Metric Indexing for the Vector Model in Text Retrieval

Tomáš Skopal[1], Pavel Moravec[2], Jaroslav Pokorný[1], and Václav Snášel[2]

[1] Charles University in Prague, Department of Software Engineering,
Malostranské nám. 25, 118 00 Prague, Czech Republic, EU
tomas@skopal.net, jaroslav.pokorny@mff.cuni.cz
[2] VŠB – Technical University of Ostrava, Department of Computer Science,
17. listopadu 15, 708 33 Ostrava, Czech Republic, EU
{pavel.moravec,vaclav.snasel}@vsb.cz

Abstract. In the area of Text Retrieval, processing a query in the vector model has been verified to be qualitatively more effective than searching in the boolean model. However, in case of the classic vector model the current methods of processing many-term queries are inefficient, in case of LSI model there does not exist an efficient method for processing even the few-term queries. In this paper we propose a method of vector query processing based on metric indexing, which is efficient especially for the LSI model. In addition, we propose a concept of approximate semi-metric search, which can further improve the efficiency of retrieval process. Results of experiments made on moderate text collection are included.

1 Introduction

The Text Retrieval (TR) models [4, 3] provide a formal framework for retrieval methods aimed to search huge collections of text documents. The classic vector model as well as its algebraic extension LSI have been proved to be more effective (according to precision/recall measures) than the other existing models[1]. However, current methods of vector query processing are not much efficient for many-term queries, while in the LSI model they are inefficient at all. In this paper we propose a method of vector query processing based on metric indexing, which is highly efficient especially for searching in the LSI model.

1.1 Classic Vector Model

In the classic vector model, each document D_j in a collection C ($0 \leq j \leq m$, $m = |C|$) is characterized by a single vector d_j, where each coordinate of d_j is associated with a term t_i from the set of all unique terms in C ($0 \leq i \leq n$, where n is the number of terms). The value of a vector coordinate is a real number $w_{ij} \geq 0$ representing the *weight* of the i-th term in the j-th document. Hence,

[1] For a comparison over various TR models we refer to [20, 11].

A. Apostolico and M. Melucci (Eds.): SPIRE 2004, LNCS 3246, pp. 183–195, 2004.

a collection of documents can be represented by an $n \times m$ *term-by-document* matrix A. There are many ways how to compute the term weights w_{ij} stored in A. A popular weight construction is computed as $tf * idf$ (see e.g. [4]).

Queries. The most important problem about the vector model is the querying mechanism that searches matrix A with respect to a query, and returns only the relevant document vectors (appropriate documents respectively). The query is represented by a vector q the same way as a document is represented. The goal is to return the most similar (relevant) documents to the query. For this purpose, a similarity function must be defined, assessing a similarity value to each pair of query and document vectors (q, d_j). In the context of TR, the *cosine measure* $\mathrm{SIM}_{cos}(q, d_j) = \frac{\sum_{k=1}^{n} q_k \cdot w_{kj}}{\sqrt{\sum_{k=1}^{n} {q_k}^2 \cdot \sum_{k=1}^{n} {w_{kj}}^2}}$ is widely used. During a query processing, the columns of A (the document vectors) are compared against the query vector using the cosine measure, while the sufficiently similar documents are returned as a result. According to the query extent, we distinguish *range queries* and *k-nearest neighbors (k-NN) queries*. A range query returns documents similar to the query more than a given similarity threshold. A k-NN query returns the k most similar documents.

Generally, there are two ways how to specify a query. First, a *few-term query* is specified by the user using a few terms, while an appropriate vector for such a query is very sparse. Second, a *many-term query* is specified using a text document, thus the appropriate query vector is usually more dense. In this paper we focus just on the many-term queries, since they better satisfy the similarity search paradigm which the vector model should follow.

