Distributed Information Retrieval using LSI

Markus Watzl and Rade Kutil *

Abstract. *Latent semantic indexing (LSI) is a recently developed method for information retrieval (IR). It is a modification of the usual vector space model, that offers better retrieval performance at the cost of increased computational complexity. This makes LSI a good candidate for high performance computing and the grid. Therefore, methods for using IR in the grid are introduced in this work, including newly developed application interfaces that allow the usage of LSI. However, distributed document sets have separate indices, which possibly decreases the global retrieval performance. It can be shown that it is important for the distributed parts of the document set that similar documents are grouped into the same part. These quality issues are investigated and new approaches to improve the performance are presented.*

1. Introduction

The term information retrieval (IR) is used for information systems where queries are not presented as precise or formal requests and where the system has no additional information about the content of the information source but the information source itself. Prominent examples for IR systems are desktop and internet search engines, while database queries using SQL are an example of information searching that is not IR.

An IR system consists of an indexing function that creates an index out of the documents and a retrieval function that takes a query as input and returns a subset of the documents using the index.

1.1. Vector Space Model

The Vector Space Model is a very basic and wide-spread mechanism for IR. It uses linear algebra to describe the tasks of indexing and querying. The basic principles are:

- We have n documents and a set containing all m different terms $t_1, t_2, \ldots, t_m$ that occur in our set of documents.
- Each document is represented by a **document vector**: a vector $d^{(i)}$, where the value of $d^{(i)}_j$ reflects the importance of the term t_j in $d^{(i)}$.
- The $m \times n$-matrix A consisting of the column-vectors $d^{(1)}, d^{(2)}, \ldots, d^{(n)}$ is the index of the document set. It is called **term-document matrix**.
- A query is similar to a document vector. It contains terms of our term set $t_1, t_2, \ldots, t_m$ with some weighted values representing their importance. It can, therefore, be represented as a query

*Department of Scientific Computing, University of Salzburg, A–5020 Salzburg, Austria

vector q, where q_i is the importance of t_i in the query. As a measure of correspondence between the query and a document vector we could use the angle between their vectors. However, it is computationally less expensive and conceptionally more convenient to compute the cosine between these vectors.

$$\cos \angle(q,d) = \frac{\langle q,d \rangle}{|q|\,|d|}$$

Note that since the cosine is monotonic on $[0,\pi]$, a greater cosine always means a smaller angle.

To build our index matrix, we clearly have to identify the m terms that occur in our n documents. But how do we receive the values that reflect the importance of a term in a document? According to Luhn [9], the frequency of occurrence of a term in a document is an appropriate measure for its importance. So our first step is to count the occurrences of the terms. Then, weighting mechanisms are applied: the absolute frequency of a term in a document is replaced by a relative frequency, which is called **local weighting**, and words that occur very often in our text collection get smaller values which is called **global weighting**.

1.2. Latent Semantic Indexing

Latent semantic indexing (LSI) [8] is a relatively new extension to the vector space model. It can improve the quality of the results, but it also increases the size of the index and it massively increases the indexing complexity.

The idea is to replace the term-document matrix by a lower rank approximation. An accurate explanation of the effectiveness of reducing the rank of the term-document matrix can be found in [6]. We describe documents as combinations of so-called *latent concepts*: in terms of linear algebra, rank reduction means replacing vectors of a matrix by linear combinations of a limited set of vectors. In terms of information retrieval we can see rank reduction as the identification of common concepts and the replacing of vectors by combinations of these concepts.

To receive a lower rank approximation of the matrix, we use the truncated singular value decomposition (SVD) as described in 1.3..

