Distributed, Large-Scale Latent Semantic Analysis by Index Interpolation

Sebastiano Vigna
DSI, Università degli Studi di Milano, Italy

Abstract

Latent semantic analysis [12] is a well-known technique to extrapolate concepts from a set of documents; it discards noise by reducing the rank of (a variant of) the term/document matrix of a document collection by singular value decomposition. The latter is performed by solving an equivalent symmetric eigenvector problem on a related matrix. Scaling to large set of documents, however, is problematic because every vector-matrix multiplication required by iterative solvers requires a number of multiplications equal to twice the number of postings of the collection. We show how to combine standard search-engine algorithmic tools in such a way to compute (reasonably) quickly the cooccurrence matrix C of a large document collection, and solve directly the associated symmetric eigenvector problem. Albeit the size of C is quadratic in the number of terms, we can distribute its computation among any number of computational unit without increasing the overall number of multiplications. Moreover, our approach is advantageous when the document collection is large, because the number of terms over which latent semantic analysis has to be performed is inherently limited by the size of a language lexicon. We present experiments over a collection with 3.6 billions of postings—two orders of magnitudes larger than any published experiment in the literature.

Keywords: Latent semantic analysis, search engine

1 Introduction

Latent semantic analysis [12] applies *dimensionality reduction* to the term/document matrix of a document collection. In general, dimensionality reduction is a technique exploiting the existence of *low-rank approximations* of matrices correlating some kind of *entities* and *features* (in the case of document collections, documents and terms). Dimensionality reduction gives a low-rank approximation that has the same salient features of the original matrix, but can be represented very compactly in a low-dimensional space. When considering term/document matrices, for instance, good approximations are typically of rank 200/300.

These reductions are usually computed by *singular value decomposition*, a method originally developed by Eugenio Beltrami and Camille Jordan to characterise linear transformations up to rotations in the source and target spaces; Eckart and Young [14] proved that it provides, in a precise sense, the best possible low-rank approximation. Since then, singular value decomposition has found literally hundreds of applications in several different fields.

The classic application of singular value decomposition to text analysis, called *latent semantic indexing* by its inventors [12], applies dimensionality reduction to the classic vector space representation, in which every document is represented by a vector in the space of features (using within-document frequencies or some weighting scheme) and relevance is measured, for instance, by cosine similarity with a query.

The interesting phenomenon experimentally measured by the inventors of latent semantic indexing is that solving queries in the low-rank approximation of the term/document matrix gives actually *better* results that solving queries in the full matrix. The interpretation given by the authors is that dimensionality reduction squeezes terms into a low-dimensionality *concept space*. At that point, documents are evaluated with respect to their relevance to a concept, rather than to a term, which frees the results of the search from the strict necessity of returning documents containing the query terms.

A particularly striking analysis of this phenomenon has been given recently by Bast and Majumdar [2]: essentially, latent semantic indexing (and more general dimensionality-reduction techniques) owes its success to the ability to identify *cooccurrence* of terms (i.e., terms that appear in the same document). Terms appearing in the same document are obviosuly most related, but also terms coccorring with a third term, and so on. Bast and Majumdar argue theoretically and show experimentally that using the eigenvectors of the cooccurrence matrix (which are part of the singular value decomposition) it is possible to deduce a *relatedness score* between terms that can be used to expand *documents*: the results, in terms of several retrieval-success scores, is even superior to that of a full latent semantic indexing.

This round of results suggests that getting the first few hundreds (or even more) eigenvectors of the cooccurrence matrix is at the core of latent semantic techniques. Actually, from these eigenvectors and the original matrix it is possible to rebuild the full singular value decomposition: many sophisticated methods for compute the singular value decomposition reduce it indeed to a symmetric eigenvector problem [4], considering usually either one of the symmetric matrices $\begin{pmatrix} 0 & M \\ M^T & 0 \end{pmatrix}$, MM^T or M^TM (for a very up-to-date review, see [19]).

This paper started from the obvious observation that inverted indices can be seen as a compact, efficient representation of the matrix M. Less obviously, standard techniques used in the resolution of disjunctive queries can be slightly modified to compute the cooccurrence matrix rather quickly. With some more work, we can also do the same in limited memory, and possibly distributing the computation among several units, without increasing the number of multiplications required. Then, we are faced again with a symmetric eigenvector problem. However, since the term/document matrix of a large collection is *skinny* (the number of rows is significantly smaller than the number of columns, in particular in web applications, were the document collection is very large), we actually have a rather small matrix, and the corresponding eigenvector problem poses easier scale problems.

Essentially, an iteration over the usual symmetric reduction we mentioned requires $2p$ multiplications, where p is the number of postings[1], whereas an iteration over the cooccurrence matrix needs $q \leq m^2$, where q is the number of pairs of cooccurrent terms m is the number of terms: since p grows more or less lineary with the number of documents n, whereas m is inherently limited by the number of terms in a language, at some point the explicit computation of the cooccurrence matrix becomes competitive. (Moreover, if m becomes large most terms end up having a very low frequency, so $q \ll m^2$.)

There is of course a major obstacle—computing and storing the cooccurrence matrix $C = MM^T$ itself. If (for the sake of simplicity) we assume that all documents contain the same number of distinct terms c, it requires $n\binom{c}{2}$ multiplications, and, computing C in the obvious way, $2m$ passes over the whole index, but of course this is not acceptable. The cooccurence matrix can also be computed (with the same number of multiplications) by scanning the document collection, provided that C fits into main memory and can be accessed randomly—again, an unacceptable assumption.[2]

We present a simple technique, *index interpolation*, that leverages standard data structures used in

[1] A *posting* is a pair $\langle d, t\rangle$ such that term t appears in document d. Postings correspond (or at least are a superset) of the nonzero entries of the term/document matrix M.