1.2 LSI Vector Model (Simplified)

Simply said, the LSI (latent semantic indexing) model [11, 4] is an algebraical extension of the classic vector model. First, the term-by-document matrix A is decomposed by singular value decomposition (SVD) as $A = U\Sigma V^T$. The matrix U contains *concept vectors*, where each concept vector is a linear combination of the original terms. The concepts are *meta-terms* (groups of terms) appearing in the original documents. While the term-by-document matrix A stores document vectors, the *concept-by-document* matrix ΣV^T stores *pseudo-document* vectors. Each coordinate of a pseudo-document vector represents a weight of an appropriate concept in a document.

Latent Semantics. The concept vectors are ordered with respect to their significance (appropriate singular values in Σ). Consequently, only a small number of concepts is really significant – these concepts represent (statistically) the main themes present in the collection – let us denote this number as k. The remaining concepts are unimportant (noisy concepts) and can be omitted, thus the dimensionality is reduced from n to k. Finally, we obtain an approximation (rank-k SVD) $A \approx U_k \Sigma_k V_k^T$, where for sufficiently high k the approximation error will

be negligible. Moreover, for a low k the effectiveness can be subjectively even higher (according to the precision/recall values) than for a higher k [3]. When searching in a real-world collection, the optimal k is usually ranged from several tens to several hundreds. Unlike the term-by-document matrix A, the concept-by-document matrix $\Sigma_k V_k^T$ as well as the concept base matrix U are dense.

Queries. Searching for documents in the LSI model is performed the same way as in the classic vector model, the difference is that matrix $\Sigma_k V_k^T$ is searched instead of A. Moreover, the query vector q must be projected into the concept base, i.e. $U_k^T q$ is the *pseudo-query vector* used by LSI. Since the concept vectors of U are dense, a pseudo-query vector is dense as well.

1.3 Vector Query Processing

In this paper we focus on efficiency of vector query processing. More specifically, we can say that a query is processed efficiently in case that only a small proportion of the matrix storage volume is needed to load and process. In this section we outline several existing approaches to the vector query processing.

Document Vector Scanning. The simplest method how to process a query is the sequential scanning of all the document vectors (i.e. the columns of A, $\Sigma_k V_k^T$ respectively). Each document vector is compared against the query vector using the similarity function, while sufficiently similar documents are returned to the user. It is obvious that for any query the whole matrix must be processed. However, sequential processing of the whole matrix is sometimes more efficient (from the disk management point of view) than a random access to a smaller part of the matrix used by some other methods.

Term Vector Filtering. For sparse query vectors (few-term queries respectively), there exists a more efficient scanning method. Instead of the document vectors, the term vectors (i.e. the rows of the matrix) are processed. The cosine measure is computed simultaneously for all the document vectors, "orthogonally" involved in the term vectors. Due to the simultaneous cosine measure evaluation a set of m accumulators (storing the evolving similarities between each document and the query) must be maintained in memory. The advantage of term filtering is that only those term vectors must be scanned, for which the appropriate term weights in the query vector are nonzero. The term vector filtering can be easily provided using an inverted file – as a part of the boolean model implementation [15].

The simple method of term filtering has been improved by an approximate approach [19] reducing the time as well as space costs. Generally, the improvement is based on early termination of query processing, exploiting a restructured inverted file where the term entries are sorted according to the decreasing occurrences of a term in document. Thus, the most relevant documents in each term

entry are processed first. As soon as the first document is found in which the number of term occurrences is less than a given addition threshold, the processing of term entry can stop, because all the remaining documents have the same or less importance as the first rejected document. Since some of the documents are never reached during a query processing, the number of used accumulators can be smaller than m, which saves also the space costs. Another improvement of the inverted file exploiting *quantized weights* was proposed recently [2], even more reducing the search costs.

Despite the above mentioned improvements, the term vector filtering is generally not so much efficient for many-term queries, because the number of filtered term vectors is decreased. Moreover, the term vector filtering is completely useless for the LSI model, since each pseudo-query vector is dense, and none of the term vectors can be skipped.

