$
$$\sum_{k=1}^{t} {x_k}^2 = 1$$
$$\sum_{k=1}^{t} q_{jk} \cdot x_k = \textbf{Max!}$$

Unfortunately, such a problem would have to be solved every time a new element is inserted into *NPQ* during a distance scan. As a consequence, the usual algorithms to solve such problems cannot be applied, because they are too time-consuming. Therefore, we have to use a heuristics, calculating an upper bound for $SIM\,(Q_j, D(w))$.

Since the distance scan algorithm works correctly for each upper bound of $SIM\,(Q_j, R(w))$ a straightforward choice for the priority would be

$$SIM'\,(Q_j, R(w)) = \sum_{k=1}^{t} q_{jk} \cdot w_{v_k}$$

Unfortunately using $SIM'\,(Q_j, R(w))$ yields a completely unsatisfactory performance since $SIM'\,(Q_j, R(w))$ is greater than 1 in most cases.

One way to improve this bound exploits the fact that all document representation vectors are located on the surface of the t-dimensional unit sphere. Because of $\sum_{i=1}^{t} x_i = 1$ we get $x_k \leq \sqrt{1 - \sum_{i \in \{1,\ldots,t\} \wedge i \neq k} {w_{\lambda_i}}^2}$ since $x_j \geq w_{\lambda_j}$ must hold for all dimensions.

Hence, we can use the following improved upper bound w'_{v_k} instead of w_{v_k}:

$$w'_{v_k} = \min\left(w_{v_k}, \sqrt{1 - \sum_{i \in \{1,\ldots,t\} \wedge i \neq k} {w_{\lambda_i}}^2}\right)$$

and we get an improved version of $SIM'\,(Q_j, D(w))$:

$$SIM'\,(Q_j, D(w)) = \sum_{k=1}^{t} q_{jk} \cdot w'_{v_k}$$

[2]It has to be mentioned, that the lower bound in each dimension is not contained in the data region of a normal LSD-tree, because the objects with $o[k] = s_{dim}$ are stored in the left bucket, for which s_{dim} is the upper bound. However, because of the cluster splits this rule in implicitly relaxed, and objects with $o[k] = s_{dim}$ can be stored in the left and the right subtree.

Another way to yield a bound for $SIM\,(Q_j, R(w))$ exploits that situations where the lower and the upper bound of $R(w)$ in a given dimension are zero occur relatively often due to the use of cluster splits. In such a situation the corresponding coefficient of the document representation vector in w which is most similar to Q_j must be zero too. Hence we can calculate a vector $X' = (x'_1, x'_2, \ldots, x'_t)$ for which the similarity to Q_j is an upper bound for $SIM\,(Q_j, D(w))$.

To yield a vector X' which is most similar to Q_j the relation between the coefficients in X' which are not zero should be the same as the relation between the corresponding coefficients in Q_j. This becomes obvious if we take into consideration that $sim(Q_j, X)$ decreases when the angle between Q_j and X' increases.

When we consider $Q_j = (0.5; 0.331662; 0.8)$ as an example and assume $w_{v_3} = 0$, we can set x'_3 to zero. Then x'_1 and x'_2 can be determined projecting Q_j on the plane spanned by dimensions 1 and 2 and elongating this projected point to the surface of the unit sphere. This is illustrated in figure 4.

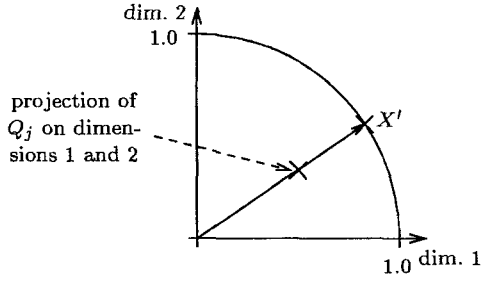

Figure 4: Computation of X'

In general we can calculate x'_i for the dimensions with $w_{v_i} \neq 0$ as $\xi \cdot q_{ji}$, where ξ is the elongating factor used to elongate X' to the surface of the unit sphere. From $\sum_{i \in \{1,\ldots,t\} \wedge w_{v_i} \neq 0} q_{ji} \cdot \xi \cdot q_{ji} = 1$ we get

$$\xi = \frac{1}{\sum_{i \in \{1,\ldots,t\} \wedge w_{v_i} \neq 0} q_{ji}^2}$$

Using this equation we get:

$$x'_i = \begin{cases} \xi \cdot q_{ji} & \text{if} \quad w_{v_i} \neq 0 \\ 0 & \text{if} \quad w_{v_i} = 0 \end{cases}$$

Then we can use

$$\min\left(\sum_{k=1}^{t} q_{jk} \cdot w'_{v_k}, \; sim(Q_j, X')\right)$$

as the priority for the entries in *NPQ*.