[2] More sophisticated approached could involve storing partial images of C during the scan of the document collection. The partial images could be saved and merged later. The space occupied by such images would however be enormous, and indeed there are no published reports in the literature using such an approach for large-scale collections.

search engines to split and distribute the computation of the cooccurrence matrix. If a $b \times b$ support matrix can be accessed randomly in main memory, by index interpolation the number of passes over the index is reduced by a factor $\log b/b^2$, and can be performed easily in a distributed fashion.

In Section 2 we define the basic notation. Then, we discuss the standard queue-based techniques for disjunctive query resolution and we introduce the *front*, a new operation on priority queues that we use to interpolate the index. Finally, we present some experiments that show the feasiblity and the efficiency of our approach.

We remark that in this paper we do not discuss the pros and cons of latent semantic analysis, which has its own formations of supporters and detractors, and the numerical problems related to eigenvector computations, which are treated by a vast literature; rather, we concentrate on the algorithmic problem of computing distributedly the cooccurrence matrix of large document collections starting from readily available search-engine tools.

Scalability of latent semantic analysis has always been a core concern [8]. In their upcoming book on information retrieval [18], Manning, Raghavan and Schütze remark that "at the time of this writing, we know of no successful experiment with over one million documents. This has been the biggest obstacle to the widespread adoption to LSI". In a most recent effort based on out-of-core (e.g., disk-based) singular value decomposition [19], experiments are performed on less than one million document, about 300 000 terms and 62 millions of postings.

We present experimental results run on easily scalable commodity hardware for up to 300 000 terms, 25M documents and 3.6 *billions* of postings. They show that using index interpolation latent semantic analysis can be applied to collections significantly larger than it was previously possible.

2 Basic Setup

We consider a collection of n documents D, a set of m terms T, and a term/document matrix M that has m rows and n columns; M_{td} is the value correlating term t with document d: for instance, 0 or 1 if we consider just occurrence, or the *count* (a.k.a. *within-document frequency*), that is, the number of occurrences (possibly zero) of term t in document d, or more sophisticated weighting schemes such as TF/IDF [22] or BM25F [21]. The *singular value decomposition*

$$M = U\Sigma V^T$$

expresses M as the product of two $m \times m$ and $n \times n$ orthogonal matrices (U and V, respectively) and an $m \times n$ matrix that is zero everywhere except on the diagonal, where it has nonnegative entries (Σ). The columns of U and V are referred to as left and right *singular vectors* of M and the values of Σ as the *singular values* of M. Every real matrix has a singular value decomposition.[3]

Dimensionality reduction by singular value decomposition is obtained by sorting in decreasing order the singular values and keeping just the first k (the others are replaced with zeroes). This gives us a new matrix M' of rank at most k that is the best rank-k approximation of M, in the sense that it minimises the difference with M in *Frobenius norm* [14].

A important fact is that U is the eigenvector matrix of MM^T, as

$$MM^T = U\Sigma V^T V \Sigma^T U^T = U\Sigma\Sigma^T U^T$$

and the same holds dually for V. Moreover, any of the two matrices is sufficient to rebuild the singular value decomposition completely (actually, the standard linear-algebra existence proof for the singular value decomposition starts from the eigenvector decomposition of $M^T M$ or MM^T).

We are particularly interested in U because its first k columns provide a measure of correlation of each term with the first k "concepts" (dimensions of the low-dimensional space) defined by dimensionality reduction. This columns can be used, for instance, for query expansion by measuring cosine

[3] Actually, every complex matrix has a singular value decomposition, but in that case U and V are *unitary*.

enqueue(P, x)	insert item with index x in the queue
top(P)	returns the index of the top item
dequeue(P)	returns the index of the top item and deletes it from the queue
change(P)	signals that the top item has changed
size(P)	returns the number of indices currently in P

Table 1: The operations available for an indirect priority queue.

similarity between the k-dimensional vectors associated to each term, or for document reranking after a query has been resolved with standard keyword-based techniques.

Actually, as we already mentioned, starting from the first columns of U it is possible to deduce a *relatedness score* [2] between terms: then, by resolving a query against the original collection, where however documents have been expanded so to contain related terms, one can obtain results comparable or better than those of full latent semantic indexing. The relatedness score is computed on the basis of the eigenvectors of C only, and this was one of the main motivations for the present paper.[4]

3 Disjunctive Queries

We briefly present a quite standard framework for the resolution of disjunctive queries in a search engine. We assume that the document collection has been indexed, so that for each term we obtain an iterator returning the documents[5] in which the term appears (i.e., nonzero columns of M for that term) in increasing order. (For a comprehensive treatment of inverted indices, see [18].)

Given a set of terms $Q \subseteq T$, we denote with $\bigvee Q$ the associated *disjunctive query*. Its semantics is given by the set of documents which contain a nonzero element on at least one column with index in Q (in the simpler case, the set of documents containing the terms in Q). The standard lazy (a.k.a. *document-at-a-time*) way to resolve a disjunctive query is by means of an *indirect priority queue*.

An indirect priority queue P is a data structure based on an *reference array* A, which is managed outside the queue itself, and a priority order that compares items from the reference array. At each time, the queue contains a set of indices into the reference array (initially, a specified set, possibly empty). An array index x can be added to the queue calling the function enqueue(P, x).