Signature Methods. Signature files are a popular filtering method in the boolean model [13], however, there were only few attempts made to use them in the vector model. In that case, the usage of signature files is not so straightforward due to the term weights. Weight-partitioned signature files (WPSF) [14] try to solve the problem by recording the term weights in so-called TF-groups. A sequential file organization was chosen for the WPSF which caused excessive search of the signature file. An improvement was proposed recently [16] using the S-trees [12] to speedup the signature file search. Another signature-like approach is the VA-file [6]. In general, usage of the signature methods is still complicated for the vector model, and the results achieved so far are rather poor.

2 Metric Indexing

Since in the vector model the documents are represented as points within an n-dimensional vector space, in our approach we create an index for the term-by-document matrix (for the concept-by-document matrix in case of LSI) based on metric access methods (MAMs) [8]. A property common to all MAMs is that they exploit only a metric function for the indexing. The metric function stands for a similarity function, thus metric access methods provide a natural way for similarity search. Among many of MAMs, we have chosen the M-tree.

2.1 M-Tree

The M-tree [9, 18, 21] is a dynamic data structure designed to index objects of metric datasets. Let us have a metric space $\mathcal{M} = (\mathbb{U}, d)$ where $\mathbb{U}$ is an object universe (usually a vector space), and d is a function measuring distance between two objects in $\mathbb{U}$. The function d must be a metric, i.e. it must satisfy the axioms of reflexivity, positivity, symmetry and triangular inequality. Let $\mathbb{S} \subseteq \mathbb{U}$ be a dataset to be indexed. In case of the vector model in TR, an object $O_i \in \mathbb{S}$ is represented by a (pseudo-)document vector of a document D_i. The particular metric d, replacing the cosine measure, will be introduced in Section 2.2.

Like the other indexing trees based on B^+-tree, the M-tree structure is a balanced hierarchy of nodes. In M-tree the objects are distributed in a hierarchy of *metric regions* (each node represents a single metric region) which can be, in turn, interpreted as a hierarchy of object clusters. The nodes have a fixed capacity and a minimum utilization threshold. The leaf nodes contain *ground entries grnd(O_i)* of the indexed objects themselves, while in the inner nodes the *routing entries rout(O_j)* are stored, representing the metric regions and routing to their covering subtrees. Each routing entry 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. For the hierarchy of metric regions (routing entries $rout(O_j)$ respectively) in the M-tree, the following requirement must be satisfied:

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

The most important consequence of the above requirement is that many regions on the same M-tree level may overlap. An example in Figure 1 shows several objects partitioned among metric regions and the appropriate M-tree. We can see that the regions defined by $rout_1(O_1)$, $rout_1(O_2)$, $rout_1(O_4)$ overlap. Moreover, object O_5 is located inside the regions of $rout_1(O_1)$ and $rout_1(O_4)$ but it is stored just in the subtree of $rout_1(O_4)$. Similarly, the object O_3 is located even in three regions but it is stored just in the subtree of $rout_1(O_2)$.

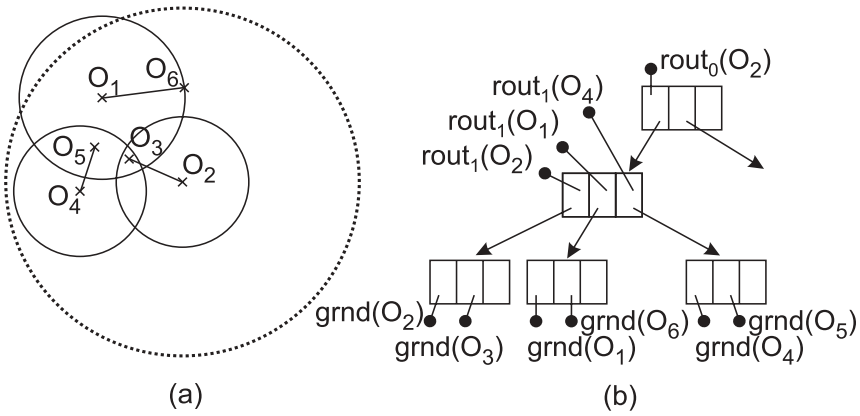

Fig. 1. Hierarchy of metric regions (a) and the appropriate M-tree (b)

Similarity Queries in the M-Tree. The structure of M-tree natively supports similarity queries. The similarity function is represented by the metric function d where the close objects are interpreted as similar.

