Index Structures for Information Filtering Under the Vector Space Model *

Tak W. Yan and Hector Garcia-Molina

Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

With the ever increasing volumes of electronic information generation, users of information systems are facing an information overload. It is desirable to support information filtering as a complement to traditional retrieval mechanism. The number of users, and thus profiles (representing users' long-term interests), handled by an information filtering system is potentially huge, and the system has to process a constant stream of incoming information in a timely fashion. The efficiency of the filtering process is thus an important issue.

In this paper, we study what data structures and algorithms can be used to efficiently perform large-scale information filtering under the vector space model, a retrieval model established as being effective. We apply the idea of the standard inverted index to index user profiles. We devise an alternative to the standard inverted index, in which we, instead of indexing every term in a profile, select only the significant ones to index. We evaluate their performance and show that the indexing methods require orders of magnitude fewer I/Os to process a document than when no index is used. We also show that the proposed alternative performs better in terms of I/O and CPU processing time in many cases.

1 Introduction

Information is increasingly available in electronic form. The number and size of full text document databases are rapidly increasing. Users of such database systems are facing an information overload; it is becoming difficult for users to rely solely on traditional retrospective search and retrieval mechanisms to keep themselves apprised of new documents that are relevant to their interest. As a complement to conventional search mechanism, information

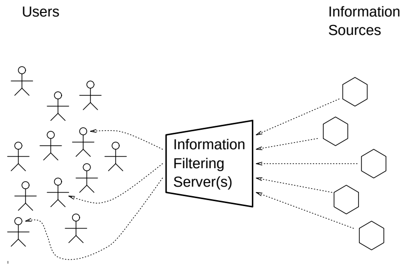

Figure 1: Information Filtering Server(s)

systems can provide an *information filtering* mechanism, through which a user subscribes *profiles*, or queries that are continuously evaluated, to represent his long-term interests, and then passively receives information filtered by the system according to the profiles.

Research in information filtering has received a lot of attention lately. However, previous work has focused on the effectiveness (precision and recall) of the filtering, and little has been done to address the efficiency (performance) aspect of the problem. We believe that information filtering is going to be used on a large scale and hence the efficiency issue must be addressed. In this paper, we present data structure and algorithms to support information filtering.

Wide area information retrieval is now a reality; large-scale world-wide information filtering is also foreseeable. Consider a population of users and a number of information sources in a networked information filtering environment. The filtering can be done either at the information sources, at the user sites, or at an intermediate *information filtering server* (Figure 1). Relying solely on user filtering is expensive since network bandwidth is wasted to transmit irrelevant information and a lot of wasteful local processing is done. Relying on filtering at the sources themselves is also expensive since users need to replicate their profiles at *all* possible sources. The information filtering server is a good compromise. It collects information from a set of sources and routes it to interested users. Of course, there can be multiple information filtering servers on the network, each servicing a different set (maybe overlapping) of users and information sources.

In this paper, we focus on one information filtering server and consider what data structure and algorithms

*This research was sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Grant No.MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of ARPA, the U.S. Government, or CNRI.

it can employ to speed up the filtering process. This is important because, firstly, the number of users and profiles a server has to handle is potentially huge. Secondly, as the rate of information generation is high, a filtering server will have to process a large number of new documents everyday, especially if the server collects information from a number of sources. Thirdly, it is important to deliver relevant information to users in a timely fashion for such a service to be useful. In summary, information filtering servers will have to handle huge number of profiles and process a constant stream of incoming documents in a timely fashion. Thus, to develop efficient processing methods for a single filtering server can be seen as the first but important step in achieving efficient filtering on a global scale.

To further motivate the need for efficient information filtering methods, let us look at a popular information source today – Netnews. The study [11] reports that, as of January 1993, the total Netnews readership world-wide is estimated to be 1.9 million. The estimates for the average traffic are 49.5 MB and 19,210 messages per day (counting cross-posted messages only once). If we consider a Netnews filtering server that serves a small fraction (say 5%) of this user population, and each user has say five profiles, the server will have to handle hundreds of thousands of profiles. To match this large number of profiles against a daily influx of tens of thousands of documents in a timely fashion, it is apparent that efficient data structures and algorithms are needed. Furthermore, keep in mind that these Netnews numbers are for a *single* information source *today*. In the future, one would expect many more sources with even higher volumes.

Netnews does support a rudimentary filtering mechanism by categorizing articles into newsgroups and allowing users to subscribe to newsgroups of interest. However, a finer granularity of information need matching, by means of information retrieval techniques, will cater much better to individual interests. Research in information retrieval has given rise to many retrieval models, notably the boolean model, the vector space model, and the probabilistic model, that are applicable to information filtering [1]. Reference [17] presents data structures and algorithms for information filtering under the boolean model. In this paper, we consider the vector space model (VSM), which is widely recognized as an effective retrieval model. It uses a natural language interface, which makes it easy to use. A well-known technique, called relevance feedback, provides an easy way to improve retrieval effectiveness. Some of the ideas in the VSM have been implemented in the WAIS system [8]. The popularity of WAIS demonstrates the appeal of the VSM. Our methods are thus for documents and profiles represented in the VSM.

Our algorithms make use of an *inverted index* to speed up the filtering process. Inverted indexes have been used by information retrieval systems to facilitate traditional retrospective search, namely by building an index of documents. In this paper, we investigate how the idea of an inverted index can be used to speed up profile processing. Specifically, we propose to use an inverted index of profiles. [1] In the information retrieval scenario, a user query is matched against a document index. Here, an incoming document is matched against a profile index. We investigate what modifications need to be made, and what alternatives are feasible.

Incidentally, we have implemented two experimental filtering servers at Stanford to disseminate Netnews articles and computer science technical reports. The reader is encouraged to try out these services. For instructions on how to use these services, send an electronic mail message to either `elib@db.stanford.edu` (for technical reports) or `netnews@db.stanford.edu` (for Netnews) with the word "help" in the message body. Instructions will be returned automatically. The current versions of these servers are not efficient (they use the Brute Force method described later on). However, as more users subscribe to our servers, there is an obvious need for an efficient implementation, and this motivated the work reported in this paper.

The rest of the paper is organized as follows. In Section 2, we give a brief summary of the VSM, as applied to information filtering. In Section 3, we present three methods to process profiles. Details of the analysis and simulations used to evaluate the performance of the methods are described in Section 4. The results of the evaluation are presented in Section 5. Section 6 is a survey of related work and Section 7 is for conclusion.