1.3. Singular Value Decomposition

The singular value decomposition (SVD) is a matrix decomposition that is applicable on every matrix containing real or complex values. It splits a matrix A into three matrices U, Σ and V such that $A = U\Sigma V^\top$, where U and V are orthogonal and Σ is a diagonal matrix. For a detailed description of the SVD see [7]. We are interested in the matrix Σ, that has some interesting properties:

- It is a diagonal matrix containing the *singular values* of A in decreasing order along its diagonal.
- The rank of the matrix A is equal to the number of non-zero singular values in Σ.
- The Frobenius norm of the matrix A is equal to the Frobenius norm of the matrix Σ.
- We can build a matrix A_k of rank $k \leq rank_A$ if we compute $A_k = U\Sigma_k V^\top$, where Σ_k is a diagonal matrix containing the k maximum values of Σ in decreasing order along its diagonal.

- A theorem by Eckart and Young [7] states that A_k, as defined above, is the best rank-k approximation in terms of *Frobenius norm distance*.

The SVD has the following properties when used in an information retrieval engine:

- **Synonymy and polysemy**: the automatic transformation of the documents from the term space into a latent concept space is able to link documents which use different words for the same thing (synonymy) and to separate documents who use the same words in differing meanings (polysemy).
- **Computational complexity**: SVD algorithms are computationally quite complex. If we can not use powerful performance optimisations, the complexity of the computation can hardly be handled for large document sets.
- **Loss of sparsity**: generally, term-document matrices are very sparse. According to [5], the ratio of non-zero elements in these matrices is typically no more than 1%, while the U, Σ and V matrices or reduced-rank approximations computed from the results of the SVD have no significant ratio of non-zero elements. Although rank reduction reduces up to 90% of the size of the large U and V matrices, they still remain significantly larger than the original term-document matrix.
- **Rank optimisation**: we know that we can generally improve the searching performance if we reduce the rank of the term-document matrix. However, the algorithms to compute the optimal rank (e.g. as described in [10]) do not seem to provide a reliable estimation in our test situations.

Our IR system supports two implementations of the SVD: The *parallel two-sided block-Jacobi SVD algorithm with dynamic ordering* by Gabriel Oksa [4], which optimised for Grid nodes that are systolic parallel computers and the *xGESDD* algorithm of the LAPACK library [3], which is one of the fastest general purpose algorithms to compute the SVD directly.

1.4. Performance measures

There are two basic performance measures for IR systems:

- The precision reveals the proportion of relevant documents of all documents retrieved.
- The recall reveals the proportion of the retrieved documents of all relevant documents.

Both of these measures return a value between 0 and 1, where 0 means *worst* and 1 means *best*. However, they share the disadvantage that none of them is very useful without the other: If we just return all documents, we will certainly obtain a recall of 1, but nobody will be satisfied by the result. Similarly, we could return no documents at all and get a precision of 1. As one-dimensional measure for query performance, we use the **precision-recall break-even point** in this paper: the (possibly approximated) point where precision and recall are equal.

Another important thing to mention is that we need reference query results to compute our performance measures. Because there is no perfect IR approach, we have to evaluate queries by hand, a procedure that is both time consuming and not necessarily valid: different human searchers will probably deliver different results – these are, after all, two of the reasons why we need powerful IR systems. To cope with these problems, there are some freely available sets of documents, queries and results that are used to compare IR systems. In this paper, we use the *MEDLINE text collection* [1].

2. Data Indexing Interfaces

There are several official efforts to evolve information retrieval on the grid. In this section we present the most important ones and then introduce the features of our new indexing interface.

2.1. OGSA-DAI – Apache Lucene

OGSA-DAI, the powerful and flexible grid-middleware for transparent data access on the grid, contains an unsupported information retrieval functionality since its version number 3.0. More concrete, it includes indexing and searching activities built on the wide-known *Lucene* engine of the Apache Project. It incorporates wide-known searching-, indexing- and optimisation features, but is optimised for query- and indexing performance – so it does of course not take use of complex and time-consuming features like LSI.

The indexing and searching engine is accessed by three OGSA-DAI Activities: **addIndexFile** imports a new text file into the file system and generates an index, **searchIndexedFiles** searches an index and **readFile** is used to randomly access a file.