A range query `RangeQuery(`Q`,`r_Q`)` is specified as a *query region* given by a query object Q and a query radius r_Q. The purpose of a range query is to retrieve all such objects O_i satisfying $d(Q, O_i) \leq r_Q$. A k-nearest neighbours query (k-NN query) `kNNQuery(`Q`,`k`)` is specified by a query object Q and a number k. A k-NN query retrieves the first k nearest objects to Q.

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

2.2 Application of M-Tree in the Vector Model

In the vector model the objects O_i are represented by (pseudo-)document vectors d_i, i.e. by columns of term-by-document or concept-by-document matrix, respectively. We cannot use the cosine measure $\mathrm{SIM}_{cos}(d_i, d_j)$ as a metric function directly, since it does not satisfy the metric axioms. As an appropriate metric, we define the *deviation metric* $d_{dev}(d_i, d_j)$ as a vector deviation

$$d_{dev}(d_i, d_j) = arccos(\mathrm{SIM}_{cos}(d_i, d_j))$$

The similarity queries supported by M-tree (utilizing d_{dev}) are exactly those required for the vector model (utilizing SIM_{cos}). Specifically, the range query will return all the documents that are similar to a query more than some given threshold (transformed to the query radius) while the k-NN query will return the first k most similar (closest respectively) documents to the query.

In the M-tree hierarchy similar documents are clustered among metric regions. Since the triangular inequality for d_{dev} is satisfied, many irrelevant document clusters can be safely pruned during a query processing, thus the search efficiency is improved.

3 Semi-metric Search

In this section we propose the concept of semi-metric search – an approximate extension of metric search applied to M-tree. The semi-metric search provides even more efficient retrieval, considerably resistant to the curse of dimensionality.

3.1 Curse of Dimensionality

The metric indexing itself (as is experimentally verified in Section 4) is beneficial for searching in the LSI model. However, searching in a collection of high-dimensional document vectors of the classic vector model is negatively affected by a phenomenon called *curse of dimensionality* [7, 8]. In the M-tree hierarchy (even the most optimal hierarchy) the curse of dimensionality causes that clusters of high-dimensional vectors are not distinct, which is reflected by huge overlaps among metric regions.

Intrinsic Dimensionality. In the context of metric indexing, the curse of dimensionality can be generalized for general metric spaces. The major condition determining the success of metric access methods is the *intrinsic dimensionality* of the indexed dataset. The intrinsic dimensionality of a metric dataset (one of the interpretations [8]) is defined as

$$\rho = \frac{\mu^2}{2\sigma^2}$$

where μ and σ^2 are the mean and the variance of the dataset's *distance distribution histogram*. In other words, if all pairs of the indexed objects are almost equally distant, then the intrinsic dimensionality is maximal (i.e. the mean is high and/or the variance is low), which means the dataset is poorly intrinsically structured. So far, for datasets of high intrinsic dimensionality there still does not exist an efficient MAM for exact metric search. In case of M-tree, a high intrinsic dimensionality causes that almost all the metric regions overlap each other, and searching in such an M-tree deteriorates to sequential search.

In case of vector datasets, the intrinsic dimensionality negatively depends on the correlations among coordinates of the dataset vectors. The intrinsic dimensionality can reach up to the value of the classic (embedding) dimensionality. For example, for uniformly distributed (i.e. not correlated) n-dimensional vectors the intrinsic dimensionality tends to be maximal, i.e. $\rho \approx n$.

In the following section we propose a concept of semi-metric modifications that decrease the intrinsic dimensionality and, as a consequence, provide a way to efficient approximate similarity search.

3.2 Modification of the Metric

An increase of the variance of distance distribution histogram is a straightforward way how to decrease the intrinsic dimensionality. This can be achieved by a suitable modification of the original metric, preserving the similarity ordering among objects in the query result.

