An Efficient Implementation of the Vector Model in Information Retrieval

© Tomáš Skopal* Pavel Moravec* Michal Krátký* Václav Snášel*
Jaroslav Pokorný**

*Department of Computer Science
VŠB–Technical University of Ostrava, Czech Republic
{tomas.skopal, pavel.moravec}@vsb.cz
{michal.kratky, vaclav.snasel}@vsb.cz

**Department of Software Engineering
Charles University, Prague, Czech Republic
pokorny@ksi.ms.mff.cuni.cz

Abstract

In the area of Information Retrieval, one of the most typical task is to develop methods for querying and retrieving relevant documents from a huge collection of documents. The three decades old vector model, proved to be qualitatively better than the widely used boolean model, is still lacking an efficient implementation which would start its broader usage. In this paper we propose an efficient implementation of vector model, based on metric indexing. This method produces a hierarchical organization of the documents via clusters of related documents. We have chosen the M-tree for this purpose as a suitable indexing structure.

Keywords: vector model, vector query, M-tree, LSI, efficient indexing, random projections

1 Introduction

In the area of information retrieval [2, 12, 17], one of the most typical task is to develop methods for querying and retrieving relevant documents from a huge collection of documents. The area of text retrieval [2] specializes this task into the context of *text documents*. The demands for efficient implementation of various text retrieval methods are high and still increasing with the boom of many applications managing e.g. the digital libraries, internet collections (web pages and sites), editorial texts, newspaper articles, etc.

The *effectiveness* (in the sense of retrieval quality) of various text retrieval systems (TRSs) is usually measured using two parameters, the *precision* P and *recall* R. Given a collection C of documents, a query Q on C, and the size $|A_Q|$ of the answer $A_Q \subset C$ (retrieved by processing the query Q on C), then

$$P = \frac{|R_Q|}{|A_Q|}, \qquad R = \frac{|R_Q|}{|R_C|}$$

where $|R_Q|$ is the number of *relevant documents* (it means documents relevant to the query Q in the collection C) in the answer (i.e. $R_Q \subset A_Q$) and $|R_C|$ is the number of *relevant documents* (relevant to the query Q) in the whole collection C ($R_C \subset C$). In other words, the precision measures the proportion of relevant documents in the query answer A_Q, while the recall measures the proportion of all relevant documents in the query answer A_Q. The relevancy of a document to a query is determined explicitly, usually by human classification. For these purposes, editions of classified document collections are being published periodically, e.g. the TREC collections [15]. For an effective TRS, both parameters, the precision and the recall are close to 1 (according to most queries). The P and R are usually negatively correlated, i.e. when the TRS is tuned to achieve higher P, the R decreases and vice versa.

In the text retrieval models, the *efficiency* (in the sense of system performance or response time) is often regarded as unimportant or secondary. However, in the real-world applications, the efficiency is much more important than the effectiveness since the user needs to obtain an answer quickly (or in a reasonable time). Even the most effective system is useless if it is inefficient, thus an indexing method is needed to achieve an acceptable efficiency. The efficiency can be further specified as *indexing* and *retrieval* efficiency. The indexing efficiency is the time needed for index construction or index update while the retrieval efficiency is the time needed for answering the queries issued by users.

For internet-based TRSs managing huge collections of documents (which are our major subject to research) there is a strong requirement for fast retrieval since thousands of users might want to retrieve an information at a moment. In such TRSs, the needs for fast indexing are much less important than the needs for fast retrieval since the document collection increase is much lower than the total number of queries issued by all the users (within a given time period). For that reason, we deal with the retrieval efficiency in our approach.

Among all of the text retrieval models developed so far, two of them (and their modifications) are still

Proceedings of the 5th Russian Conference on Digital Libraries RCDL2003, St.-Petersburg, Russia, 2003

dominant – the *boolean model* and the *vector model*. In the boolean model, the document is represented by a set of terms (a term is the basic lexical unit of the text, e.g. a word of natural language) and the queries are based on boolean expressions. The boolean model is easy to implement using inverted files where the indexing and querying is very efficient. However, the querying in the boolean model is very simple and tightly related to the terms, not to the underlying *semantics* of the document. In particular, for retrieving relevant documents in the boolean model we must know which terms should (or should not) the document contain, but this is usually the information we do not know in advance. The drawbacks of the boolean model are reflected by the lower values of precision and recall parameters when compared to the vector model. However, the majority of current well-known commercial TRSs managing huge collections of documents (like the internet searching engines *Google*, *Yahoo*, *AltaVista*, *MSN*, etc.) exploit the boolean model for one simple reason – for the vector model there still does not exist an efficient implementation. The effectiveness evaluation of the vector model and the boolean model is well-known, we refer to e.g. [17].

In this paper we introduce an efficient method for vector model implementation based on metric indexing. In the next subsection, basic concepts of the vector model are mentioned. In the Section 2, existing approaches to the vector model implementation are discussed. The new method of metric indexing is presented in the Section 3. Experimental results are presented in Section 4. Finally, the Section 5 concludes the paper and gives an outlook to the future.

1.1 Vector Model

In the vector model, each document D_i ($0 \leq i \leq m$, $m = |C|$) in the collection C is characterized with a single vector d_i where each coordinate d_{ij} of the vector is associated with term t_j from the set of all unique terms over C ($0 \leq j \leq n$, where n is the number of terms). The value of a vector coordinate d_{ij} is a real number $w_{ij} \geq 0$ representing the *weight* (or significance) of the j-th term to the i-th document[1]. From this point of view, the documents can be represented as points within an n-dimensional vector space. Hence, in the vector model, the collection of documents can be represented using an $n \times m$ *term-document matrix* A. For an example of a term-document matrix see Figure 1.