Note that this system does not support global searches on distributed document sets, whereas the system presented in this paper is designed to provide a framework for document sets that are spread over several nodes in the grid. However, one could combine several OGSA-DAI indices by OGSA-DQP services [2].

2.2. The GridIR-Working Group

The GridIR-Working Group is an official working group of the Globus Alliance. It aims at a common standard for IR-engines on the grid. Until now, they have published overviews of the project and the proposed architecture which consists of the following parts:

- The communication layer that can be based either on **grid** technology or simply on common **internet** technologies like web services.
- **Collection managers**: special nodes that track source documents for updates.
- **Indexers** that maintain the index databases and provide methods for searching this databases.
- The **query processor**, the interface to the user. It can e.g. be a page on the internet or a grid service.

This system promises to closely resemble the aims of our approach. It does, however, not seem that there is much activity in this project in the recent years. Until now, the information published by this

working group is far from a complete standard description, so it was not possible to implement this standard.

2.3. Activities for Information Retrieval using LSI

Our approach to implement the usage of LSI and different reordering methods made it necessary to introduce a hierarchical structure. So in our system we have master nodes which propagate queries and add documents to well-known other nodes to build up a tree-like structure of parent and child nodes.

We also have new activities compared to the Apache Lucene approach: **createIndex** creates an empty index. This is necessary because we need a place to define whether we use LSI and term-document matrix distribution. This should happen before documents are added. **commitIndex** commits operations like adding files to the index. This is useful because of the potential time consumed by changing the index. Another necessary activity is **gatherSimilarities**, which compares a document with all reachable documents and returns the searching node where it fits best.

2.3.1. Architecture of Grid-IR using LSI

The architecture of our approach has two kinds of grid nodes.

- Simple node: a computer in the network that is responsible to hold and search parts of indices. It is also the node which optimises the indices by our SVD-based rank reduction. A node may hold several indices but need not hold complete indices. A node is basically only able to locally create and search the part of the index which it holds.
- Master node: a special node that knows the names and addresses of all nodes the hold parts of the index. It may but does not have to hold itself a part of the index. Unless explicitly prohibited by a service option, it sends search queries and any other types of jobs it receives to all nodes it knows before it searches its own indices.

  It is possible to have one single master node which provides a well-known entry point for a client or it would also be possible to have a tree-like structure of parent and children nodes.

The following structuring elements are introduced:

- Index: an index is practically a set of files which hold the important data to make efficient searching on raw and unsorted text documents or document vectors possible. One index is normally distributed over a set of nodes.
- Document vector: a document vector is a preprocessed file in a well-defined form. Every text document will automatically be transformed to a document vector. However, it is also possible for a client to pass a document vector directly. This allows e.g. for multimedia data to be indexed on the basis of extracted features.

If a hierarchical structure is not wanted by the user, e.g. if migrating from the standard OGSA-DAI activities or if the configuration is too static for an application, every node has to be configured as master who has its own index.

2.3.2. Usage scenarios

The need to distribute document sets and indices can result from several situations. First, the retrieval may be part of a bigger application which produces and manipulates documents in a distributed way. The retrieval subsystem then has to accept the location of the documents and cannot move them to another node. Thus, document vectors are always added locally to the index. Only the queries can (but do not have to) be performed globally to locate matching documents across sites.

Second, data may become very large so distributed storage may be necessary. However, moving documents to a certain node once at the time they are included into the index may be acceptable. In both cases, it usually is better to store documents and their index on the same node because in a grid application it is often required to add or remove nodes, which is more easily done if each node brings its own indices.

Of course, another reason to distribute indices is that indices may become too large or the computational demands for creating and manipulating indices are too high for a single node. Given this variety of applications, our system is designed for flexibility to support all of them.