2 Information Filtering Under VSM

In this section, we give a brief summary of the VSM as used in information filtering. The purpose of this is to explain some terminology and assumptions necessary for the exposition of our algorithms in Section 3. For an in-depth introduction to the VSM and information filtering the reader is referred to [12] and [1] respectively.

2.1 Document and Profile Vector

In the VSM, we identify documents by terms. A document D is represented as a vector of dimension m, where m is the total number of terms used to identify content. Each term is given a weight that signifies its statistical importance. We write $D = (w_1, ..., w_m)$, where w_i is the weight assigned to the i-th term (and is 0 for terms not present in D).

To compute the vector representation of a document, we first collect the individual words occurring in the document. Words that belong to the *stop list*, which is a list of high-frequency words with low content discriminating power, are deleted. Then a stemming routine is used to reduce each remaining word to word-stem form. For each remaining word stem (a term), a weight is assigned in an attempt to represent how "important" that term is. One common way to compute the weight of a term is to multiply the term

[1] Other retrieval methods (e.g., signature files [4]) can also be used to speed up filtering (e.g., building a signature file of profiles). In this paper, we focus on inversion-based methods. Further work would need to be done to compare the performance of signature-based and inversion-based methods for information filtering.

frequency (tf) factor with the inverse document frequency (idf) factor. The tf factor is proportional to the frequency of the term within the document. The *idf* factor corresponds to the content discriminating power of the term: a term that appears rarely in documents (e.g., "queue") has a high idf, while a term that occurs in a large number of documents (e.g., "system") has a low idf. [2] (See Section 4.1.1 for examples of formulas used to calculate these factors.)

As profiles in the VSM are expressed in natural language, we can represent profiles just like documents. A profile P appears as $P = (u_1, ..., u_m)$. Sometimes we follow the convention of writing a document or profile vector as a vector of (term, weight) pairs; those terms not listed have weights equal to 0. Thus, a profile P with p non-zero weighted terms can be written as $P = ((y_1, u_1), ..., (y_p, u_p))$. For instance, in the profile $P = ((\text{"queue"}, 0.93), (\text{"system"}, 0.37))$, term "queue" has a weight 0.93, "system" has 0.37, and all other terms have a zero weight. The weights again describe the "importance" of each term.

2.2 Similarity Measure

We can measure the degree of similarity between a document-profile pair based on the weights of the corresponding matching terms. The cosine measure has been used for this purpose; given a document $D = (w_1, ..., w_m)$ and a profile $P = (u_1, ..., u_m)$, the cosine similarity measure is:

$$sim(D, P) = \frac{D \cdot P}{\|D\| \|P\|} = \frac{\sum_{i=1}^{m} w_i u_i}{\sqrt{\sum_{i=1}^{m} w_i^2 \sum_{i=1}^{m} u_i^2}}.$$

In this paper we assume that the document and profile vectors are normalized by their lengths; thus the above simplifies to:

$$sim(D, P) = D \cdot P = \sum_{i=1}^{m} w_i u_i.$$

2.3 Relevance Threshold

In an information retrieval setting, a query is run against a database of documents, and the relevant documents are returned to the user, ranked by their scores, i.e., the similarities between the query and the documents. In an information filtering setting, a profile is compared with a single document or a small number of documents. It is undesirable to filter documents based on the ranks among a small batch of documents. In [5], a fixed number of top ranked documents is returned over a certain period of time. This is only possible if the period is long enough to allow a significant number of documents to be collected to make the ranking meaningful; and in doing so, the timeliness of the documents is sacrificed. Also, the filtering effectiveness (precision and recall) depends on the particular set of documents received during a period. If all documents are relevant, then some will be missed (low recall). If few documents are relevant, then some documents delivered will be irrelevant (low precision). Reference [5] indeed reports such drawbacks.

An alternative, as suggested in [5], is to allow the user to specify some kind of absolute relevance threshold – documents above the threshold are considered relevant, and those below are not. With this strategy, instantaneous processing of documents is possible (i.e., a document can be processed one at a time, as soon as it is received). Also, the precision and recall of the filtering are independent of when it is performed. Such relevance threshold can also be used in conventional information retrieval; [13] describes such an experiment. We sum up this discussion with the following definition.

Definition 1: Given a profile P and a relevance threshold θ, a document D is *relevant* to P if $sim(D, P) > \theta$. □

2.4 Relevance Feedback

Relevance feedback is a technique used to improve the effectiveness of retrieval. This technique can be applied to information filtering as well. Our methods will work regardless of whether relevance feedback is used. We did perform a set of experiments to evaluate the methods under relevance feedback. Due to space limitations, we present the results in [18].

3 Data Structures and Algorithms

In this section we describe three methods that match a document against a number of profiles and determine the profiles to which the document is relevant. We assume that a document is processed one at time, as soon as it arrives. Our methods can easily be extended to handle the case when a number of documents is batched together for processing, but we do not address this here.

In two of the methods, we make use of an inverted index. In an index, for each term x, we collect profiles that contain it to form an inverted list. [3] The mapping from terms to the location of their inverted lists on disk is implemented as a hash table, called the *directory*. We assume that the inverted lists are stored on disk while the directory fits in main memory.

Our focus in this paper is on efficient VSM filtering algorithms. The issue of how to efficiently update profiles in the data structures is not addressed. We assume that such updates are batched and are periodically installed. However, in the evaluation of our indexing methods, we do consider two options of storing inverted lists on disk. One option is to pack all the lists into contiguous blocks, and the other

[2] In an information filtering setting, the number of new incoming documents processed at one time is small, so the inverse document frequencies within the batch may not be the most reliable. Instead, we may extract the *idf*s from a pre-existing reference corpus of text, as is done in [5].

[3] As detailed later, we may collect all or some of the profiles that contain a term to form its inverted list.

is to store each list individually in an integral number of blocks. While handling updates in the first option requires reading and writing all the lists, it is much easier in the second option. On the other hand, the storage space requirement for the first option is higher. In our evaluation we examine the trade-off involved.

3.1 Brute Force (BF) Method

If we store profiles sequentially on disk without any index structures, then all profiles must be evaluated when a new document is received. This is the *Brute Force (BF)* method.

When a document arrives, we first compute its vector representation as described in Section 2. Then we examine each profile in turn. For each (term, weight) pair (x, u) in a profile, we find x's weight w in the document vector, and calculate the product $w \times u$. The sum of such products is the cosine similarity measure. The document is relevant to a profile if the cosine measure is greater than the relevance threshold associated with the profile.

We store a profile on disk as a variable-length *record* with these fields: the profile identifier, the length – i.e., the number of terms in the profile, the (term, weight) pairs, and finally the relevance threshold.