6 Experimental Results

To assess the performance of the proposed algorithms, we inserted 9135 verses of a New Testament in German

into the access structure. The corresponding vocabulary contained $t = 1299$ terms. The optimal cluster size c has been 242 for these documents. Using the split strategy proposed in section 4 we achieved an average bucket utilization of 67.3 %. The number of buckets in the access structure has been 1796 with an average of 5.1 objects per bucket. Although there are only 1795 nodes in the directory, due to the cluster splits there are on the average 112.3 splits per term.

We performed similarity queries with four different query texts. For Q_1 one term was set in the query vector, for Q_2 there were three non-zero coefficients in the query vector, for Q_3 five and for Q_4 seven.

In figure 5 the bucket accesses performed during query processing, as well as the current number of entries in *OPQ* and *NPQ* are given in relation to the similarity of the last object taken from *OPQ*. For the correct interpretation of these charts it has to be mentioned, that objects with a similarity of 0.0 are not inserted into *OPQ* and nodes or buckets with an upper bound of 0.0 for the similarity are not inserted into *NPQ*.

Obviously the presented results are not completely satisfactory. Nevertheless, they show that especially documents with a high similarity can be found relatively fast. Furthermore the presented results show, that because of our adaptations exploiting the peculiarities of document representation vectors, the achieved performance is by far better than the analytical results mentioned in section 2 would have suggested.

Though a comparison with experimental results presented for the inverted lists approach e.g. in [BL85] shows that our approach is competitive if $q_{jk} > 0$ holds for at least a couple of terms, there are various other advantages, which would suffice to justify further research in the depicted direction. First, we have one homogeneous access structure containing only one entry per document. This eases and speeds up especially insertions and updates in a dynamic environment. Second, an integration of document representation vectors and standard attributes like the name of the author or the creation date of a document in one multi-attribute access structure becomes possible. Such an integration with standard attributes has already been done for spatial objects [HM95].

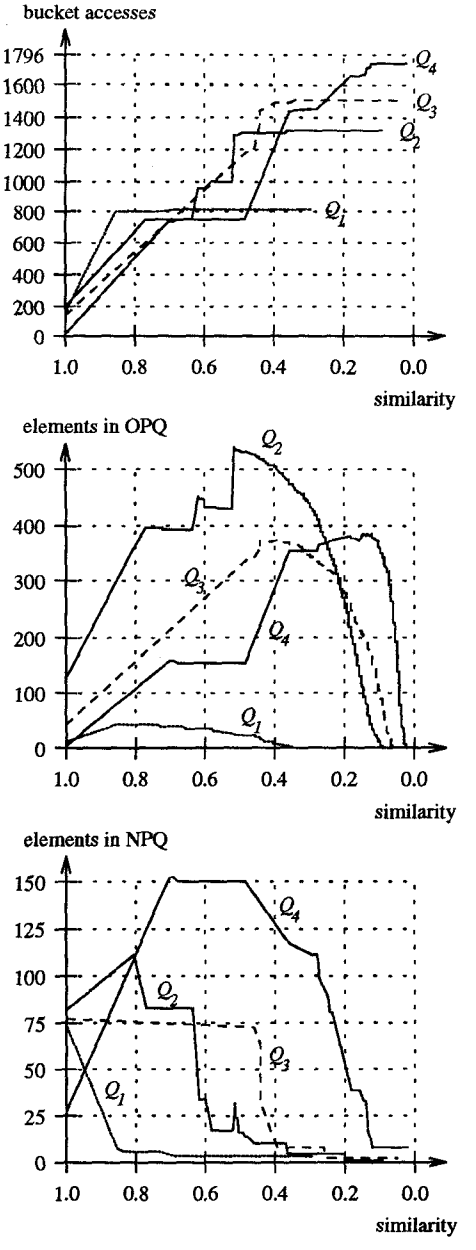

Figure 5: Experimental results for similarity queries

7 Conclusion

We have presented a first approach to use a multidimensional access structure originally designed for spatial objects to maintain document representation vectors. The two main contributions are the cluster split approach which allows to address multiple dimensions in one split and the adaptation of the distance scan algorithm which exploits the fact that all document representation vectors are located on the surface of the t-dimensional unit sphere.