2.3.3. Examples of Grid-IR activities using LSI

The following is an example OGSA-DAI *perform document* that initiates the creation of an index container on the node where it is sent to and on all nodes below in the hierarchy. It defines a name and the level of optimisation which controls the method of index creation such as rank reduction.

```
<createIndex name="testCreateIndex">
     <index>recipes</index>
     <optimizeLevel>8</optimizeLevel>
     <output name="testOutput"/>
</createIndex>
```

The following perform document posts a query to all underlying nodes containing the index “recipes” and expects 20 documents to be returned.

```
<searchIndex name="testSearch">
    <keyWord>apple</keyWord>
    <keyWord>pie</keyWord>
    <index>recipes</index>
    <numberOfAnswers>20</numberOfAnswers>
    <local="false"/>
    <output name="testOutput"/>
</searchIndex>
```

3. Efficiency of LSI on distributed documents

In “grid-enabled” information retrieval systems, we will most certainly never have a single document base. If we want to use the power of the grid methodology, we have a true distributed system of servers without centralised control, so we have no way to run a rank reduction on the global document base.

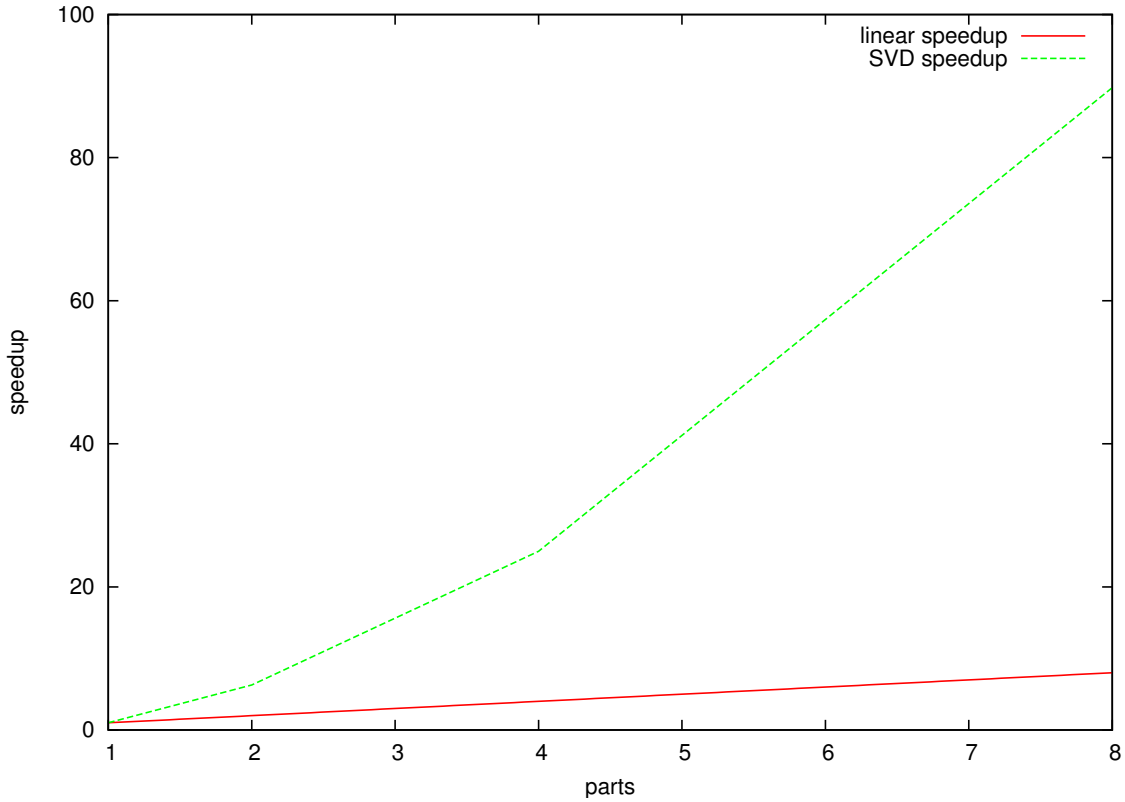

Figure 1. Performance increase achieved by distribution of the document set.