There are many ways how to compute the term weights w_{ij} stored in A. One of the most popular way is based on *inverse document frequency* (IDF) of a term t_j in C,

$$IDF_j = log(\frac{m}{DF_j})$$

where DF_j is the number of documents containing the term t_j. In other words, IDF_j for a term t_j decreases with increasing number of documents in which t_j is occuring and vice versa. In fact, the IDF_j classifies the significance of a term t_j in C – the interpretation is intuitive: If t_j is occuring in the majority of documents then it cannot differentiate among them and thus IDF_j is low. For example, the English terms "the", "and", "then" are usually meaningless (in the classic vector model). On the other side, if t_j is occuring in only few documents or even in a single document, the IDF_j is high.

[1] In the simple model, $w_{ij} = 1$ means that the term is present (at least once) in the document while for $w_{ij} = 0$ it is not. The integer weight can be also treated as a frequency of the term in a document.

document / term \	D_1	D_2	D_3	D_4	D_5
database	0	0.48	0.05	0	0.70
vector	0.23	0	0.23	0	0
index	0.43	0	0	0	0
image	0	0	0.10	0	0.54
compression	0	0	0	0	0.21
multimedia	0.12	0.52	0.62	0	0
metric	0	0	0.32	0.40	0
space	0.42	0	0	0.24	0

Figure 1: Term-document matrix A containing five 8-dimensional document vectors (columns).

Weights w_{ij} are then computed as

$$w_{ij} = TF_{ij} * IDF_j$$

where TF_{ij} is the frequency of term t_j in document D_i. The weight is product of IDF_j (a term global qualitative factor) and TF_{ij} (a term local quantitative factor). The interpretation of TF_{ij} says that a document with multiple occurences of t_j is more appropriate to the semantics of t_j and thus the weight must be higher.

The most important problem about the vector model is the querying mechanism that searches the matrix A according to the query and returns relevant documents. The query is represented also with a vector (the same way as a document is represented) and we want to return the most similar (relevant respectively) documents to the query vector (query document respectively). For this purpose a similarity measure must be defined assigning a similarity value to each pair of query and document vectors (q, d_i) where q is the query vector and d_i is the document vector. In the context of text retrieval the *cosine measure*

$$\mathrm{SIM}_{cos}(q, d_i) = \frac{\sum_{k=1}^{n} q_k * w_{ik}}{\sqrt{(\sum_{k=1}^{n} q_k)^2 * (\sum_{k=1}^{n} w_{ik})^2}}$$

is widely used for its best properties. By the query processing, the columns of the matrix A (document vectors) are compared using the cosine measure with the query vector and sufficiently similar documents are returned as a result.

2 Existing Approaches

Theoretically, the vector model is elegant and clear. For a small collection (say up to 1000 documents) and a small set of unique terms (say up to 1000 terms), the implementation of the term-document

matrix as well as the query processing can be realized by simple methods, sequentially scanning the matrix. However, the problem arises when the number of terms and documents is huge, say 1,000,000 documents and 100,000 terms. In such case, the matrix would have 10^{10} of cells and its storage would take over 350 GB (assuming 4 bytes for a cell). The term-document matrix is fortunately sparse, approximately up to 1% of cells is non-null in real-world collections, thus the matrix storage size can be reduced by a factor near to 100. However, this is only a slight optimization since 3.5 GB matrix is still hard to manage. To keep the vector model viable even for such huge collections it is necessary to develop a fundamentally efficient vector model implementation, significantly reducing the amount of data needed to be processed.

2.1 Sequential Scan

The basic straightforward implementation for querying is the sequential column-scan over the whole matrix A. For each column (document vector) of the matrix the similarity measure is computed against the query vector and based on the returned value the appropriate document is or is not added to the result. As mentioned in the example above, to evaluate a single query means, in fact, read the 3.5 GB matrix and call the similarity function one million times. For most current commercial TRSs dealing with the vector model, the sequential approach is sufficient due to the small sizes of document collections they manage.

2.1.1 Dimensionality Reduction

A way how to reduce the size of the term-document matrix is the dimensionality reduction, i.e. reduction of the set of terms or replacement of the large set of terms by a smaller set of *semantic concepts*. This can be achieved by several approaches:

- Subjectively – using human decision on which terms are not important, these terms are further ignored.
- Heuristically – using stemming techniques reducing a family of various forms of the same word into single stem or using thesauri.
- Statistically – using statistical methods that search in the collection for semantic concepts common to multiple documents in the collection. These concepts (their number is far less than the number of terms) are then used for the matrix construction instead of the original terms. The reduced document vectors consisting of concept weights instead of term weights are often called *pseudo-document vectors* [7].

Nowadays, the statistical methods represent a powerful solution because of their ability to discover latent semantics inside the documents (realized through the semantic concepts instead of terms). The methods of latent semantic indexing (LSI) [7, 3] exploit some constructions well-known in the area of matrix analysis and linear algebra. Specifically, the term-document matrix is decomposed using SVD (singular-value decomposition) and the resulted decomposition is used for creation of a lower-dimensional *concept-document* matrix, based on the most significant singular values. A particular complication in usage of LSI is its computational complexity, but the index (concept-document matrix respectively) construction time is considered to be not so important since the query response time is usually the "bottleneck" of majority TRSs. In practical usage, the LSI can reduce the dimensionality of the original space even 1000 times while the precision and recall parameters are similar or even better when compared to the original vector model [7].