The *index* of the least item in the reference array with respect to the priority order can be accessed by invoking the function top(P). The index of the least item with respect to the priority order is also returned by dequeue(P), which removes the index from P.

The data structure assumes that the only item of the reference array that might change its value is A[top(P)]. Such a change must be communicated immediately to the queue by calling the function change(P). Table 1 summarises the operations available on an indirect priority queue.

A standard heap-based implementation [23] uses linear space and logarithmic time for all operations modifying the queue. The resolution procedure for a disjunctive query $\bigvee Q$ involves one iterator per term in Q, and an indirect queue with a reference array containing the last document

[4] We remark that this document-expansion phase (in the form suggested in [2]) poses its own scale problems: it is not feasible to compare a query against each on-the-fly-expanded document of a collection. However, it is an interesting challenge to build, compress and access quickly a *conceptually expanded index* that indexes expanded instances of the original documents. Whereas the low-rank term/document matrix is large, dense, and difficult to access, such an index might benefit from the well-developped techniques used in search engines.

[5] Technically, the iterator returns *document pointers*—integers denoting the documents themselves. To avoid excessive verbosity, we use "document" for "document pointer" whenever no confusion is possible.

Term	Document pointer
a	0, 3, 5, 7, 9
b	3, 7, 10
c	0, 1, 6, 10
d	2, 3, 5, 9
e	1, 4

Document pointer	Front
0	a
2	d
3	a, b, d
5	a, d
7	a, b
9	a, d
10	b

Table 2: On the left, a toy index in tabular form. On the right, the result (document pointers) of the query $a \vee b \vee d$ and the corresponding fronts.

returned for each term by its respective iterator. The queue comparator is the natural document ordering. Initially, the queue is filled with indices $0, 1, \ldots, |Q| - 1$ and the reference array contains the first document returned by each iterator. The first document d satisfying the query is $A[\text{top}(P)]$. Then, we iteratively advance the iterator of index $\text{top}(P)$, update the corresponding entry $A[\text{top}(P)]$ and update the queue by invoking $\text{change}(P)$. When the top element is greater than d, we have the next document satisfying the query. When an iterator is exhausted, its index is dequeued, and when the queue is empty the process terminates.

The front. We now introduce a new basic operation on an indirect queue: the *front*. The front of an indirect priority queue P, denoted by $\text{front}(P)$, is given by the set of indices for which the corresponding value in the reference array is equal to $\text{top}(P)$:

$$\text{front}(P) := \{\, i \mid A[i] = A[\text{top}(P)]\,\}.$$

A simple example of a query, its result and the corresponding fronts is given in Table 2. The front can be trivially computed in linear (in the number of front element) time in a heap-based queue by visiting the heap as a binary tree (other priority-queue implementations might or might not support front construction in linear time; this has to be proved on a case-by-case basis).

In our context, the front provides the terms of Q actually appearing in the last document returned by the iterator. In other words, besides knowing that *some* terms of a disjunctive query appear in a certain document, the front let us know exactly *which* are such terms.

4 Building the Cooccurrence Matrix by Index Interpolation

We assume that the inverted-index we access contains, beside document pointers, the *counts* (i.e., the number of occurrences of a term in a document, a.k.a. *within-document frequency*). After the iterator for term t has returned a document d, it gives access to the corresponding count (i.e., the number of occurrences of t in d). From the count and some global data (e.g., document lengths) we assume to be able to compute the normalised entry M_{td} for each term t appearing in a document d. This arrangement makes it possible to compute all normalisations we are aware of.

We note the following fundamental fact: given two sets of terms $X, Y \subseteq T$, the values of the entries with rows in X and columns in Y of the cooccurrence matrix C are entirely determined by the sequence of fronts appearing in the resolution of $\bigvee X$ and $\bigvee Y$. This happens because the entry C_{tu} is equal to

$$\sum_{d \in D} M_{td} (M^T)_{du} = \sum_{d \in D} M_{td} M_{ud},$$

and this sum can be obviously restricted to documents in which both t and u appear. But such documents are exactly those in the intersection of the document-pointer lists associated to the disjunctive

queries X and Y. On the other hand, for each document d appearing both in the document list associated to $\bigvee X$ and in that of $\bigvee Y$ the two fronts $f_X(d)$ and $f_Y(d)$ fetched just after d has been retrieved contain exactly the terms of X and Y, respectively, that appear in d. We conclude that by searching for all such d's, and enumerating the pairs of terms $\langle t, u\rangle \in f_d(X) \times f_d(Y)$, we can compute the entries of C with rows in X and columns in Y.

We call this technique *index interpolation*. Note that index interpolation can be computed using completely standard inverted-index tools, provided that they give access to the front of the queues solving disjunctions: we just have to scan the documents returned by the query

$$\Bigl(\bigvee X\Bigr) \wedge \Bigl(\bigvee Y\Bigr),$$

because the semantics of the middle $\wedge$ is exactly given by the intersection of the document lists associated to X and Y. Thus, we leverage an existing highly optimised infrastructure.

First approximation: one-pass construction. Suppose we solve a disjunction containing the whole term set T: in this case, the front associated to each document contains exactly the terms occurring in the document. Thus, if the cooccurence matrix would fit our main memory we could easily write directly in memory the cooccurrence matrix C: we just iterate on pairs of elements of the front (which provides all pairs of terms $\langle t, u\rangle$ appearing in the same document), and we update the entry C_{tu} suitably. At the end, we save C in a compressed format for sparse matrices. In practise, we are rebuilding (in a somewhat contrived way) the *direct* information about the collection—the list of terms appearing in each document.