The results presented in this section discuss whether LSI remains effective if we split up the document base before rank reduction is applied and what we can do to improve the resulting quality. For our investigations we have used the *MEDLINE text collection* [1], a text collection which is often used in literature to compare the performance of information retrieval systems. It contains 1033 documents and a set of 30 queries and results; data that is necessary to compute the efficiency measures described in 1.4..

3.1. Indexing Performance

We have seen that LSI is a computationally very expensive process, so our first question we have to ask is: does the indexing process become significantly faster if we can take advantage of parallelisation? According to Figure 1, the performance increase achieved by parallelisation is very high: splitting to two nodes results in a speedup of 8 for the creation of the index. This superlinear speedup results from the fact that the complexity of the SVD is not linear. Therefore, calculating two SVDs on half the data is faster, even if not performed in parallel. Of course, this is not possible without losing information: computing a rank reduction of two halves of a matrix using the truncated SVD leads to different results than a rank reduction on the whole matrix, as we will see in the next section.

3.2. Efficiency of Random Distribution

The first question for our experiments was: what efficiency can we expect if we allow arbitrary distribution? To simulate this, the text collection was just split up randomly and LSI was applied on the resulting parts. The results are disappointing: even a splitting into 2 parts makes a rank reduction

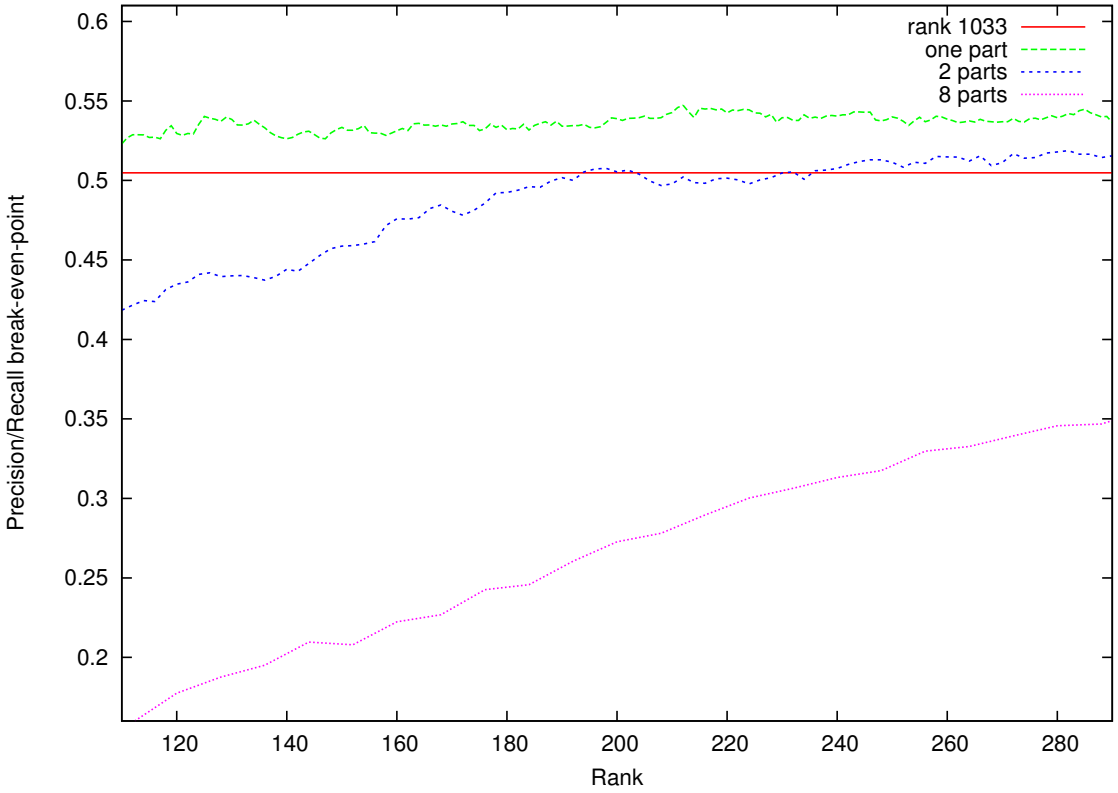

Figure 2. Query performance after splitting up the text collection randomly.

less effective. Starting with 8 parts, every rank reduction will result in a performance even worse than what we will get if we do not apply LSI (see Figure 2). This is not very surprising if we know that the optimal rank for the distribution using our information retrieval engine seems to be approximately $\frac{1}{10}$ of the starting rank: if the documents are really distributed randomly, each part should approximately contain as many latent concepts as documents, so we can not reduce the rank without losing latent concepts.