Another and computationally much cheaper possibility is a random projection [4, 1] of the high-dimensional term-document matrix to a lower-dimensional concept-document matrix (here the used concepts are not so sophisticated as by LSI). In practice, using random projections we can reduce the dimensionality up to 100 times while the precision and recall parameters stay over 90% of the precision and recall of the original vector model [4].

2.1.2 Summary

Sequential scan over the term-document matrix (even in compact form) is advantageous only if the matrix is small. The methods of statistical dimensionality reduction (LSI, random projections) are profitable mainly from the semantic point of view – they qualitatively improve the original vector model which is reflected in the higher precision and recall values. On the other side, the dimensionality reduction does not imply the overall storage reduction since the new concept-document matrix is not sparse yet. In the case of random projections, storage of the concept-document matrix can be even much larger than storage of the original sparse term-document matrix in compact form (e.g. in the CCS format [9]). Nowadays, the query processing over the concept-document matrix is still realized using the sequential scan, so the retrieval inefficiency remains.

2.2 Signature Methods

Signature files are a popular filtering method in the boolean model [10]. However, there were only few attempts to use them in the vector model, because their usage is not so straightforward due to the term weights.

A *signature* is a bit vector of F bits, where F is called the *signature length*. The bits are used to record the presence of terms in a document. Each term can be encoded into a signature using a hashing function setting m bits to 1. Number of bits set to 1 is called the *signature weight*.

Signature of a document is created from signatures of terms either by chaining them – which leads to long signatures – or by ORing them. The latter technique, called *superimposed coding*, is widely used. Query signature is generated similarly by superimposing signatures of terms contained in query.

When a conjunctive query in the boolean model is evaluated, query signature S_Q is compared with documents' signatures S_{D_i}. A match with document D_i is found if and only if $S_{D_i} \wedge S_Q = S_Q$.

Because the hashing function and the superimposing are not uniquely invertible functions, some signatures can be matched even they are not relevant to the query. They are referred to as *false drops*. The probability of a false drop is an important factor for signature files, because it is quite expensive to eliminate false drop – selected documents must be searched by a different method to do so. One possible solution is a segmentation of document into several blocks of given length where every block has its own superimposed signature. In this case, we must process the query for every term contained separately. The result is obtained by intersection of these semi-results.

The major problem concerning signatures in the vector model is, that not all the words used in a query have to appear in relevant document. This means, that condition $S_{D_i} \wedge S_Q = S_Q$ is not usable, because relevant documents would be filtered out.

One possibility is to expect that document signatures must match at least $k\%$ of query signatures. However, the relevant documents could be still filtered out. The number of retrieved non-relevant documents is increasing rapidly when we decrease the number of common words needed.

2.2.1 Weight-Partitioned Signature Files

Weight-partitioned signature files (WPSF), proposed by Lee and Ren [13], try to record the term weights in so-called TF-groups. For the best performance we need to separate term frequencies in the current document (TF) and in the whole collection (IDF). For every TF a separate signature file (TF-group) is created. Document is retrieved by searching query terms signatures in every signature file. The length of a signature, weight of a term signature, and a maximal number of terms per signature are precomputed for every signature file based on available data or expected TF distribution to minimize the effect of false drops. The index is searched from the group with highest TF – which ensures that all relevant documents are retrieved – or vice versa. The similarity is directly computed and gives us approximate results (slightly higher than the real ones in the case of false drops) even without additional similarity computation. This result can be either used directly or the distance can be recomputed for selected vectors.

Lee and Ren used in their proposal classical sequential signature file, which caused excessive search of signature file. We have proposed a speedup to this method recently [14] – the S-trees [8], a modification of B^+-tree, minimizing signature weights in nodes by insertion into node with least weight increase. The search continues only in branches matching the queried signature and it can stop before reaching leaf nodes, in case that no signature matches the query.

2.2.2 VA-file

VA-file [5] is sometimes marked as a signature method, while sometimes only as a signature-like. It creates short bit-strings of the length l for every dimension by segmenting it into 2^l intervals and taking interval number as the signature. Space is segmented into small cells (like in the case of grid files) by concatenating these bit-strings into "signatures". The search is executed on "signatures" first (all signatures are scanned sequentially), the filtered documents are searched by a classical method in the second phase.

2.2.3 Summary

Classical signatures are hardly usable with the vector model because of the term weights. They do not guarantee finding all the relevant documents as they do in the boolean model. The concept of Weight-partitioned signature files offers a better solution, but it works well only with discrete weights and has another drawback – all signature files must be searched for terms with zero or one appearance in document, which is true for the majority of queried terms.

In case of VA-files, the "signatures" are scanned sequentially, so the efficiency has to be improved. VA-files are also hardly usable with text data – the length of the "signature" would be too long. Hence, some kind of dimensionality reduction should be introduced.

3 Metric Indexing

In our approach, we wanted to develop a method based on an assumption that the documents (as points in the high-dimensional space) are distributed in many *natural clusters* (according to the cosine measure) within the vector space. We can imagine such a cluster as a major topic common to all the documents in the cluster, thus semantically related documents (their vectors respectively) should reside inside a single cluster. This intuitive idea will work only in case that these clusters really exist – this is a data-specific factor (e.g. randomly generated vectors certainly do not form any clusters).

After we know that the documents in a certain collection are used to be distributed in clusters, we can reuse an existing data structure designed for multidimensional indexing [20] (i.e. structure allowing efficiently store and retrieve high-dimensional vector datasets). This structure will transform the implicit clusters into a "real form" and will create a cluster hierarchy to speedup the query processing. Let us emphasize that a well-built cluster hierarchy is the key objective when creating an efficient index. Furthermore, the clusters should be disjunctive or, at least, their total overlap should be minimized. Such a hierarchical index allows to prune irrelevant documents during the retrieval, thus only a small part of the index needs to be processed.