However, assuming that the C can fit main memory is not realistic, in particular if we want to work on, say, more than 100 000 terms. Moreover, in many cases building an iterator requires some resources (e.g., opening files, mapping memory, etc.) and requiring such resources for the whole term set T could be prohibitive.

Second approximation: batched construction. We now refine our strategy: we fix a *batch size* b such that we can comfortably store in main memory a $b \times b$ matrix J (of integer of float data, depending on the kind of normalisation applied to M). We assume a numeration $t_0, t_1, \ldots, t_{m-1}$ of the terms, and we enumerate rows and columns in batches $B_k = \{ t_{kb}, t_{kb+1}, \ldots, t_{(k+1)b-1} \}$. When we process the row batch B_i and the column batch B_j, we solve the query $(\bigvee B_i) \wedge (\bigvee B_j)$, and for each document we look at the fronts of the queues of $\bigvee B_i$ and $\bigvee B_j$: as we remarked, the submatrix of the cooccurrence matrix corresponding to row indices from B_i and column indices from B_j is entirely defined by the sequence of such fronts. It is just a matter, for each document satisfying $(\bigvee B_i) \wedge (\bigvee B_j)$, of enumerating the cartesian product of the two fronts, and update J accordingly so that, at the end of the enumeration of the documents, J contains exactly the submatrix of M we were looking for. We can at this point store J on disk and try the next pair of batches.

A pseudocode implementation of the computation of J is provided in Table 3. We assume that D_i and D_j are iterators over the queries $\bigvee B_i$ and $\bigvee B_j$, respectively; the front of D_i can be obtained with front(D_i) and the count of each term t in the front with count(D_i, t); analogously for B_j. The iterator A is obtained by intersection of D_i and D_j, and returns the documents satisfying $(\bigvee B_i) \wedge (\bigvee B_j)$; finally, we assume, as usual, that after A has returned a document d, both B_i and B_j are positioned over d. All iterators are advanced by a function next() returning the special value $\perp$ when iteration is completed. We note that an obvious optimised version can be used if $B_i \cap B_j \neq \varnothing$, as in that case some entries of J are identical by symmetry.

Batch reuse. If we enumerate row and column batches in natural order, it is obvious that several column batches will be in turn combined with the same row batch. Thus, it is a good idea to store (either in memory or in secondary storage) the result of the query resolution of the row batch B_i, so that it can be retrieved for each column batch more quickly. Unfortunately, the size of the result can be quite large.

```
0   forever begin
1      d ← next(A);
2      if d = ⊥ then return;
3      foreach t ∈ front(D_i) do
4         foreach u ∈ front(D_j) do
5            J_tu = J_tu + count(D_i, t) · count(D_j, u)
6         end;
7      end;
8   end;
```

Table 3: The procedure for computing the submatrix of C corresponding to the terms in row batch B_i and column batch B_j. The iterator D_i enumerates $\bigvee B_i$, D_j enumerates $\bigvee B_j$, and A intersects D_i and D_j to enumerate $(\bigvee B_i) \wedge (\bigvee B_j)$.

We notice that we have to solve a problem similar to that approached by the authors of [3]—storing (in our case, caching) efficiently the results of disjunctive queries. Thus, we use the same approach: we compress and cache the results of solving the row batches queries so that document pointers are gap-encoded, and terms are represented by their frequency ranks. In this way, terms with high frequency are assigned small values, which can be represented using suitable universal instantaneous codes. In any case, the sequence of terms forming the front associated to a certain document is saved in gap encoding, so to reduce further space occupancy. In [3], the authors show that if we assume a Zipf-like distribution for terms in each batch this encoding provides a compression close to the *empirical entropy* of the index, and, in practise, the result is a very low cost (in our experiments, about one byte) per posting. Thus, not only in theory the I/O cost is not larger than scanning the index, but as a matter of fact we half the number of necessary iterators, the use of the associated resources, and we incur into the $\log b$ speed penalty caused by heap operations just once per row.

Distributing the computation. The most interesting feature of the batched approach is that its computations can be trivially distributed (e.g., by assigning a pair of row/column batches per machine). The computation of each batch is completely independent, and it just require access to the inverted lists of the relevant terms; the latter are highly compressed and easily extractible from the whole index, or could be accessed using a high-performance distributed file system. Once again, this kind of access is common in a search engine.

Note that such an assignment corresponds to delegate the computation of a submatrix of C to each computational unit. In a parallel or grid-like computation environment (e.g., Google's MapReduce [11]), say endowed with u computational units, the subdivision will give rise to submatrices of size $m/\sqrt{u} \times m/\sqrt{u}$ which will be manageable even with a large number of terms.

We assume to use an *implicitly restarted* Lanczos-method iterative solver [24], which is the tool of choice for approximated large-scale eigenvector computations. To work on a symmetric eigenvalue problem, the solver just require the ability to compute the product of the involved matrix with a vector.

Such a solver can be easily run in a distributed fashion: to compute $C\boldsymbol{v}$ for a vector $\boldsymbol{v}$, we just split $\boldsymbol{v}$ (which is of size m) into $\sqrt{u}$ pieces $\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_{\sqrt{u}-1}$ of size $m/\sqrt{u}$: each computational unit can then independently and in parallel compute the product of its own $m/\sqrt{u} \times m/\sqrt{u}$ submatrix with a suitable $\boldsymbol{v}_j$. The results can be gathered exchanging overall just $O(m\sqrt{u})$ floating-point numbers and performing just $O(m\sqrt{u})$ sums.