Therefore, arbitrary distribution is only feasible for very large data sets or data sets with a relatively small amount of concepts.

3.3. Maximum Cosine Reordering Algorithm

Because the performance for randomly distributed documents is bad, we try to improve it by reordering them in order to reduce the number of latent concepts per part, i.e. to distribute documents in a way so that documents containing a certain concept are located in the same part, if possible, so the negative effects of partitioning should be smaller. Thus, the idea behind the reordering algorithm is to group similar documents.

As similarity measure between documents, we use the cosine value of their angle; the same measure we use for the computation of similarities between documents and queries. To compute a partitioning of a large document set into parts of equal sizes in a reasonable amount of time, we use the following heuristic algorithm:

1. Compute similarities between all n documents, store it in an $n \times n$ matrix.
2. Find the smallest value of the similarity matrix: this value tells us which two documents are least similar.
3. Each of these documents is put in one half.
4. For each document that is not yet in one of the halves: add similarities of the document to all documents in the first half. The document with the highest sum of similarity values is added to the first half.
5. Repeat step 4 for the second half.
6. Repeat steps 4 to 5 until each document is in one of the halves.
7. Repeat the algorithm on each of the halves until we have the requested number of parts.

The purpose of this algorithm is, for the time being, only to investigate the impact of the order of documents on the retrieval performance. However, parts of the algorithm can actually be used to find good nodes to move documents to when they are indexed.

3.4. Efficiency of Reordering

Our algorithm tries to split the matrix into parts containing documents as similar as possible, so there should be a substantially reduced amount of latent concepts per part. The results look much better than with random distribution: reordering with our "maximum cosine" algorithm described in 3.3. and splitting into 2 parts leads to an efficiency almost as good as without any splitting, as we can see in Figure 3. Considering the large computational performance benefit (see 3.1.) this seems to be very promising. However, after splitting into 8 parts (see Figure 4) we can not reach the efficiency of the non-distributed approach. The query results are, however, still better than what we achieve without using LSI.

Moreover, there seem to exist special permutations of documents where we even gain performance compared to the original LSI approach if we sort our document base by topic and then split it up. While it was not yet possible to develop a generic algorithm that is able to sort arbitrary document sets this way, this is at least a very interesting and unexpected result that shows the potential of our approach.

3.5. Application of Reordering

We have seen that reordering leads to enhanced query performance. The question is: how can we applicate reordering in a grid-like environment? The following approaches could be convenient for several applications:

- If the location of the documents is dictated by another application or just by the fact that documents are already stored and should not be moved for some reason, we might face a random-distribution situation as in Figure 2. So we have to check if an application of LSI makes sense in the first place. However, the application that generates the documents might as well produce an order by topic and the performance could even increase.