3.1 M-tree

As a promising indexing structure, we have chosen the M-tree (introduced in [6], elaborated in [16] and revisited in [18]) which is a dynamic data structure for indexing objects of metric datasets. The structure of M-tree was primarily designed for multimedia databases to natively support similarity queries.

Let us have a metric space $\mathcal{M} = (\mathcal{D}, d)$ where $\mathcal{D}$ is a domain of objects $O_i \in \mathcal{D}$ and d is a function measuring distance between two objects. An object O_i is a sequence of features extracted from an original database object. In our case, the object O_i is the (pseudo-)document vector of D_i. The function d must be a metric, i.e. d must satisfy the following metric axioms (reflexivity, positivity, symmetry, triangular inequality):

$$\begin{array}{rcll} d(O_i, O_i) & = & 0 & \\ d(O_i, O_j) & > & 0 & (O_i \neq O_j) \\ d(O_i, O_j) & = & d(O_j, O_i) & \\ d(O_i, O_j) + d(O_j, O_k) & \geq & d(O_i, O_k) & \end{array}$$

Like other dynamic and persistent trees, the M-tree structure is a balanced hierarchy of nodes. In M-tree the objects are distributed within a hierarchy of *metric regions* (each node represents a single metric region) which can be, in turn, interpreted as a hierarchy of clusters. As usual, the nodes have a fixed capacity and a utilization threshold. The leaf nodes contain entries of the objects themselves (here called the ground objects) while entries representing the metric regions are stored in the inner nodes (the objects here are called the routing objects). For a *ground object* O_i, the entry in a leaf has a format:

$$grnd(O_i) = [O_i, oid(O_i), d(O_i, P(O_i))]$$

where $O_i \in D$ is the object, $oid(O_i)$ is an identifier of the original DB object (stored externally), and $d(O_i, P(O_i))$ is a precomputed distance between O_i and its parent routing object.

For a *routing object* O_j, the entry in an inner node has a format:

$$rout(O_j) = [O_j, ptr(T(O_j)), r(O_j), d(O_j, P(O_j))]$$

where $O_j \in D$ is the object, $ptr(T(O_j))$ is pointer to a covering subtree, $r(O_j)$ is a covering radius, and $d(O_j, P(O_j))$ is a precomputed distance between O_j and its parent routing object (this value is zero for the routing objects stored in the root). The entry of a routing object determines a metric region in space $\mathcal{M}$ where the object O_j is a center of that region and $r(O_j)$ is a radius bounding the region. The precomputed value $d(O_j, P(O_j))$ is redundant and serves for optimizing the algorithms on the M-tree. In Figure 2, a metric region and its appropriate entry $rout(O_j)$ in the M-tree is presented. For the hierarchy of metric regions (routing objects $rout(O)$ respectively) in the M-tree, only one invariant must be satisfied. The invariant can be formulated as follows:

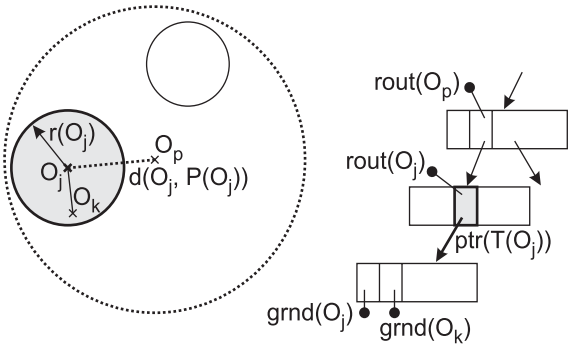

Figure 2: A metric region and its routing object in the M-tree structure.

• All the ground objects stored in the leafs of the covering subtree of $rout(O_j)$ must be spatially located inside the region defined by $rout(O_j)$. •

Formally, having a $rout(O_j)$ then $\forall O \in T(O_j)$, $d(O, O_j) \leq r(O_j)$. If we realize, this invariant is very weak since there can be constructed many M-trees of the same object content but of different structure. The most important consequence is that many regions on the same M-tree level may overlap. An example in Figure 3 shows several objects partitioned into metric regions and the appropriate M-tree. We can see that the regions defined by $rout_1(O_p)$, $rout_1(O_i)$, $rout_1(O_j)$ overlap. Moreover, object O_l is located inside the regions of $rout(O_i)$ and $rout(O_j)$ but it is stored just in the subtree of $rout_1(O_j)$. Similarly, the object O_m is located even in three regions but it is stored just in the subtree of $rout_1(O_p)$.

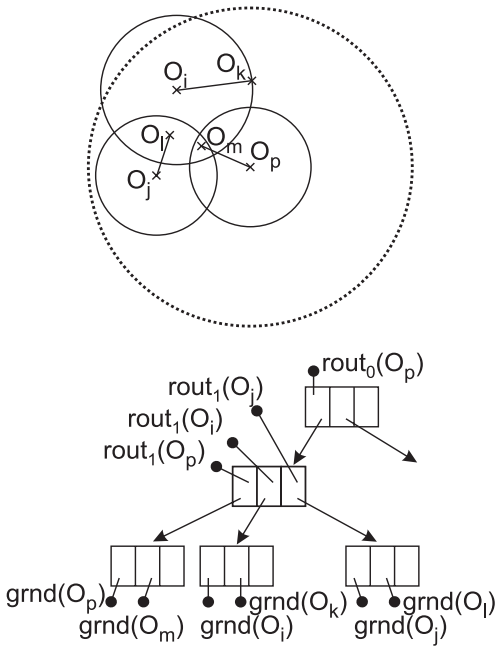

Figure 3: Hierarchy of metric regions and the appropriate M-tree.