There is also some literature about parallel (e.g., [25]) solution of symmetric eigenvector problems using Lanczos iterative methods that could be applied to further speed up this phase.

5 Comparing Computation Time

In this section we give a rather rough comparison of the effort required to use a centralised, standard approach and ours.

Let us say that we have a set of n documents and a set of m terms giving rise to p postings. We denote with k the number of required eigenvectors, and with s the number of steps required by the solver to compute them. Note that since we are just precomputing C, the number of steps required is the same both in the standard approach and in our case.

It is however important to note that the iterative solver has to multiply by M and M^T; to accomplish this task keeping in memory just a vector of m element, access by row to M^T, rather than to M, is needed (first we multiply M^T and dump the resulting vector $\boldsymbol{v}$ to disk; then, we read $\boldsymbol{v}$ in streaming and build incrementally, rather than coordinatewise, the product $M\boldsymbol{v}$, always accessing M^T by rows). While storing and compressing M is an everyday activity in a search engine, storing and compressing M^T is a different task needing different techniques. (We must also remark, however, that starting from a non-indexed document collection M^T is actually easier to build, as no inversion is needed.)

The number of multiplications required by a Lanczos iterative solver using the standard approach is $2ps$; from an I/O viewpoint, $2s$ passes over some representation of M^T are required (or a pass over M and another pass over M^T, in case both matrices are available). The corresponding time bound is $O(ps)$, assuming the matrices have been compressed so that only to nonzero entries have to be read.

In our case, we have two distinct activities: the construction of C, and the iterative computation of C's eigenvectors. For both cases, we can just try to provide some estimates, as the number of multiplications depends deeply on the distribution of terms within documents.

If the n documents in the collections have each $c_0, c_1, \ldots, c_{n-1}$ distinct terms, to build C we will need

$$\sum_{0 \le d < n} \binom{c_d}{2}$$

multiplications (as we already noted, this is independent on the number of batches and of the degree of distribution of the computation). Note that $\sum_d c_d = p$.

To relate (at least to some degree of approximation) this number of p, we assume a *lognormal* distribution for the c_i; our assumption stems from the lognormal model for the distribution of document lengths [13, 20], but of course this is a very rough approximation, as document lengths count the number of occurrences of terms, not the number of distinct terms. Nontheless, in Figure 1 we show that actually the assumption fits rather well the distribution of the number of distinct terms in the GOV2 collection.

This means that we can try to estimate the number of multiplications required for the computation of C by considering the sum of the squares of a large number of independent, identically distributed lognormal random variables with average p/n:

$$\sum_{0 \le d < n} \binom{C_d}{2} = \frac{1}{2} \sum_{0 \le d < n} C_d^2 - \frac{1}{2} \sum_{0 \le d < n} C_d.$$

Since $C_d = e^{X_d}$ for some normally distributed variable X_d, say with variance σ, $C_d^2 = e^{2X_d}$ has still lognormal distribution. If we denote with μ the mean of X_d (so $p/n = e^{\mu+\sigma^2/2}$), we have that the mean of L_d^2 is

$$e^{2(\mu+\sigma^2)} = (p/n)^2 e^{\sigma^2}$$

We conclude that

$$\sum_{0 \le d < n} \binom{C_d}{2} \approx \frac{1}{2}\left(e^{\sigma^2} p^2/n - p\right).$$

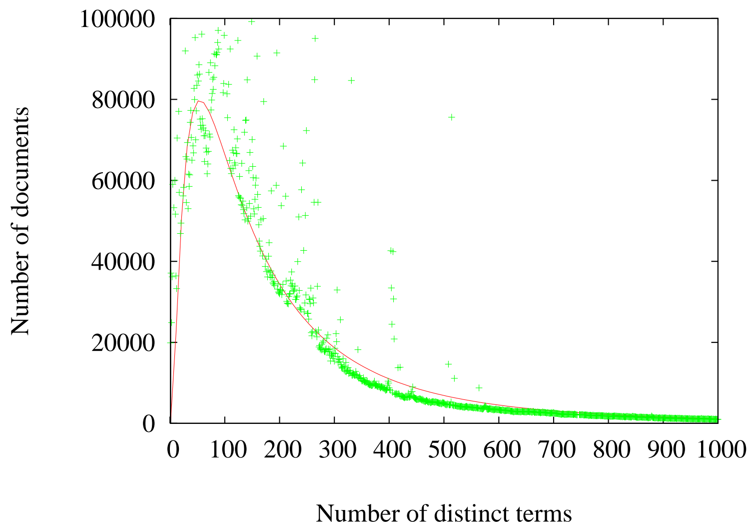

Figure 1: A plot of the number of documents for each value of the number of distinct terms, compared with a suitably scaled lognormal distribution with parameters $\mu = 5$, $\sigma = 1$.

In the end, if we denote with $c = p/n$ the *average* number of distinct terms per document, the experimental fact that for GOV2 $\sigma = 1$ tells us that we need $\approx 1.5\,cp$ multiplications. Essentially, modulo a small multiplicative factor the collection behaves as if all documents had the same number of distinct terms c (in which case we would need *exactly* $nc(c+1)/2 = p(c+1)/2$ multiplications).

Once C is computed, we need to run an iterative solver. Estimating the number of multiplications required is now rather difficult, because the trivial estimate $q \leq m^2$ for the number q of nonzero entries of C does not take into account that, albeit not sparse as M, C is still sparse, in particular if stopwords are eliminated. We have no simple model to propose: experimentally, Table 4 shows that if we eliminate stopwords and proceed including more and more terms of low frequency, the matrix becomes quadratically larger, but it also becomes sparser.