Definition 1. Let us call the *increasing modification* d^f_{dev} of a metric d_{dev} a function

$$d^f_{dev}(O_i, O_j) = f(d_{dev}(O_i, O_j))$$

where $f : \langle 0, \pi \rangle \to R_0^+$ is an increasing function and $f(0) = 0$. For simplicity, let $f(\pi) = 1$.

Definition 2. Let $s : \mathbb{U} \times \mathbb{U} \to R_0^+$ be a similarity function (or a distance function) and $SimOrder_s : \mathbb{U} \to \mathcal{P}(\mathbb{S} \times \mathbb{S})$ be a function defined as

$$\langle O_i, O_j \rangle \in SimOrder_s(Q) \Leftrightarrow s(O_i, Q) < s(O_j, Q)$$

$\forall O_i, O_j \in \mathbb{S}, \forall Q \in \mathbb{U}$. In other words, the function $SimOrder_s$ orders the objects of dataset $\mathbb{S}$ according to the distances to the query object Q.

Proposition. For the metric d_{dev} and every increasing modification d^f_{dev} the following equality holds:

$$SimOrder_{d_{dev}}(Q) = SimOrder_{d^f_{dev}}(Q), \forall Q \in \mathbb{U}$$

Proof:
"$\subset$": The function f is increasing. If for each $O_i, O_j, O_k, O_l \in \mathbb{U}$, $d_{dev}(O_i, O_j) > d_{dev}(O_k, O_l)$ holds, then $f(d_{dev}(O_i, O_j)) > f(d_{dev}(O_k, O_l))$ must also hold.
"$\supset$": The second part of proof is similar. □

As a consequence of the proposition, if we process a query sequentially over the entire dataset $\mathbb{S}$, then it does not matter if we use either d_{dev} or d^f_{dev}, since both of the ways will return the same query result.

If the function f is additionally *subadditive*, i.e. $f(a) + f(b) \geq f(a+b)$, then f is *metric-preserving* [10], i.e. $f(d(O_i, O_j))$ is still metric. More specifically, concave functions are metric-preserving (see Figure 2a), while convex (even partially convex) functions are not – let us call them *metric-violating* functions (see Figure 2b). A metric modified by a metric-violating function f is a *semi-metric*, i.e. a function satisfying all the metric axioms except the triangular inequality.

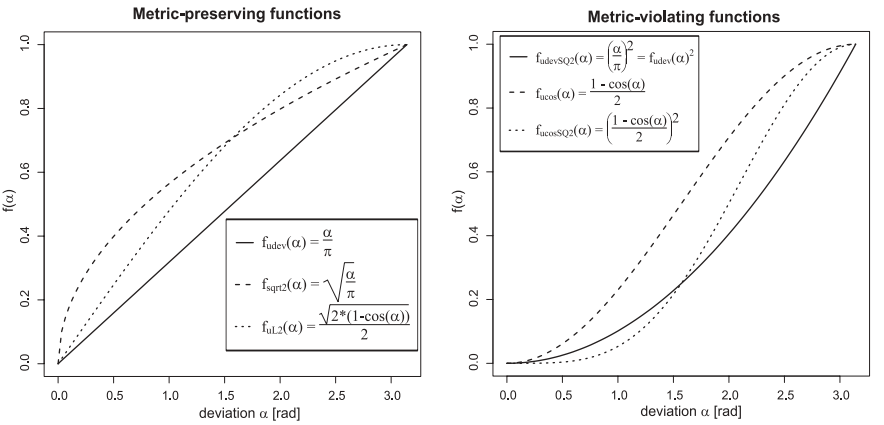

Fig. 2. (a) Metric-preserving functions (b) Metric-violating functions

Clustering Properties. Let us analyze the clustering properties of modifications d^f_{dev} (see also Figure 2). For concave f, two objects close to each other according to d_{dev} are more distant according to d^f_{dev}. Conversely, for convex f, the close objects according to d_{dev} are even closer according to d^f_{dev}. As a consequence, the concave modifications d^f_{dev} have a negative influence on clustering, since the object clusters become indistinct. On the other side, the convex modifications d^f_{dev} even more tighten the object clusters, making the cluster structure of the dataset more evident. Simply, the convex modifications increase the distance histogram variance, thereby decreasing the intrinsic dimensionality.