3.2 Profile Indexing (PI) Method

To reduce the number of profiles that must be examined, we build an inverted index of profiles. We call this the *Profile Indexing (PI)* method. For each term x, we collect all the profiles that contain it to form its inverted list. The list is made up of *postings*; each contains the identifier of a profile involving x and the weight of x in it. Thus, a profile with p terms will be found in p postings; each in a different list. When processing a document D, we only need to examine those profiles in the inverted lists of the terms that are in D.

To match a document against these profiles, we need two (main memory) arrays, THRESHOLD and SCORE. (This method and the next use more main memory than the BF method.) The number of entries in each array is equal to the number of profiles the system handles. Each profile has an entry in each array: the THRESHOLD entry stores the relevance threshold, and the SCORE entry is used to keep the score of the profile.

When a document D arrives, we initialize the SCORE array to all 0's. For each term x with weight w in the document, we use the directory to retrieve x's inverted list. Then we process each profile P in the list. That is, if the weight of x in P is u, we increment SCORE$[P]$ by the product of $w \times u$. After all document terms are processed, a profile whose SCORE entry is greater than the THRESHOLD entry matches the document.

To illustrate, consider three profiles: $P_1 = ((a, 0.46), (b, 0.14), (c, 0.17), (d, 0.62), (e, 0.59))$, $P_2 = ((a, 0.95), (b, 0.30))$, and $P_3 = ((c, 0.14), (e, 0.49), (f, 0.17), (g, 0.42), (h, 0.11), (i, 0.10), (j, 0.72))$, with relevance thresholds of $\theta_1 = 0.25$, $\theta_2 = 0.20$, and $\theta_3 = 0.25$ respectively.

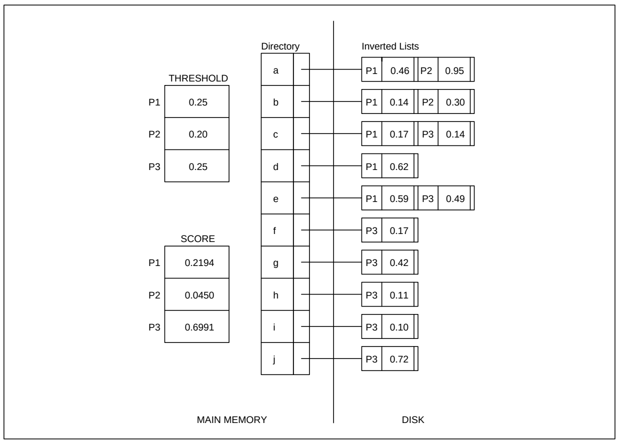

Figure 2: Data Structures for Profile Indexing

The inverted index for these profiles is shown in the right-hand side of Figure 2. For example, the a list contains the postings for P_1 and P_2. The 0.46 value in the first entry in this list is the weight of a in P_1. Now suppose this document arrives:

$$D = ((b, 0.15), (d, 0.32), (f, 0.21), (h, 0.14), (j, 0.90)).$$

To process this document, first we read the b list, and increment the SCORE entries of P_1 and P_2 by $0.15 \times 0.14 = 0.021$ and $0.15 \times 0.30 = 0.045$ respectively. The lists of d, f, h, and j are processed similarly. The final values of the SCORE array are as shown in the figure. This document is relevant to P_3.

Notice the PI method is almost symmetrical to the method used in information retrieval to match a query against a database of documents with an index of documents, with the roles of documents and queries (profiles) reversed. The difference is that the THRESHOLD array is not used; instead, after the computation of similarities, the SCORE array is sorted to find the rank of the documents.

3.3 Selective Profile Indexing (SPI) Method

In the PI method, we index a profile by all its terms. In this subsection we investigate an alternative in which we only select a number of terms for indexing.

Consider the term b in P_1 in our running example. Suppose a document arrives and it does not contain the terms a, c, d, or e. The maximum score P_1 could have against this document is 0.14 (if b's weight in the document is the highest possible, 1.0), which is less than the threshold specified. At a threshold of 0.25, the term b is insignificant in that it alone cannot produce enough score for a document to be relevant. Thus, we may choose not to index the profile with the term b – a document that contains only b and no other terms in the profile will not be relevant anyway. However, a document that contains b and another term in the profile may be relevant; so we need to duplicate $(b, 0.14)$ in the postings of the other terms in their respective lists. (Since we assume that the inverted lists are stored on disk, it is better to duplicate the pair than to store it elsewhere and keep a pointer in the postings to reference it (extra I/Os will be needed to look it up). If the entire index fits in main memory, it is better to use the pointer option. See comments in Section 7.)

Similarly, consider the subvector $((h, 0.11), (i, 0.10))$ in P_3. Suppose a document arrives that does not have the other terms in P_3. Then an upper bound to the similarity between P_3 and this document is $0.11 + 0.10 = 0.21$ (we can actually find a tighter upper bound, by a theorem proved below). Again, with a threshold of 0.25, the subvector is insignificant. In this case, we may choose not to post the profile in the inverted lists of h and i and duplicate the pairs in the postings of the other terms in the profile. These observations lead us to this definition.

Definition 2: Given a profile vector $P = ((y_1, u_1), \ldots, (y_p, u_p))$, a subvector $P_s = ((y_{i_1}, u_{i_1}), \ldots, (y_{i_s}, u_{i_s}))$, $1 \leq i_1 < \ldots < i_s \leq p$, is *insignificant* at a threshold of θ if for any document D, $sim(D, P_s) \leq \theta$. □

Given a profile like P_3, there may be several insignificant subvectors (e.g., $((h, 0.11), (i, 0.10))$ is one, $((c, 0.14), (i, 0.10))$ is another). Which subvector should we use to reduce the number of index postings? One idea is to use the subvector that contains the most low-*idf* terms. Low-*idf* terms occur more frequently in documents; thus, by not posting these terms we expect to save the most lookup work.

Definition 3: Given a profile vector $P = ((y_1, u_1), \ldots, (y_p, u_p))$, a subvector $P_s = ((y_{i_1}, u_{i_1}), \ldots, (y_{i_s}, u_{i_s}))$, $1 \leq i_1 < \ldots < i_s \leq p$, is *most insignificant* at a threshold of θ if it has the largest number of lowest *idf* terms among the insignificant subvectors at a threshold of θ. □

Assuming *idf*s are distinct, a profile vector has a unique most insignificant subvector at a given threshold. We need a way of checking whether a subvector is the most insignificant subvector and this requires the ability to compute the maximum possible similarity between a profile subvector and any document vector. Intuitively, we can see that the similarity between a profile subvector and any unit document vector is highest when the document vector is "in the same direction" as the profile subvector. And if that happens, the similarity is given by the magnitude of the profile subvector. This is formally stated and proved as follows.