3.1.1 Similarity Queries

The structure of M-tree natively supports similarity queries. A similarity measure is here represented by the metric function d. Given a query object O_q, a similarity query returns (in general) objects close to O_q. The similarity queries are of two basic kinds: a *range query* and a *k-nearest neighbours query*.

Range Queries.
A range query is specified as a *query region* given by a query object O_q and a query radius $r(O_q)$. The purpose of a range query is to return all the objects O satisfying $d(O_q, O) \leq r(O_q)$. A query with $r(O_q) = 0$ is called a *point query*.

k-Nearest Neighbours Queries.
A k-nearest neighbours query (k-NN query) is specified by a query object O_q and a number k. A k-NN query returns the first k nearest objects to O_q. Technically, the k-NN query can be implemented using the range query with a dynamic query radius. In practice, the k-NN query is used more often since the size of the query result is known in advance.

By the processing of a range query (k-NN query respectively), the M-tree hierarchy is being passed down. Only if a routing object $rout(O_j)$ (its metric region respectively) intersects the query region, the covering subtree of $rout(O_j)$ is relevant to the query and thus further processed.

3.1.2 Fat-factor

The retrieval efficiency of an M-tree strongly depends on the metric region hierarchy. To maximally prune the irrelevant parts of an M-tree index, every two metric regions on each M-tree level should have minimal (possibly empty) spatial overlap. Such a "horizontally non-overlapping" structure allows to pass only a small number of the M-tree "branches" during the query processing and thus makes the retrieval more efficient.

To classify the amount of the total horizontal overlap we use the fat-factor (introduced in [19], applied to M-tree in [18]). For the fat-factor computation, a point query for each ground object in the M-tree is performed. Let h be the height of an M-tree T, m be the number of ground objects in T, p be the number of nodes in T, and I_c be the total DAC[2] of all the m point queries. Then,

$$fat(T) = \frac{I_c - h \cdot m}{m} \cdot \frac{1}{(p - h)}$$

is the fat-factor of T, a number from interval $\langle 0, 1 \rangle$. For an ideal M-tree, the $fat(T)$ is zero. On the other side, for the worst possible M-tree the $fat(T)$ is equal to one. For an M-tree with $fat(T) = 0$, every performed point query costs h disk accesses while for an M-tree with $fat(T) = 1$, every performed point query costs p disk accesses, i.e. the whole M-tree structure must be passed.

[2] disk access costs, i.e. the number of logical accesses to M-tree nodes (to their disk pages respectively)

3.2 Application of M-tree to the Vector Model

For the vector model, the objects O_i are the document vectors, i.e. columns of either term-document matrix or concept-document matrix. Metric function cannot be directly the cosine measure $\mathrm{SIM}_{cos}(d_1, d_2)$ since it does not satisfy the metric axioms. An appropriate *deviation metric* can be defined as vector deviation

$$d(O_i, O_j) = arccosine(\mathrm{SIM}_{cos}(O_i, O_j))$$

An approximation of the deviation metric can be achieved using the L_2 metric and a projection of the vectors onto the surface of a unitary hyper-sphere. Such a metric could be also expressed using the cosine measure as (see details in [11])

$$d(O_i, O_j) = \sqrt{2 * (1 - \mathrm{SIM}_{cos}(O_i, O_j))}$$

The similarity queries natively supported by the M-tree are exactly the queries required for the vector model. Specifically, the range query will return all the documents that are similar to a query document more than some given threshold (transformed to query radius) while the k-NN query will return the first k most similar documents to the query document.

The benefits of the M-tree to the vector model could be considerable. The term-document matrix (concept-document matrix respectively) is hierarchically indexed according to the semantic clusters hidden in the document collection. The structure of the M-tree is designed to gather the similar documents together into metric regions (i.e. into document clusters). Since the triangular inequality for d is satisfied, many irrelevant document clusters can be safely pruned, thus the retrieval efficiency is improved.

4 Experimental Results

In the preliminary experiments, we have examined the disk access costs (DAC) and the computation costs (CC) for various k-NN queries processing. The DAC is a well-known criterion of the efficiency evaluation while the CC comprises the number of metric (or similarity function) computations needed for an operation on the M-tree (e.g. for a query evaluation). Since we deal with high-dimensional vector spaces the metric computation could be regarded as expensive (the vectors are very long), thus measuring the DAC together with CC is important for the overall efficiency evaluation.

The test collection has included the first 45,000 documents of the Los Angeles Times collection (a part of TREC text collections). For this collection, three matrices were constructed. The classical term-document matrix A, the concept-document matrix obtained by the singular-value decomposition of A, and the concept-document matrix obtained by the random projection of A. The dimensionality of the term-document matrix (i.e. the row count) was near to 90,000. The dimensionality of the concept-document matrices was chosen to 100.

The metric indexing was realized over the above mentioned matrices. The M-tree indexes have contained just the identifiers of the (pseudo-)document vectors (column IDs respectively), thus the indexes were relatively small (up to 1.5MB). The M-tree disk management was separated from the matrix disk management (the implementation of which is out of scope here). The M-trees were constructed using `MinMax`, `Multi-Way`, and slim-down techniques (see details in [18]). The deviation metric (described in Section 3.2) was chosen for the experiments. The sequential file was choosen as a reference index where the DAC as well as the CC were set to 100%.