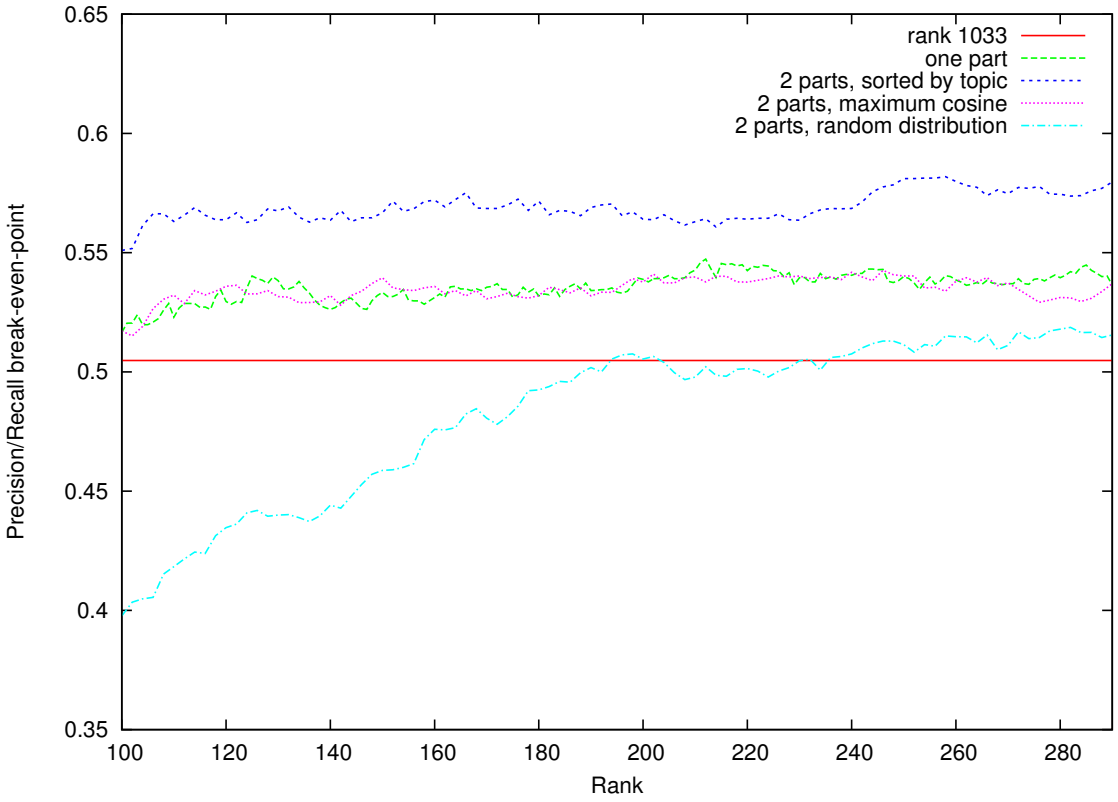

Figure 3. Query performance after several reordering methods and splitting into 2 parts.