Theorem 1: For any P and any D, $\|D\| = 1$, we have $sim(D, P) \leq \|P\|$.

Proof: This follows easily from the Cauchy-Schwarz Inequality [6]:

$$sim(D, P) = D \cdot P \leq |D \cdot P| \leq \|D\| \|P\| = \|P\|. \quad \blacksquare$$

To find the most insignificant subvector of a profile vector, we can sort the terms by *idf* and include as many terms as possible. For example, consider P_3 again. We assume that the term weights are directly proportional to the *idf*s

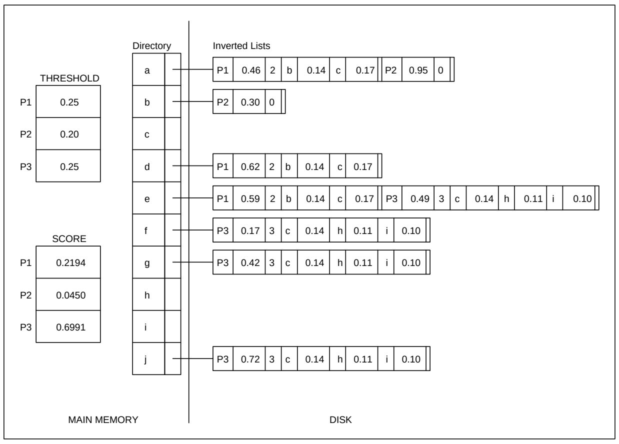

Figure 3: Data Structures for the SPI Method

(which is true if the tf components are the same). As

$$\|((c, 0.14), (h, 0.11), (i, 0.10))\| = 0.2042 \leq 0.25, \text{ and}$$

$$\|((f, 0.17), (c, 0.14), (h, 0.11), (i, 0.10))\| = 0.2657 > 0.25,$$

$((c, 0.14), (h, 0.11), (i, 0.10))$ is the most insignificant subvector of P_3 at a threshold of 0.25. This also shows that Theorem 1 is stronger than the naive way of finding an upper bound by simply adding the weights, as we have done earlier.

With this knowledge, we can indeed index the profiles selectively. For each profile, we find the most insignificant subvector at the threshold specified. The profile is then posted in the inverted lists of the significant (relative to the most insignificant subvector) terms. In each posting, we include the insignificant terms and their weights; i.e., they are replicated in the lists of all the significant terms. This is called the *Selective Profile Indexing (SPI)* method.

Each posting contains the profile identifier, the weight of the term indexed, the number of insignificant pairs, and the pairs of insignificant terms and weights. Postings in the same list are stored sequentially in blocks.

We also require the THRESHOLD and SCORE arrays as in the PI method. When a document comes along, we construct its vector representation. Next we initialize the SCORE array to all 0's. Then we index the directory to retrieve the inverted lists of each term. Suppose we are processing the term x with weight w in the document. For each profile P in the x list, suppose the weight of x in P is u, and the insignificant pairs are $(y_{i_1}, u_{i_1}), \ldots, (y_{i_s}, u_{i_s})$. We examine P's SCORE entry. There are two cases: if the SCORE entry is zero, we first add the product $w \times u$. Then we look up each term y_{i_j} in the document vector. Suppose its weight in the document is w_{i_j}. We add the product $w_{i_j} \times u_{i_j}$ to the SCORE entry. In the second case, the SCORE entry is not zero, meaning that we have already added the contribution of the insignificant terms in some earlier computation. Thus we only add the product $w \times u$. After all document terms have been processed, a profile matches the document if its SCORE entry is greater than the THRESHOLD entry.

Figure 3 shows the index for our running example. For instance, suppose we are processing the first pair (b, 0.15) from the document vector. The list of b has only one posting, that of P_2. We add the product $0.15 \times 0.30 = 0.045$ to P_2's SCORE entry. As there is no insignificant subvector, we are done with this posting and also with the b list. Next we process the pair (d, 0.32). Only P_1's posting is in the d list. First we add the product $0.32 \times 0.62 = 0.1984$ to SCORE[P_1]. Then we process the insignificant subvector ((b, 0.14), (c, 0.17)). To do this, we look up the term b in the document vector, getting a weight of 0.15. Thus we increment SCORE[P_1] by the product $0.15 \times 0.14 = 0.021$. Next, we look up c, which is not in the document vector. We are now done with this list. The other pairs are processed similarly. The final values for SCORE are as shown in the figure.

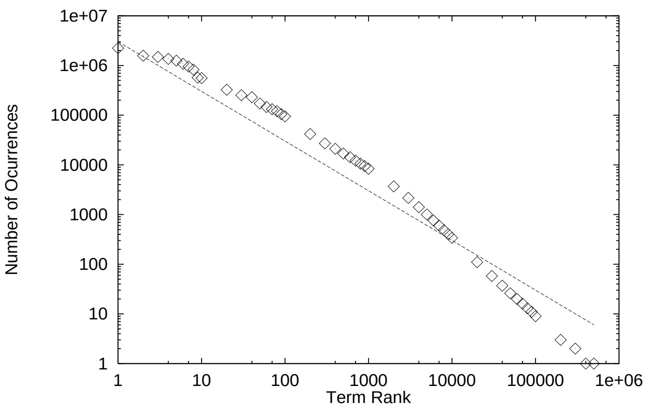

Figure 4: Term Rank vs Frequency Graph for Netnews Database

4 Performance Evaluation

4.1 Models

We use analysis and simulations to evaluate the performance of the methods. To allow flexibility in our performance evaluation, we use synthetic document and profile models. To make them realistic, we base our models on properties of a database of Netnews (text) articles received by our Department's Netnews host during the period of April 22 to April 29, 1993. A total of 212,972 articles were collected, making up a 550MB database. Below we describe our models.

4.1.1 Document Model

The following steps were carried out to study the occurrence frequency of terms in the database. First, a lexical analysis screened out all non-alphabetical characters from the documents (i.e., articles). Then a stemming routine (Porter's algorithm [10]) was run to reduce the remaining words to word-stem form. Each stem thus obtained is a term. Next we measured the occurrency frequency of each term in the database, obtaining the plot shown in Figure 4 (note the log/log scale). The straight line in the graph was derived by curve fitting using [16]. We can see the database does demonstrate Zipfian characteristics [19]. The x-intercept (i.e. size of the term vocabulary, which we denote by v) is found to be 521,915. Also, the average number of words per document (denoted by d) is found to be 323.