For the simplicity, each k-NN query was specified directly by the identifier of a randomly chosen indexed document (query document), thus no additional query transformation (i.e. a projection of the query vector to the concept space) was needed. Hence, the first retrieved neighbour in a k-NN query result was the query document itself.

In the presented figures the disk access costs denoted as "M-tree index DAC" represent a size proportion of the M-tree index needed to be processed during the query processing. The disk access costs denoted as "M-tree index seqDAC" represent the same, but the proportion is related to the sequential index size. The latter way is more correct if we wanted to compare the M-tree index efficiency with the sequential index efficiency since the lower values of "M-tree index DAC" are deteriorated by the M-tree storage overhead. Similar notations were chosen to represent the computation costs where, in the case of "M-tree index CC", the proportion is related to the maximal number of objects (both the routing and the ground objects) stored in the M-tree index. In the case of "M-tree index seqCC", the proportion is related to the number of all indexed documents (i.e. to the number of similarity computations needed when querying the sequential index).

In order to ensure the validity of experiments, each particular query evaluation was tested 200 times independently (using 200 different query documents) and the results were averaged.

Some configurations of the M-tree indexes used in experiments are presented in Table 1.

Metric:	deviation
Node capacity:	20
Dimensionality:	10 – 100; 90,000
Documents:	2,500 – 45,000
Tree height:	2 – 3
M-tree index size:	100 kB – 1.5 MB

Table 1. Configurations of the M-tree indexes.

4.1 Vector Model and Random Projections

In the first set of experiments, metric indexing of the classical term-document matrix as well as indexing of the concept-document matrix created by random projection was performed.

Figure 4 shows the disk access costs needed to evaluate a 10-NN query over the term-document matrix in dependence to the number of indexed documents (denoted with prefix "Vector"). It can be observed that for every index size almost the whole M-tree must be passed. If we recalculate these values in order to sequential index size then we will get 120% to 160% of sequential index size to be processed.

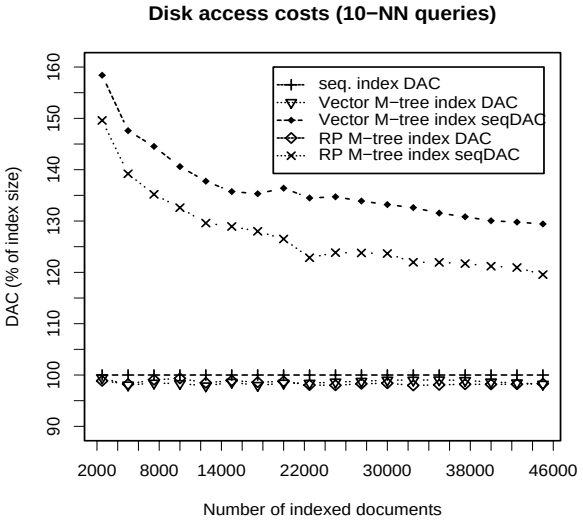

Figure 4: The disk access costs for the term-document matrix and RP concept-document matrix indexes.

Similar results were achieved also in the case of random projections (denoted with prefix "RP").

The computation costs needed for evaluation of a 10-NN query are slightly lower than CC needed when querying the sequential index (see Figure 5).

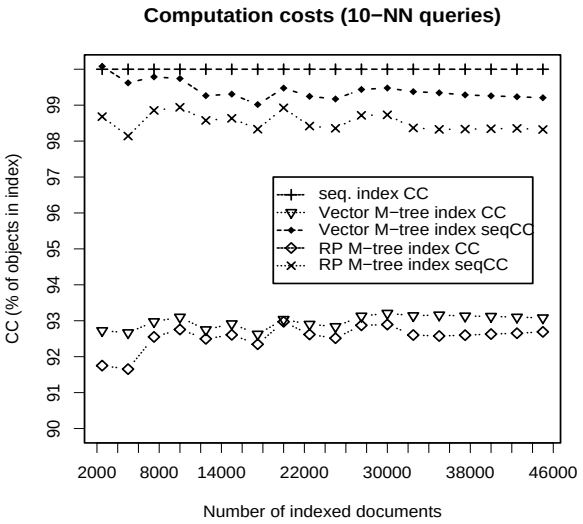

Figure 5: The computation costs for the term-document matrix and RP concept-document matrix indexes.

It is obvious that the results achieved by metric indexing of the term-document matrix as well as the

RP concept-document matrix are insufficient. There are two reasons for such unsatisfactory results. First, the dimensionality 90,000 of the term-document matrix is extremely high. In the area of database indexing the consequences of high dimensionality are called as *curse of dimensionality* [20], a phenomenon that negatively affects the performance of any indexing structure. Second, the (pseudo-)document vectors are not sufficiently clustered (according to the metric) which is the crucial requirement for efficient indexing of high-dimensional datasets. However, there might exist another M-tree configurations (in particular a choice of another metric) for which the indexes over the term-document matrix and/or the RP concept-document matrix will be more efficient. This is also subject of our future research.

4.2 The Efficient Latent Semantic Indexing

In the second set of experiments, metric indexing of the 100-dimensional concept-document matrix created using SVD was performed.

In Figures 6 and 7 the DAC and CC in dependence to the number of indexed documents are presented. We can observe that with the increasing number of indexed documents the costs are decreasing.