In the range shown, with respect to the number of terms q varies almost linearly: this is due to the fact that when we add low-frequency terms we are mostly increasing q by cooccurrences with *frequent* terms, as terms appearing in less than 0.1% of a collection have a very low probability of cooccurrence. This also implies that by a more careful stopword selection q can be reduced significantly.

All in all, up to 150 000 terms the number of multiplications required by our method is less than half of that required by the standard method. At 300 000, the ratio is reversed and our method requires almost twice the number of multiplications. We claim, however, that the possibility of perfect parallelisation of the matrix/vector products more than compensate for this loss.

In any case, the overall time required by the iterative solver for the s passes over C is $O(qs)$. The time required for the passes over the index during the construction of C requires some more consideration instead, as it depends on the batch size b and on batch reuse.

It is clear that there are two extremes in the choice of batch size: if $B_0 = T$, we make a single pass over the whole index, but we need to access the whole cooccurrence matrix randomly, whereas if $|B_i| = 1$ for all i we essentially compute the product MM^T in the standard way using constant additional memory. Note that the overall number of multiplication is independent of the batch size, as we examine the cooccurrence of two terms in a document exactly once, but the number of passes over the index is 1 in the first case and $2m$ in the second case. On the other hand, in the first case we

Min. Freq	Terms	Postings	Nonzero	Interp.	Iter. (seq)	Total (seq)	Total (16 CPUs)
100	$\approx 300\,000$	3.6 G	6.1 G (13.5%)	23.2 h	1640 s	493 h (0.52 ms/p)	42.5 h (0.043 ms/p)
300	$\approx 150\,000$	3.58 G	3.5 G (31.1%)	18.3 h	800 s	234 h (0.23ms/p)	22.3 h (0.022 ms/p)
600	$\approx 100\,000$	3.56 G	2.2 G (44%)	18.2 h	460 s	134 h (0.14ms/p)	16.2 h (0.016 ms/p)
2100	$\approx 50\,000$	3.5 G	0.8 G (64%)	17.7 h	130 s	42 h (0.04ms/p)	7.3 h (0.0075 ms/p)

Table 4: Postings, nonzero entries in the upper triangular part of the cooccurrence matrix, interpolation time, sequential time for a single iteration of the eigensolver and total sequential and parallel time for different subsets of terms. The notation ms/p denotes milliseconds per posting. The terms were selected by discarding stopwords, terms containing digits, terms longer than 30 characters and then thresholding the minimum frequency. The overall number of iterations required to compute the 300 required eigenvectors is 1051 in all cases.

have to manage a large heap of size m, imposing a $\log m$ slowdown on query resolution, whereas in the second case there is no cost associated to heaps.

By batching, we reduce the number of passes to $(m/b)^2$, but each pass has a $\log b$ slowdown due to heap management. Actually, since most of the cost of index scanning is iterator access and query resolution, and these costs are to be counted just once per row by batch reuse, an accurate estimate the time required by index scanning is $O(p(m/b + 1)\log b)$, which we confirm experimentally in Section 6.

Nonetheless, there are several additional factors to take into account: a batch of large size means that several posting lists have to be scanned in parallel, which, depending on the size of the index, might give rise to a significant disk-seek activity. Finally, if the batch is too large the cooccurrence matrix might be significantly larger than the processor second-level cache, resulting in a high cost in term of cache misses. All these problems, however, are easily solvable once the computation is distributed among a large number of units.

6 Experiments

We report some experiments witnessing the feasibility of our approach using commodity PCs with a Xeon 2.80 GHz processor and 4 GiB of RAM (but we actually use a fourth of that). We report also timings for the same experiments running in parallel on 16 CPUSs of the same kind.

We start from an MG4J[6] index of the TREC GOV2 collection (about 25M documents). MG4J has a number of facilities to manipulate and transform inverted indices that are particularly suited to the computation we described. First of all, we fix a stopword list of 541 terms (obtained by the Cross Language Evaluation Forum), we eliminate terms containing digits and terms longer than 30 characters. Then, we use MG4J's built-in facilities to eliminate unwanted terms from the index and generate indices with an increasing number of terms by thresholding the minimum frequency (see Table 4).

These limitations are dictated by the need of including all "meaningful" terms: by thresholding the frequency we avoid the enourmous amount of garbage of a web collection.

This point deserves a full discussion, as, of course, the eigenvectors of C do change if terms are removed from the collection. In the literature attacking large-scale latent semantic analysis, different approaches have been used in the past: for instance, Martin *et al.* [19] uses a preprocessed collection (where preprocessing is not completely defined, and includes at list the removal of a "large stop list") but then do not remove any term from the collection, whereas Bast and Majumdar [2] performs

[6]MG4J (Managing Gigabytes for Java) is a Java search engine [7] developed by the author in collaboration with Paolo Boldi.