Hence, we adopt the following probabilistic document model, which is similar to the one in [15]. The terms in a document come from a vocabulary V of size v. Each term is uniquely represented by an integer x, $1 \leq x \leq v$. The probability that any term appears is described by the probability distribution Z. We rank the terms in non-increasing order of frequencies, i.e., $\forall x, y, 1 \leq x < y \leq v$, we have $Z(x) \geq Z(y)$; for convenience, we use the rank to identify the terms. We assume the frequency distribution follows Zipf's Law; i.e.,

$$Z(x) = \frac{1}{x \sum_{y=1}^{v} 1/y}.$$

A document has d term occurrences and is generated by a sequence of d independent and identically distributed trials; each trial produces one term from V according to the distribution Z. The most frequent s terms form the stop list; stop-listed terms are deleted from a document before its vector representation is computed. We choose s to be 100 in the evaluation.

Finally, the vector representations of the documents are computed as described in Section 2. The exact formulas used to compute the weight of a term x_i are from [13], which have been empirically found to be effective:

$$\begin{aligned} tf_i &= 0.5 + 0.5 \times \frac{f_i}{\max_j f_j}, \text{ and} \\ idf_i &= \log(1/\text{fraction of documents with } x_i), \end{aligned}$$

where f_i is the frequency of the term x_i in the document. We analytically compute the fraction in idf as the probability that x_i appears in a document.

4.1.2 Profile Model

Looking at our database, we find that a large fraction of the terms in the vocabulary occur very infrequently. Those terms are mostly from misspellings, typos, or self-invented words. We do not expect these terms to appear in profiles, which represent long term interests. We model this by assuming that profile terms are chosen from the set $Q = \{s + 1, ..., q\}$, called the queried vocabulary, out of the vocabulary $V = \{1, ..., v\}, q < v$. (Recall that we are identifying terms by their ranks.) A base value of 50,000 is chosen for q, covering more than 97% of the total occurrences of terms in the Netnews database.

We assume that each term in Q is equally likely to be chosen for a profile. This uniform distribution is justified as queries tend to use a mix of frequent and relatively infrequent words [15]. Also, terms rarely occur more than once in a profile [12]; thus we assume that a profile is a set of p terms chosen randomly without replacement from the queried vocabulary Q.

The number of profiles in the system is n. To simplify the study of the effect of profile size (p) on performance, we assume all profiles have the same length, i.e., p is fixed for all profiles.

Some of these assumptions may not be valid when relevance feedback is used. In [18], we modify our profile model in the evaluation of the methods under relevance feedback.

4.1.3 Choice of Relevance Threshold

It is hard to model the relevance threshold distribution. For a user, a suitable relevance threshold for his profile depends on the individual profile terms (their *idf*s), the degree of correlation among the terms, the amount of relevant, as well as irrelevant, information in the incoming stream, and his desired level of precision and recall (is it crucial to receive all possibly relevant documents, or is it more desirable to receive those that are likely to be relevant?)

Instead of deriving a complicated model of relevance threshold, we assume the relevance threshold is fixed for all profiles. This allows us to study clearly its impact on the methods. A reasonable base case value was found by the following procedure. First a random document was generated. Then a profile was created to contain a number of overlapping terms, randomly selected from the document. The similarity between the document and the profile was computed. The procedure was repeated a large number of times. For a base case profile length of 5, we found that a profile with 4 or more matching terms has an average similarity of about 0.2. Thus we use this as the base value of the relevance threshold for our evaluation. Of course, this is not saying that the relevance threshold simply translates to the number of matching terms. We are merely settling with a reasonable starting point in our evaluation. In Section 5.5, we vary the threshold over the entire range of possible values from 0 to 1 and examine its effect on the performance.

Parameter	Base Value	Description
v	521,915	size of vocabulary
d	323	# term occurrences per document
s	100	end of stop list
q	50,000	end of queried vocabulary
n	300,000	# profiles
p	5	# terms per profile
θ	0.2	relevance threshold
i	4	# bytes for profile identifier
l	2	# bytes for profile length
t	4	# bytes to represent a term
f	4	# bytes for a floating point number
b	512	# bytes in a disk block

Table 1: Summary of Parameters Used in Performance Evaluation

Table 1 summarizes the parameters used in the models, together with some parameters that specify the sizes of various fields in the data structures, and the disk block size. Keep in mind that the base values shown are simply starting points for our evaluation. We explore different sets of values in our experiments – Section 5 shows some of the results.

4.2 Metrics

We compare the methods with respect to their space and time requirements. For space requirement, we look at how much disk space each structure takes. (Although main memory space requirements of the methods differ, we assume they fit in main memory.) We study two ways of storing the inverted lists in the indexing methods: the first is to pack all lists contiguously into sequentially blocks, leaving no disk space in between lists; the second way is to store each list in an integral number of blocks, allowing easy list expansions. By comparing the space requirement for these two options, we can see the amount of internal fragmentation the second option produces.

For time requirement, in an I/O bound system, the critical measure is the number of I/O's to process a document; in a CPU bound system (including the case when a large portion of the data structures can be cached in main memory), the amount of computation is the critical component. Hence, we look at both aspects in our comparison. For the CPU computation, we count the number of floating-point multiplications each method requires to process a document. The number of multiplications is one of the major computation costs in processing a document, so we believe it is a good measure of CPU cost.

In summary, we look at these metrics:

- the expected total disk space required in number of blocks (with contiguous allocation and fragmented allocation for indexing methods),
- the expected number of disk reads needed to match a document, and
- the expected number of floating point multiplications performed to process a document.

4.3 Analysis and Simulations

Except those for the SPI method, the results in the Section 5 were obtained by deriving analytical solutions and then numerically evaluating the expressions. Due to space limitations, we present the analysis in [18].

Simulations were conducted to obtain the results for the SPI method. We also constructed simulations to validate the analysis. The simulation results did match the analytical ones. We wrote our simulation program in C. The program first generates n profiles according to the profile model, and then computes the size of the index structures needed to store the profiles. Next the simulation program generates a document according to the document model and counts the number of disk reads and multiplications needed to match it against the n profiles. For each scenario we have tested, the program is run enough times (with different random number generator seeds) to make sure that the results are within $\pm 5\%$ of the true values, with a 90% level of confidence.

5 Results

5.1 Base Case Results

The results for the base case are given in Table 2. In the case when the inverted lists of the indexing methods are packed contiguously, the total space requirement for the three methods are roughly comparable. PI is better than the BF method, since the threshold values are stored in main memory. The SPI method requires more space than PI, because some (term, weight) pairs are duplicated in a number of lists in the index.