3.3 Semi-metric Indexing and Search

The increasing modifications d^f_{dev} can be utilized in the M-tree instead of the deviation metric d_{dev}. In case of a semi-metric modification d^f_{dev}, the query processing is more efficient because of smaller overlaps among metric regions in the M-tree. Usage of metric modifications is not beneficial, since their clustering properties are worsen, and the overlaps among metric regions are larger.

Semi-metric Search. A semi-metric modification d^f_{dev} can be used for all operations on the M-tree, i.e. for M-tree building as well as for M-tree searching. With respect to M-tree construction principles (we refer to [21]) and the proposition in Section 3.2, the M-tree hierarchies built either by d or d^f_{dev} are the same. For that reason, an M-tree built using a metric d can be queried using any modification d^f_{dev}. Such *semi-metric queries* must be extended by the function f, which stands for an additional parameter. For a range query the query radius r_Q must be modified to $f(r_Q)$. During a semi-metric query processing, the function f is applied to each value computed using d as well as it is applied to the metric region radii stored in the routing entries.

Error of the Semi-metric Search. Since the semi-metric d^f_{dev} does not satisfy the triangular inequality property, a semi-metric query will return more or less approximate results. Obviously, the error is dependent on the convexity of a modifying function f. As an output error, we define a *normed overlap error*

$$E_{NO} = 1 - \frac{|result_{Mtree} \cap result_{scan}|}{max(|result_{Mtree}|, |result_{scan}|)}$$

where $result_{Mtree}$ is a query result returned by the M-tree (using a semi-metric query), and $result_{scan}$ is a result of the same query returned by sequential search over the entire dataset. The error E_{NO} can be interpreted as a *relative precision* of the M-tree query result with respect to the result of full sequential scan.

Semi-metric Search in Text Retrieval. In the context of TR, the searching is naturally approximate, since precision/recall values do never reach up to 100%. From this point of view, the approximate character of semi-metric search is not a serious limitation – acceptable results can be achieved by choosing such a modifying function f, for which the error E_{NO} will not exceed some small value, e.g. 0.1. On the other side, semi-metric search significantly improves the search efficiency, as it is experimentally verified in the following section.

4 Experimental Results

For the experiments we have chosen the Los Angeles Times collection (a part of TREC 5) consisting of 131,780 newspaper articles. The entire collection contained 240,703 unique terms. As "rich" many-term queries, we have used articles consisting of at least 1000 unique terms. The experiments were focused on disk access costs (DAC) spent during k-NN queries processing. Each k-NN query was repeated for 100 different query documents and the results were averaged. The access to disk was aligned to 512B blocks, considering both access to the M-tree index as well as to the respective matrix. The overall query DAC are presented in megabytes. The entries of M-tree nodes have contained just the document vector identifiers (i.e. pointers to the matrix columns), thus the M-tree storage

volume was minimized. In Table 1 the M-tree configuration used for experiments is presented (for a more detailed description see [21]).

The labels of form `Dev`xxx in the figures below stand for modifying functions f used by semi-metric search. Several functions of form `DevSQ`$p(\alpha) = \left(\frac{\alpha}{\pi}\right)^p$ were chosen. The queries labeled as `Dev` represent the original metric queries presented in Section 2.2.

Table 1. The M-tree configuration

Page size: 512 B; Capacity (leaves: 42, nodes: 21)
Construction: `MinMax` + `SingleWay` + `SlimDown`
Tree height: 4; Avg. util. (leaves: 56%, nodes: 52%)

4.1 Classic Vector Model

First, we performed tests for the classic vector model. The storage of the term-by-document matrix (in CCS format [4]) took 220 MB. The storage of M-tree index was about 4MB (i.e. 1.8% of the matrix storage volume (MSV)).