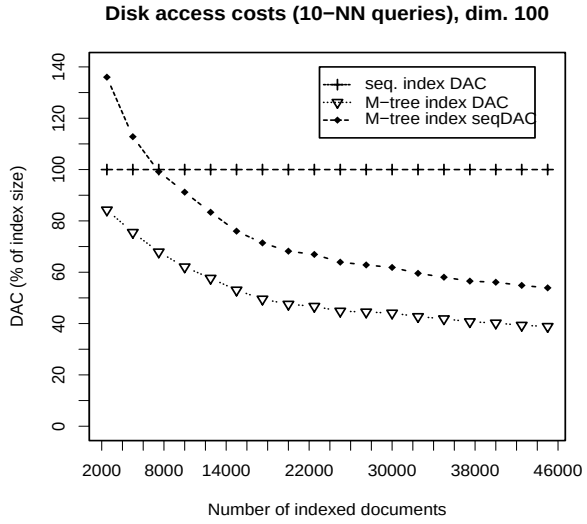

Figure 6: The disk access costs for the SVD concept-document matrix index (in order to increasing number of documents).

This shows that the increasing number of documents forms relatively tight clusters and this is reflected by the lower costs. It is worth to mention that the computation costs needed for evaluation of a 10-NN query are up to four times lower when compared with the sequential index.

On the other side, with the increasing dimensionality the costs are also increasing (see Figures 8 and 9) which is the direct consequence of the dimensionality curse. It is an open question (which we would like to answer in the future) whether the costs

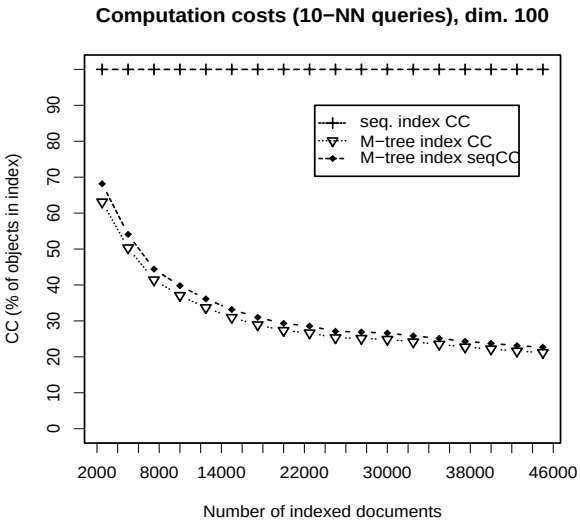

Figure 7: The computation costs for the SVD concept-document matrix index (in order to increasing number of documents).

decrease caused by the increasing number of documents will be faster than the costs increase caused by the increasing dimensionality for even higher dimensionalities and higher numbers of documents.

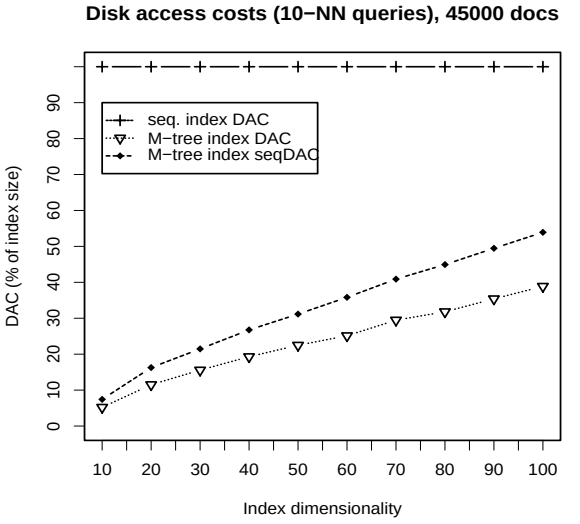

Figure 8: The disk access costs for the SVD concept-document matrix index (in order to increasing dimensionality).

4.2.1 The Choice of Metric

The last experiment shows how the choice of metric for the M-tree could affect the retrieval efficiency. In addition to the M-tree indexes exploiting the deviation metric we have also examined indexes exploiting the approximation metric (in figures denoted as UL2) described in the Section 3.2.

For both M-tree indexes (for the one exploiting

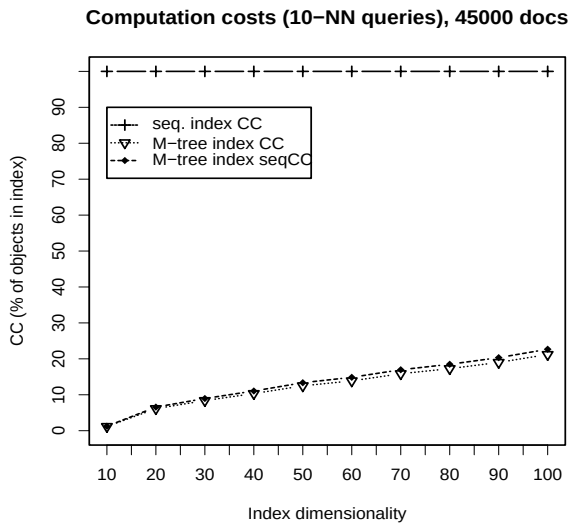

Figure 9: The computation costs for the SVD concept-document matrix index (in order to increasing dimensionality).