When the inverted lists are not packed, but are stored individually in an integral number of blocks, internal fragmentation leads to an increase in total space requirement of about 68% for SPI to 113% for PI. The split-list strategy allows for easier updates, but we have to pay the price of higher storage costs.

For the number of disk reads performed per document, we see orders of magnitude improvement of the indexing methods over the BF method. The SPI method is best, due to the fact that certain frequent terms in a profile are not indexed. For this same reason, the number of multiplications for SPI is lower than that for BF and PI (the latter two perform the same number of multiplications; see the analysis in [18]).

Method	Contiguous Size (Blocks)	Fragmented Size (Blocks)	Disk Reads	Multi-plications
BF	29,297	–	29,297	4,314
PI	23,438	49,900	144	4,314
SPI	29,630	49,804	127	3,434

Table 2: Results for the Base Case

In what follows, we describe several sensitivity studies in which we vary the parameter values.

5.2 Size of Queried Vocabulary

The first parameter that we exercise is q, which controls the size of the queried vocabulary. Figures 5 to 7 show the results.

In Figure 5, when the contiguous-list strategy is used, the total space requirement for each method is insensitive to q. However, when the split-list strategy is used for the indexing methods, their space requirement does vary with q. The fluctuations in the graph for SPI can be explained as follows. When q is 20,000, each inverted list occupies 2 blocks. As q increases, the number of lists increases, and so the total size increases. At the same time, the number of postings in a list decreases, since they are distributed over a larger number of lists. At some point (around $q =$ 30,000), the lists begin to shrink in size to 1 block, and this explains the drop in total size. Thereafter, the total space requirement increases linearly with q, as each list fits in 1 block. The same reasoning can be applied to the fluctuations in the graph of PI.

Figure 6 shows the results for the number of blocks read per document. The number of blocks read for the BF method is constantly equal to its total space requirement, and thus the graph is omitted to show the variations