In Figure 3a the comparison of document vector scanning, term vector filtering as well as metric and semi-metric search is presented. It is obvious that using document vector scanning the whole matrix (i.e. 220 MB DAC) was loaded and processed. Since the query vectors contained many zero weights, the term vector filtering worked more efficiently (76 MB DAC, i.e. 34% of MSV).

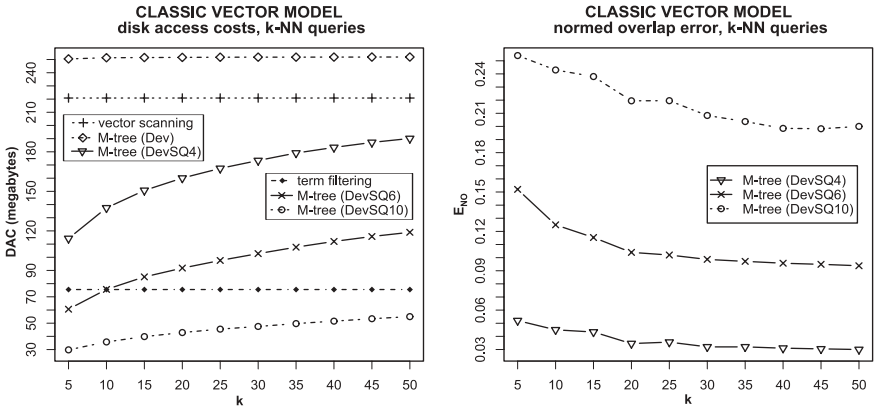

Fig. 3. Classic vector model: (a) Disk access costs (b) E_{NO} error

The metric search `Dev` did not performed well – the curse of dimensionality (n = 240,703) forced almost 100% of the matrix to be processed. The extra 30 MB DAC overhead (beyond the 220 MB of MSV) was caused by the non-sequential access to the matrix columns. On the other side, the semi-metric search performed better. The `DevSQ10` queries for $k = 5$ consumed only 30 MB

DAC (i.e. 13.6% of MSV). Figure 3b shows the normed overlap error E_{NO} of the semi-metric search. For `DevSQ4` queries the error was negligible. The error for `DevSQ6` remained below 0.1 for $k > 35$. The `DevSQ10` queries were affected by a relatively high error from 0.25 to 0.2 (with increasing k).

4.2 LSI Model

The second set of tests was made for the LSI model. The target (reduced) dimensionality was chosen to be 200. The storage of the concept-by-document matrix took 105 MB, while the size of M-tree index was about 3 MB (i.e. 2.9 % of MSV).

Because the size of term-by-document matrix was very large, the direct calculation of SVD was impossible. Therefore, we have used a two-step method [17], which in first step calculates a *random projection* [1, 5] of document vectors into a smaller dimensionality of *pseudo-concepts*. This is done by multiplication of a zero-mean unit-variance random matrix and the term-by-document matrix. Second, a rank-$2k$ SVD is calculated on the resulting *pseudoconcept-by-document* matrix, giving us a very good approximation of the classic rank-k SVD.