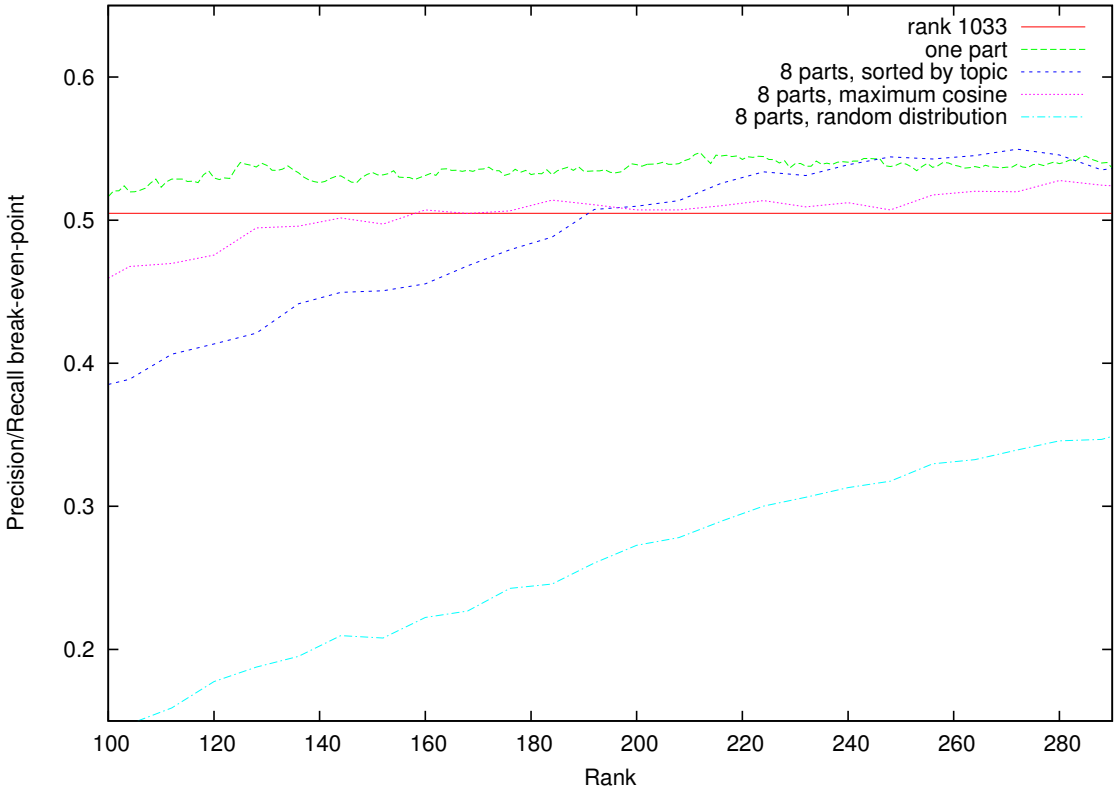

Figure 4. Query performance after several reordering methods and splitting into 8 parts.

- If we want to distribute our document set to several grid nodes or just to "initialise" the grid-based IR engine, we can first reorder the documents with an algorithm similar to the one described in 3.3. such that similar documents reside on each node. Note that this is the reason why the **gatherSimilarities** activity is required.
- If users want to submit documents to a master node, several documents can be cached and reordered such that every known node gets the same amount of documents and they are delivered where they fit best.

4. Conclusion

There is a variety of scenarios to use information retrieval in the grid. The interface structure presented in this work has the flexibility to support most of them. Documents that are statically stored on one or more nodes can be incorporated into a distributed index. Also, documents can be dynamically redistributed to be grouped according to document similarity. The system can easily handle nodes that are added to or removed from the grid, because indices are completely distributed.

However, it has been shown that using LSI in a grid-distributed information retrieval engine is not necessarily as effective as it is with a unified document set. It can even make the query performance worse.

It is, however, effective to use LSI in a distributed environment if the documents on a node share similarities, so the number of latent concepts on one node is much smaller than the number of documents on that node. This can also be the case if there are generally few latent concepts compared to the number of documents. If this is not the case, we might redistribute the documents to ensure this.

Acknowledgements

The work described in this paper is partially supported by the Austrian Grid Project, funded by the Austrian BMBWK (Federal Ministry for Education, Science and Culture) under contract GZ 4003/2-VI/4c/2004.

References

[1] Medline. ftp://ftp.cs.cornell.edu/pub/med/.

[2] M. N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A. A. Fernandes, A. Gounaris, and J. Smith. OGSA-DQP: A service-based distributed query processor for the grid. In S. J. Cox, editor, *Proceedings of UK e-Science All Hands Meeting Nottingham, EPSRC*, volume 24, 2003.

[3] A. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1999.

[4] M. Becka, G. Oksa, and M. Vajtersic. Dynamic ordering for a parallel block-jacobi svd algorithm. *Parallel Computing 28*, 2002.

[5] M. Berry. Large scale singular value computations. In *Internat. J. Supercomputer Application, 6*, 1992.

[6] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. In *J. American Society for Information Science, 41*, 1990.

[7] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. In *Psychometrika*, 1936.

[8] E. R. Jessup, M. W. Berry, and Z. Drmac. Matrices, vector spaces, and information retrieval. *SIAM Review, 41(2):335–362*, 1999.

[9] H. P. Luhn. The automatic creation of literature abstracts. In *IBM Jounal of Research and Development 2 (2)*, 1958.

[10] H. Zha. A subspace-based model for information retrieval with application in latent semantic indexing. In *Proceedings of Irregular '98, Lecture Notes in Computer Science 1457*, New York, 1998. Springer Verlag.