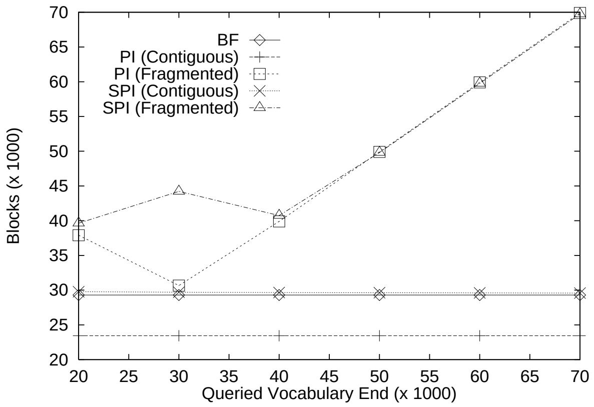

Figure 5: Total Size vs. Queried Vocabulary End q

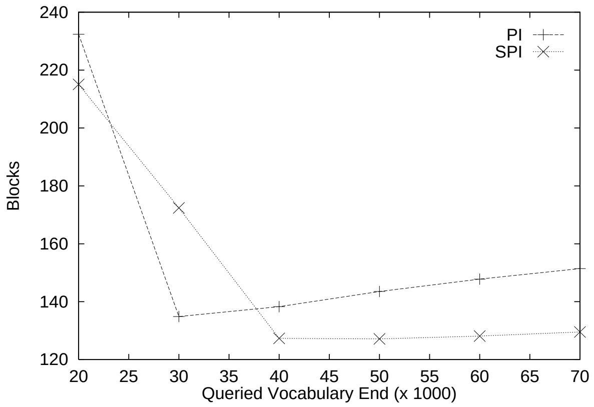

Figure 6: Disk I/Os Per Document vs. Queried Vocabulary End q

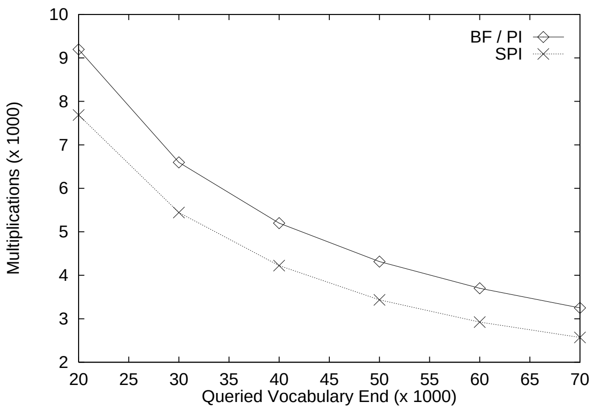

Figure 7: Multiplications Per Document vs. Queried Vocabulary End q

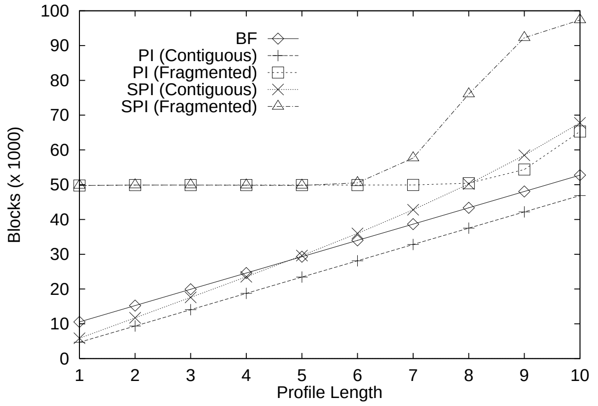

Figure 8: Total Size vs. Profile Length p

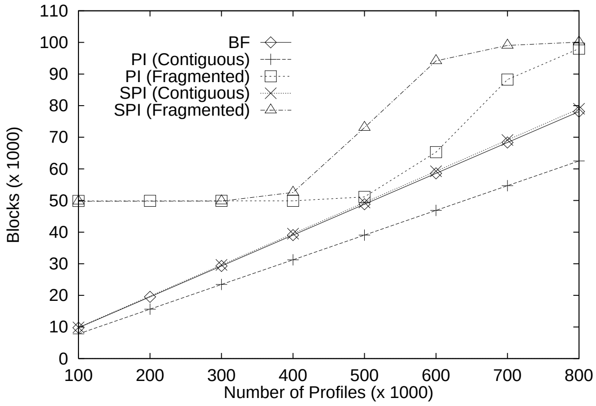

Figure 10: Total Size vs. Number of Profiles n

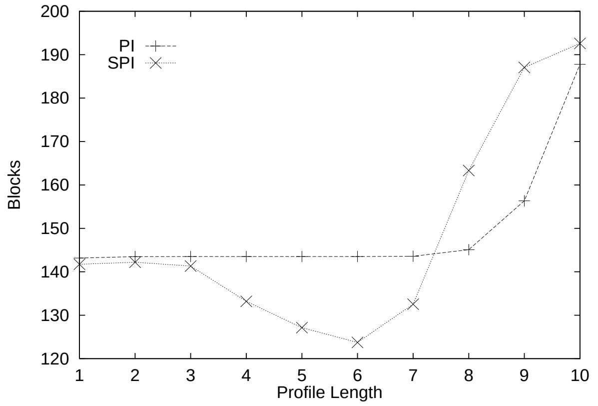

Figure 9: Disk I/Os Per Document vs. Profile Length p

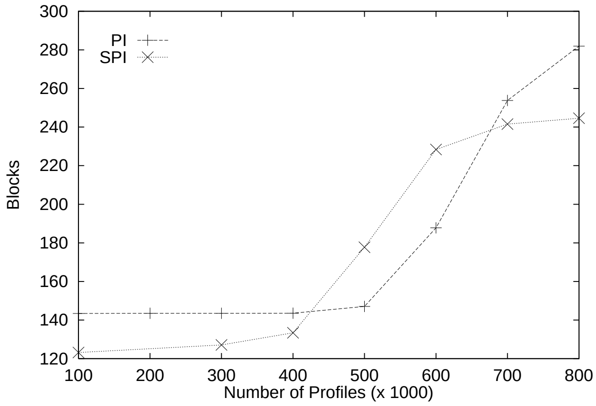

Figure 11: Disk I/Os Per Document vs. Number of Profiles n

in the other methods better. The sharp drop in the number of I/Os required corresponds to the shrinking of the list length (from 2 blocks to 1 block). Thereafter, the number of I/Os increases, as the number of lists read per document increases (due to the increase in the queried vocabulary size). The rise is more prominent in PI than in SPI.

For the number of multiplications per document (Figure 7), SPI is better throughout than the other methods. The trend is downward for all methods, as more infrequent terms appear in profiles.

5.3 Profile Length

The next parameter that we vary is the profile length. Figures 8 and 9 show the results.

For contiguous allocation, we see the total space requirement grows with p for all methods (Figure 8). For fragmented allocation, with a small p, the inverted lists each fit in one block, so the size remains constant at the queried vocabulary size. With larger p, the lists grow in length, so the total space requirement grows also. The SPI method grows at a faster rate than the PI method.

The number of disk I/Os required by the SPI method initially *decreases* as p is increased from 1 (Figure 9). This is because it becomes more likely that a profile includes infrequent terms and is thus indexed by those terms. With the longer lists at larger p (greater than 7), its performance deteriorates and then stabilizes. On the other hand, for the number of multiplications, SPI is always better than the two other methods (graphs omitted).

5.4 Number of Profiles

We vary the number of profiles from 100,000 to 800,000. For the total space requirement (results shown in Figure 10), we have a similar graph as that for p. For contiguous allocation, the space requirement grows linearly with n. For fragmented allocation, the space required is at first constant and then increases. Each inverted list fits in 1 block at the beginning, but as n increases, 2 blocks are needed to hold a list. The lists grow at a faster rate in the SPI method initially, but PI soon catches up with it.

Figure 11 shows the results for the number of disk I/Os required per document. Those for the BF method are omitted. We see there is a range of n values where SPI requires more I/Os per document; this happens when an SPI inverted list grows faster than a PI list. When the list length becomes the same in both methods, SPI again becomes better PI.

In terms of number of multiplications per document, all methods scale proportionally to the number of profiles, with the SPI method always better than the other two methods. Due to space considerations, we omit the graphs here.

5.5 Relevance Threshold

The next parameter that we vary is the relevance threshold. Although it may not make sense to have threshold value of 0 or 1, we study the entire range of possible values to confirm our intuition about the SPI method. The other methods are insensitive to the relevance threshold.

With θ increasing, we expect a more substantial portion of a profile to be insignificant and be duplicated in the lists of significant terms in SPI. Thus the total index size increases, but as θ increases further, the insignificant portion is posted in fewer lists (the number of significant terms decreases). Thus, a certain maximum would be reached somewhere in the range. This is indeed the case for our results shown in Figure 12.

Although the total size increases and then decreases with increasing θ, the number of I/Os is always decreasing (Figure 13), because profiles are indexed in fewer lists of lower frequency terms. Similarly, the number of multiplications decreases also (Figure 14).

The *relative* performance of SPI against the other two does not vary much with different values of θ. For the space requirement, it almost always requires more space that the other two, except when θ is close to 1. For the time requirement, it is always no worse than the other methods.

5.6 Document Size

The size of documents only affects the two time requirement metrics. The performance of the methods with respect to both metrics scales proportionally to the document size, with no change in relative performance. Due to space limitations, we omit the results.

6 Related Work

References [2, 5, 9] investigate the effectiveness of different retrieval models applied to information filtering.

In [17], we study what index structures can be used to speed up information filtering under the boolean model. The PI and SPI methods presented in this paper can be seen as generalizations of the Counting and Key methods in [17].

Terry et al. [14] propose the notion of continuous queries in relational databases. Users issue continuous queries, which are rewritten into incremental queries and run periodically. Their work concentrates on relational databases, while ours is concerned with the dissemination of unstructured data (documents) using information retrieval techniques.