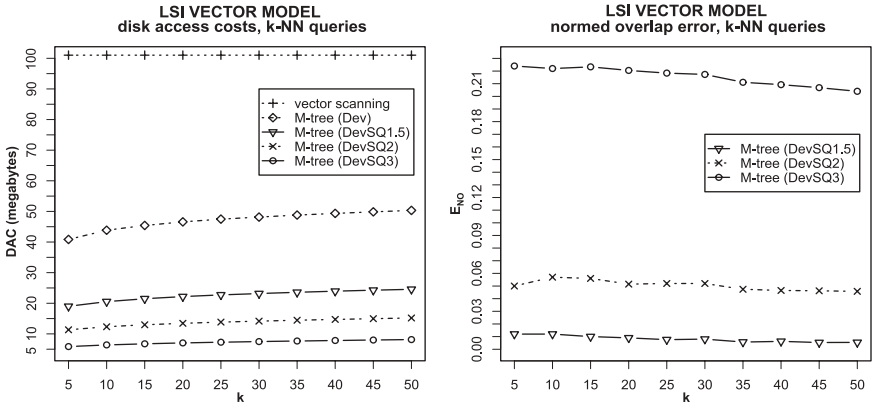

Fig. 4. LSI model: (a) Disk access costs (b) E_{NO} error

The Figure 4a shows that metric search `Dev` itself was more than twice as efficient as the document vector scanning. Even better results were achieved by the semi-metric search. The `DevSQ3` queries for $k = 5$ consumed only 5.8 MB DAC (i.e. 5.5% of MSV). Figure 4b shows the error E_{NO}. For `DevSQ1.5` queries the error was negligible, for `DevSQ2` it remained below 0.06. The `DevSQ3` queries were affected by a relatively high error.

5 Conclusion

In this paper we have proposed a metric indexing method for an efficient search of documents in the vector model. The experiments have shown that metric indexing itself is suitable for an efficient search in the LSI model. Furthermore,

the approximate semi-metric search allows us to provide quite efficient similarity search in the classic vector model, and a remarkably efficient search in the LSI model. The output error of semi-metric search can be effectively tuned by choosing such modifying functions, that preserve an expected accuracy sufficiently.

In the future we would like to compare the semi-metric search with some other methods, in particular with the VA-file (in case of LSI model). We also plan to develop an analytical error model for the semi-metric search in M-tree, allowing to predict and control the output error E_{NO}.

This research has been partially supported by GAČR grant No. 201/00/1031.

References

1. D. Achlioptas. Database-friendly random projections. In *Symposium on Principles of Database Systems*, 2001.
2. V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th annual international ACM SIGIR*, pages 35–42. ACM Press, 2001.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, New York, 1999.
4. M. Berry and M. Browne. *Understanding Search Engines, Mathematical Modeling and Text Retrieval*. Siam, 1999.
5. 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.
6. S. Blott and R. Weber. An Approximation-Based Data Structure for Similarity Search. Technical report, ESPRIT, 1999.
7. C. Böhm, S. Berchtold, and D. Keim. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.
8. E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *Proc. 3rd Workshop on Algorithm Engineering and Experiments (ALENEX'01), LNCS 2153*. Springer-Verlag, 2001.
9. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd Athens Intern. Conf. on VLDB*, pages 426–435. Morgan Kaufmann, 1997.
10. P. Corazza. Introduction to metric-preserving functions. *Amer. Math Monthly*, 104(4):309–23, 1999.
11. 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.
12. U. Deppisch. S-tree: A Dynamic Balanced Signature Index for Office Retrieval. In *Proceedings of ACM SIGIR*, 1986.
13. C. Faloutsos. Signature-based text retrieval methods, a survey. *IEEE Computer society Technical Committee on Data Engineering*, 13(1):25–32, 1990.
14. D. L. Lee and L. Ren. Document Ranking on Weight-Partitioned Signature Files. In *ACM TOIS 14*, pages 109–137, 1996.
15. A. Moffat and J. Zobel. Fast ranking in limited space. In *Proceedings of ICDE 94*, pages 428–437. IEEE Computer Society, 1994.

16. P. Moravec, J. Pokorný, and V. Snášel. Vector Query with Signature Filtering. In *Proc. of the 6th Bussiness Information Systems Conference, USA*, 2003.
17. C. H. Papadimitriou, H. Tamaki, P. Raghavan, and S. Vempala. Latent semantic indexing: A probabilistic analysis. In *Proocedings of the ACM Conference on Principles of Database Systems (PODS), Seattle*, pages 159–168, 1998.
18. M. Patella. *Similarity Search in Multimedia Databases*. Dipartmento di Elettronica Informatica e Sistemistica, Bologna, 1999.
19. M. Persin. Document filtering for fast ranking. In *Proceedings of the 17th annual international ACM SIGIR*, pages 339–348. Springer-Verlag New York, Inc., 1994.
20. G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw Hill Publications, 1st edition, 1983.
21. T. Skopal, J. Pokorný, M. Krátký, and V. Snášel. Revisiting M-tree Building Principles. In *ADBIS 2003, LNCS 2798, Springer, Dresden, Germany*, 2003.