Although the presented experimental results are not completely satisfactory, they give evidence that there is some potential in the presented approach.

One interesting aspect in this respect is the integration of document representation vectors and standard attributes in one multi-attribute access structure. Other directions for future research are the reduction of the number of terms e.g. by synonym substitution or the incorporation of actual data regions in the directory. For the latter approach minimal bounding intervals are stored for selected dimensions with each directory node giving the interval of the data region which is actually

covered by the objects stored in the corresponding subtree of the access structure.

Acknowledgments. We wish to thank our students Susanne Kipping and Marco Müller in the database research group at the University of Siegen for the discussions and for implementing part of the system.

References

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

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

[BL85] C. Buckley and A. Lewit. Optimization of inverted vector searches. In *Proc. of the 8th Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 97–105, New York, USA, 1985. ACM.

[Cro90] J.C. Crouch. An approach to the automatic construction of global thesauri. *Information Processing and Management*, 26(5):629–640, 1990.

[Fre87] M. Freeston. The bang file: a new kind of grid file. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 260–269, San Francisco, Cal., USA, 1987.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, Boston, Mass., USA, 1984.

[Har95] D. Harman. Overview of the second text retrieval conf. (TREC-2). *Information Processing and Management*, 31(3):271–289, 1995.

[Hen94] A. Henrich. A distance-scan algorithm for spatial access structures. In *Proc. of the 2nd ACM Workshop on Advances in Geographic Information Systems (GIS'94)*, pages 136–143, Gaithersburg, Md., USA, 1994.

[Hen96] A. Henrich. Retrieval services for software development environments. Technical report, Universität Siegen, 1996. in German.

[HM95] A. Henrich and J. Möller. Extending a spatial access structure to support additional standard attributes. In *Proc. 4th Intl. Symposium on Advances in Spatial Databases (SSD'95)*, volume 951 of *LNiCS*, pages 132–151. Springer, 1995.

[HS95] G.R. Hjaltason and H. Samet. Ranking in spatial databases. In *Proc. 4th Intl. Symposium on Advances in Spatial Databases (SSD'95)*, volume 951 of *LNiCS*, pages 83–95. Springer, 1995.

[HSW89] 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 Very Large Data Bases*, pages 45–53, Amsterdam, Netherlands, 1989.

[KO91] M.J. van Kreveld and M.H. Overmars. Divided *k-d* trees. *Algorithmica*, 6:840–858, 1991.

[LJF94] K.-I. Lin, H.V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, October 1994.

[LS89] D.B. Lomet and B. Salzberg. A robust multi-attribute search structure. In *Proc. IEEE 5th Intl. Conf. on Data Engineering*, pages 296–304, 1989.

[Luc88] D. Lucarella. A search strategy for large document bases. *Electronic Publishing*, 1(2):105–116, 1988.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[OMSD87] B.C. Ooi, K.J. McDonell, and R. Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. *IEEE COMPSAC*, pages 433–438, 1987.

[Rob81] J.T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1981.

[RV95] N. Roussopoulos and S.K.F. Vincent. Nearest neighbor queries. In *Proc. ACM/SIGMOD Intl. Conf. on Management of Data*, pages 71–79, San Jose, Cal., USA, 1995. ACM Press.

[Sal89] G. Salton. *Automatic Text Processing: the Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, Mass., USA, 1989.

[SB88] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[SK90] B. Seeger and H.-P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proc. 16th Intl. Conf. on Very Large Data Bases*, pages 590–601, Brisbane, Australia, 1990.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: a dynamic index for multi-dimensional objects. In *Proc. 13th Intl. Conf. on Very Large Data Bases*, pages 507–518, 1987.

[SWY75] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, November 1975.

[WJ96] D.A. 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. IEEE.