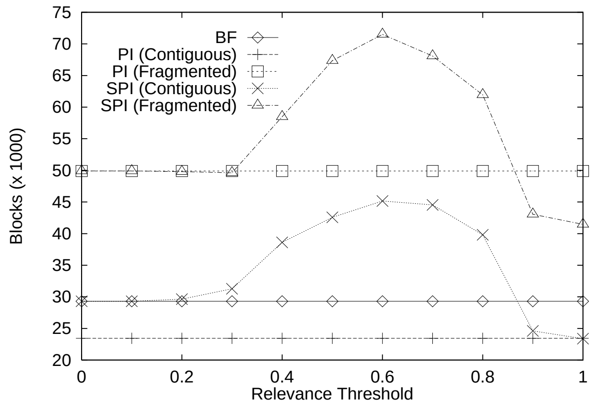

Figure 12: Total Size vs. Relevance Threshold θ

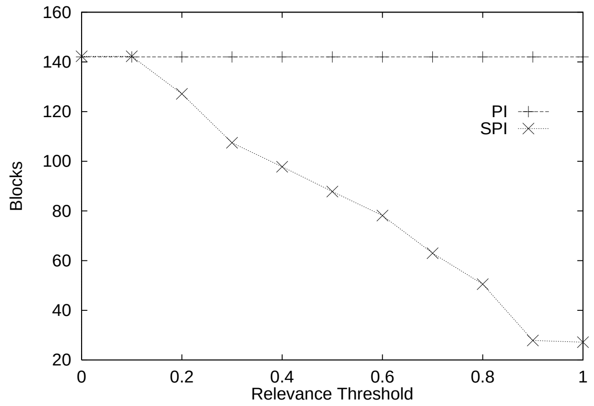

Figure 13: Disk I/Os Per Document vs. Relevance Threshold θ

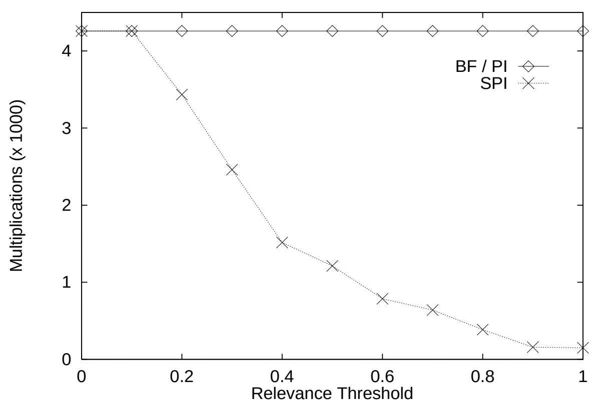

Figure 14: Multiplications Per Document vs. Relevance Threshold θ

Related to the idea of a profile index is that of the "segment tree" presented in [3]. There, Danzig et al. present a distributed indexing scheme as a way to provide efficient retrospective search of a large number of retrieval systems. Special sites, called index brokers, maintain indexes of remote retrieval systems. They subscribe "generator queries" that keep them informed of changes in these systems. The segment tree is proposed to index numerical generator queries over Library of Congress numbers (e.g., all new items in the range QA76 to QA77). Index structures for general profiles are not addressed.

7 Conclusion

In this paper, we study what data structures and algorithms can be used to facilitate large-scale information filtering under the VSM. We apply the idea of the standard inverted index to index user profiles (we call this the PI method) and show that only slight modifications are needed to use the index to speed up filtering. We devise an alternative, called the SPI method, to the standard inverted index – instead of indexing every term in a profile, we select only the significant ones to index. We evaluate their performance, together with the BF method which uses no profile index.

In summary, we see that the three methods require approximately the same disk space when inverted lists are packed into contiguous blocks. When lists are stored individually in an integral number of blocks, the indexing methods require more disk space than the BF method. On the other hand, when we compare the time requirement, the BF method is the clear loser. The indexing methods require fewer number of I/Os to match a document by orders of magnitude. Among the PI and SPI methods, SPI is always better in terms of CPU processing. It can also improve the number of I/Os required in many cases, depending mainly on the profile length and the number of profiles.

Although in those cases where SPI wins, the difference may appear small, we should remember that the results shown are for processing a single document. An information server will be doing this matching day in and day out, and the difference will be magnified. Another observation is that as SPI is always the best in CPU processing, when main memory is large enough to hold the entire index, SPI is the clear choice. In that case, instead of duplicating insignificant terms in lists of indexed terms, we can just use a pointer to reference the insignificant terms, stored separately.

Acknowledgements

Thanks to Ben Kao and Anthony Tomasic for helpful discussions on this paper, and to the anonymous referees for their comments.

References

[1] BELKIN, N.J., and CROFT, W.B. Information filtering and information retrieval: two sides of the same coin? *Communications of the ACM* 35, 12 (Dec. 1992), 29-38.

[2] CROFT, W.B. The University of Massachusetts TIPSTER project. *SIGIR Forum* 26, 2 (Fall 1992), 29-33.

[3] DANZIG, P., AHN, J., NOLL, J., and OBRACZKA, K. Distributed indexing: a scalable mechanism for distributed information retrieval. In *Proc. ACM SIGIR Conference* (Chicago, Oct. 1991), pp. 220-229.

[4] FALOUTSOS, C. Access methods for text. *ACM Computing Surveys* 17, 1 (Mar. 1985), 49-74.

[5] FOLTZ, P.W., and DUMAIS, S.T. Personalized information delivery: an analysis of information filtering methods. *Communications of the ACM* 35, 12 (Dec. 1992), 29-38.

[6] FRIEDBERG, S.H., INSEL, A.J., and SPENCE, L.E. *Linear Algebra*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[7] HORTON, M. How to read the network news. *UNIX Documentation*, AT&T Bell Laboratories.

[8] KAHLE, B., and MEDLAR, A. An information system for corporate users: wide area information servers. *Connexions – The Interoperability Report* 5, 11 (Nov. 1991), 2-9.

[9] POLLOCK, S. A rule-based message filtering system. *ACM Transactions on Office Information Systems* 6, 3 (July 1988), 232-54.

[10] PORTER, M.F. An algorithm for suffix stripping. *Program* 14, 3 (1980), 130-7.

[11] REID, B. USENET Readership Summary Report for January 1993. USENET Newsgroup `news.lists` (February 8, 1993).

[12] SALTON, G. *Automatic Text Processing*, Addison Wesley, Reading, Massachusetts, 1989.

[13] SALTON, G. Global text matching for information retrieval. *Science* 253 (Aug. 1991), 1012-5.

[14] TERRY, D., GOLDBERG, D., NICHOLS, D., and OKI, B. Continuous queries over append-only databases. In *Proc. ACM SIGMOD Conference* (San Diego, May 1992), pp. 321-30.

[15] TOMASIC, A., and GARCIA-MOLINA, H. Performance of inverted indices in distributed text document retrieval systems. In *Proc. Parallel and Distributed Information Systems Conference* (San Diego, Jan. 1993), pp. 8-17.

[16] WOLFRAM, S., *Mathematica*, Addison Wesley, Redwood City, California, 1991.

[17] YAN, T.W., and GARCIA-MOLINA, H. Index structures for selective dissemination of information. Technical Report STAN-CS-92-1454, Stanford University, 1992.

[18] YAN, T.W., and GARCIA-MOLINA, H. Index structures for information filtering under the vector space model. Technical Report STAN-CS-93-1494, Stanford University, 1993.

[19] ZIPF, G.K. *Human Behavior and the Principle of Least Effort*, Addison-Wesley Press, Cambridge, Massachusetts, 1949.