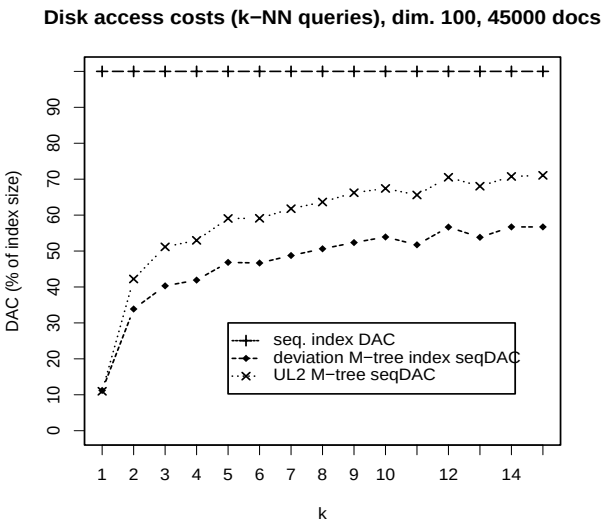

Figure 11: The disk access costs for the SVD concept-document matrix indexes (in order to metric choice and increasing number of nearest neighbours).

the deviation metric as well as for the one exploiting the UL2 metric) the answers to queries were exactly the same. However, the deviation-based M-tree shows up to 15% performance increse over the UL2-based M-tree, see Figure 10. This behaviour can be also observed in Figure 11 where the DAC in dependence to the number of nearest neighbours is depicted.

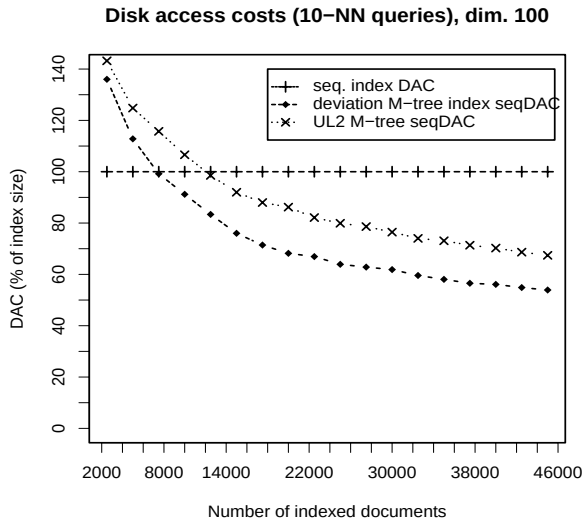

Figure 10: The disk access costs for the SVD concept-document matrix indexes (in order to metric choice and increasing dimensionality).

4.2.2 Technical Limitations

The experiments ran on an Intel Pentium®4, 2.53GHz, 512MB RAM, 120 GB HDD, under Windows XP^{TM} Professional. For the SVD experiments we have used the MatLab®6.5 which stores the matrices in system memory and lacks a persistent matrix storage management. For this reason the size of concept-document matrix was limited to 100×45,000.

In the future we are going to implement our own SVD package supporting persistent matrix management. With this package we will be able to create and index concept-document matrices of much larger size, say up to 500 × 1,000,000.

5 Conclusions And Outlook

In this paper we have proposed an efficient implementation of the vector model for Information Retrieval. For this implementation we have assumed that in a text collection there exist clusters of related documents (related according to the cosine measure). Due to the clusters it was possible to index the (pseudo-)document vectors using the M-tree structure which was designed to index metric datasets.

The experiments have shown that metric indexing is not suitable (at least for this time) for the classic vector model as well as for the random projections. On the other side, our approach seems to be very promising for an efficient implementation of the latent semantic indexing where the document clusters are more distinctive.

In the future we are going to continue the tuning of M-tree configurations, especially the choice of other suitable metrics based on the cosine measure will be fundamental. Anyway, for any further research, experiments with huge document collections must be performed.

References

[1] D. Achlioptas. Database-friendly random projections. In *Symposium on Principles of Database Systems*, 2001.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, New York, 1999.

[3] M. Berry and M. Browne. *Understanding Search Engines, Mathematical Modeling and Text Retrieval.* Siam, 1999.

[4] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Knowledge Discovery and Data Mining*, pages 245–250, 2001.

[5] S. Blott and R. Weber. An Approximation-Based Data Structure for Similarity Search. Technical report, ESPRIT, 1999.

[6] P. Ciaccia, M. Pattela, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of 23rd International Conference on VLDB*, pages 426–435, 1997.

[7] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[8] U. Deppisch. S-tree: A Dynamic Balanced Signature Index for Office Retrieval. In *Proc. of ACM "Research and Development in Information Retrieval"*, pages 77–87, 1986.

[9] I. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, (15):1–14, 1989.

[10] C. Faloutsos. Signature-based text retrieval methods, a survey. *IEEE Computer society Technical Committee on Data Engineering*, 13(1):25–32, 1990.

[11] C. Faloutsos and K.-I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 163–174, San Jose, California, 22–25 1995.

[12] G. Kowalski. *Information Retrieval Systems: Theory and Implementation.* Kluwer Academic Publishers, 1997.

[13] D. L. Lee and L. Ren. Document Ranking on Weight-Partitioned Signature Files. In *ACM TOIS 14*, pages 109–137, 1996.

[14] P. Moravec, J. Pokorný, and V. Snášel. Vector Query with Signature Filtering. In *Proceedings of 6th Bussiness Information Systems Conference, Colorado Springs, USA*, 2003.

[15] NIST. Text REtrieval Conference (TREC), `http://trec.nist.gov/`.

[16] M. Patella. *Similarity Search in Multimedia Databases.* Dipartmento di Elettronica Informatica e Sistemistica, Bologna, 1999.

[17] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill Publications, 1st edition, 1983.

[18] T. Skopal, J. Pokorný, M. Krátký, and V. Snášel. Revisiting M-tree Building Principles. In *ADBIS 2003, accepted, Dresden, Germany*, 2003.

[19] C. Traina Jr., A. Traina, B. Seeger, and C. Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. *Lecture Notes in Computer Science*, 1777, 2000.

[20] C. Yu. *High-Dimensional Indexing.* Springer–Verlag, LNCS 2341, 2002.