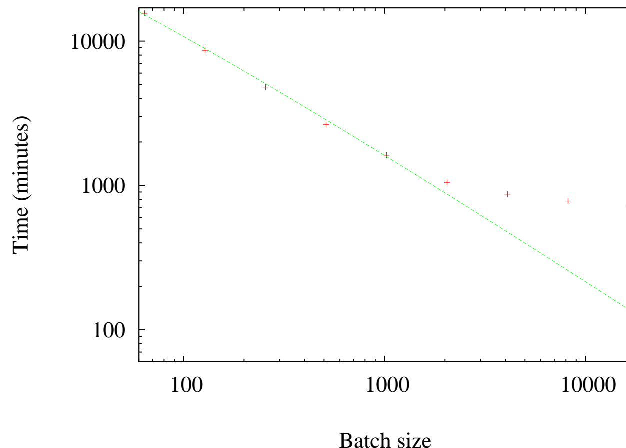

Figure 2: Computation time (single machine) plotted with respect to batch size. The line shows the predicted $\log b/b$ scaling. When the batch size becomes too large, disk-seek activity and cache misses become prominent.

several kinds of term removal that seem to be similar to ours (we cannot be more precise because the authors do not describe fully the criteria for inclusion/exclusion); since the second paper contains a full and very successful result evaluation, we assume that stopwords and low-frequency terms can be removed harmlessly.

It should also be noted that *web collections make term removal dramatically necessary*: our index of GOV2, for instance, contains 34M terms, most of which are of course garbage (e.g., strings appearing in base64 encodings of mailing-list attachments). All literature we are aware of applies latent semantic analysis to very clean collections (e.g., newspaper articles) so they can afford to keep around all terms[7]. But such collections are akin to a well-kept garden, whereas the web is a jungle: typos, distorted words, and even ASCII art, make the number of terms of an index unbelievably large.

In a real application, the selection would be probably be less arbitrary that the one we used, and would involve a dictionary-based or an n-gram based selection (to get out-of-dictionary terms with a plausible linguistic structure). Since the main goal of this paper is improving the scalability of latent semantic analysis, the actual terms involved do not actually matter: our choice, moreover, creates a set of rather dense term lists that constitute the worst case for the techniques we describe.

We then proceed to compute the cooccurence matrix C—actually, since it is symmetric, we just compute the upper right part. We store the matrix using a simple compression scheme encoding runs of zeros by their run lengths, as the matrix is rather sparse. This trivial encoding speeds up significantly the iterations of the eigensolver.

In Figure 2 we show in logarithmic scale how computation time decreases when the batch size is increased (we tried batch sizes 2^i, $6 \leq i \leq 14$). For very small batch sizes, the computation time is mostly dependent on index access, and then our model correctly predicts a $\log b/b$ scaling. When batch size goes beyond 1024, the computation time becomes dominated by multiplications and memory accesses. When the batch size exceeds 8192 the time becomes essentially stable, as we start paying for the non-locality of index access and cache misses.

Once the matrix is computed, we use ARPACK [17] to obtain its first 300 eigenvectors; sequential and parallel computation times for four different subsets of terms are given in Table 4.

[7] More precisely: all literature for which complete and detailed information about the construction of the term/document matrix is available.

We remark that these experiments can be easily replicated using the GOV2 collection, which is (more or less) publicly available, MG4J and ARPACK, which are free software, and a few hundreds lines of code (available upon request from the author).

7 Prior Work

First of all, let us notice that a line of research on large-scale focuses on a reduction of the original matrix: for instance, [19] propose to compute clusters of documents and substitute clusters with their centroids, whereas [1, 15] suggest to partition the document collection into semantically unrelated areas. In both cases the result is a much smaller matrix, which can be handled using standard techniques. This approach is clearly orthogonal to the kind of problems we want to solve.

There is of course a vast literature about parallel and distributed SVD computation (see, e.g., the comprehensive survey [6] or the reference handbook [16]) that can be immediately applied to LSA. Actually, we used for our sequential tests ARPACK's implementation of the Lanczos method, which has a parallel counterpart (PARPACK). Nonetheless, the technique we describe makes it possible to apply immediately the simplest parallelisation strategy—having distinct, nonoverlapping submatrices at each processor. Most of the effort in parallelising an iterative method is concentrated in spreading intelligently information among the processors (e.g., depending on the sparseness of the matrix), but in our case we will build information exactly where it is needed: all multiplications happen with perfect parallelism, as all computational units will compute independently exactly the part of the cooccurence matrix they need, without increase of total work.

A parallel implementation of a variant of the Lanczos method using multiplication by M and M^T has been presented by Berry and Martin in [5]. As an example, they report, on a network of 16 dual (450 MHz) UltraSPARC II 64-bit processors with 512 Mbytes SDRAM, 10 minutes (0.025 ms/posting) for computing the first 100 eigenvector of the TREC LATIMES collection (156 413 terms, 131 896 documents, $\approx$ 23.5M postings) using less than 300 iterations. A direct comparison of the final timing per posting, rescaled for the different number of iterations (0.08 ms/posting against our 0.022) is very favourable, but also not very instructive, because our method is geared to skinny term/document matrices, so it would probably not work so well on such a small collection. Moreover, the CPU performances are difficult to compare. It would rather be interesting to evaluate Berry and Martin's parallel Lanczos on a reasonably large collection such as GOV2.

Finally, there is a vast literature about matrix multiplication. Most of the literature, however, is concerned with the product of *square* matrices, and all results we are aware of for rectangular matrices (starting with the classic paper by Coppersmith and Winograd [10], later refined by Coppersmith [9]) require random access to the matrices involved, which is not feasible in our setting.

To the best of our knowledge, the published experiments with largest corpora are those performed in [19]: using an AMD Opteron 240 running at 1.4 GHz, about 20 hours were necessary to compute 300 eigenvectors of a collection of 62 millions of postings ($\approx$ 800 000 documents, $\approx$ 300 000 terms), resulting in a cost of $\approx$ 1.16 ms/posting. We are able to compute 300 eigenvectors over 3.6 *billions* of postings ($\approx$ 25 000 000 documents, $\approx$ 300 000 terms) in 516 hours, resulting in 0.52 ms/posting, with the benefit of leveraging an existing infrastructure and an easy parallelisation procedure leading us to 42.5 hours and 0.043 ms/posting.

It must be noted again, however, that a direct confrontation with figures reported in the literature (even scaled by the number of postings) is not so instructive, as we use a collection whose number of postings is at least two orders of magnitude larger, and with such a significant change of scale many nonlinear effects come into play.

8 Conclusions and Future Work

We have described a simple system to compute the cooccurrence matrix of a document collection using minor modifications to standard tools used in search engines. In a sense, we are using the machinery of a search engine as an efficient, distributed matrix multiplier. Thus, we can take advantage of the large body of knowledge accumulated, of optimised implementations, and of the compressed matrix representation implicitly provided by an inverted index. At the same time, our approach is very scalable and lends itself naturally to distributed, parallel or grid-like computation.

The inherent limit of our approach is the size of the cooccurrence matrix, which led us to (somewhat artificially) limit the number of terms to about 300 000 in our experiments: on the other hand, we do not know of published experiment using a larger number of terms. It should be noted that very frequent terms are usually stopwords, which just add noise to the term/document matrix; moreover, including low-frequency terms has little impact on computation time, and generates very sparse cooccurrence matrices, which can be compressed.

Depending on the density and on the size of the cooccurrence matrix, which in turn depend on the distribution of terms and document sizes in the document collection, the number of multiplications required by one pass of an iterative solver using the cooccurrence matrix can be larger or smaller than that required by a standard approach (i.e., multiplying by M and M^T). It is an interesting open problem to develop a probabilistic analysis of the density of the cooccurrence matrix that could lead to a choice of one technique or the other, depending on the structure of the document collection.

As we have discussed, when applicable our techniques are significantly faster than previous proposals, and the collections we are able to manage easily are (in postings) at least two orders of magnitude larger than those reported in the literature. This opens a number of interesting possibilities in the application of latent semantic analysis to collections with dozens of millions of document.

9 Acknowledgments

The author would like to thank Oerd Cukalla and Alessandro Rinaldi for implementing the first version of the index interpolator.

References

[1] Devasis Bassu and Clifford Behrens. Distributed LSI: Scalable concept-based information retrieval with high semantic resolution. In M. W. Berry and W. M. Pottenger, editors, *Proc. of the 2003 Text Mining Workshop, San Francisco, CA*, pages 72–82, 2003.

[2] Holger Bast and Debapriyo Majumdar. Why spectral retrieval works. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 11–18, New York, NY, USA, 2005. ACM Press.

[3] Holger Bast and Ingmar Weber. Type less, find more: Fast autocompletion search with a succinct index. In Efthimis N. Efthimiadis, Susan T. Dumais, David Hawking, and Kalervo Järvelin, editors, *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 364–371. ACM, 2006.

[4] Michael W. Berry. Large-scale sparse singular value computations. *International Journal of Supercomputer Applications*, 6(1):13–49, Spring 1992.

[5] Michael W. Berry and Dian I. Martin. Parallel SVD for scalable information retrieval. In *Proceedings of the International Workshop on Parallel Matrix Algorithms and Applications, Neuchatel, Switzerland*, 2000.

[6] Michael W. Berry, Dani Mezher, Bernard Philippe, and Ahmed Sameh. Parallel computation of the singular value decomposition. Rapport 00071892, Inria, 2003.

[7] Paolo Boldi and Sebastiano Vigna. MG4J at TREC 2005. In *The Fourteenth Text REtrieval Conference (TREC 2005) Proceedings*, 2005.

[8] Chung-Min Chen, Ned Stofel, Mike Post, Chumki Basu, Devasis Bassu, and Clifford Behrens. Telcordia LSI engine: Implementation scalability and issues. In Karl Aberer and Ling Liu, editors, *Eleventh International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications, Heidelberg, Germany, 1-2 April 2001*, pages 51–58. IEEE Computer Society, 2001.

[9] Don Coppersmith. Rectangular matrix multiplication revisited. *J. Complex.*, 13:42–49, 1997.

[10] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9(3):251–280, 1990.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[12] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[13] Allen B. Downey. The structural cause of file size distributions. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 328–329, New York, NY, USA, 2001. ACM Press.

[14] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[15] Jing Gao and Jun Zhang. Clustered SVD strategies in latent semantic indexing. *Inf. Process. Manag.*, 41(5):1051–1063, 2005.

[16] Erricos J. Kontoghiorghes. *Handbook of Parallel Computing and Statistics*. CRC Press, 2006.

[17] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[18] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. Preprint available online.

[19] Dian I. Martin, John C. Martin, Michael W. Berry, and Murray Browne. Out-of-core SVD performance for document indexing. *Applied Numerical Mathematics*, 2007. In press.

[20] Michael Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Math.*, 1(3):226–251, 2004.

[21] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. Simple BM25 extension to multiple weighted fields. In *CIKM '04: Proceedings of the thirteenth ACM international Conference on Information and Knowledge Management*, pages 42–49, New York, NY, USA, 2004. ACM Press.

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

[23] Robert Sedgewick. *Algorithms in C: Parts 1–4: Fundamentals, data structures, sorting, searching*. Addison–Wesley, Reading, MA, USA, 1998.

[24] Danny C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Mat. Anal. Appl.*, 13(1):357–385, 1992.

[25] Kesheng Wu and Horst Simon. A parallel Lanczos method for symmetric eigenvalue problems. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, 1997. ACM SIGARCH and IEEE.