The SIFT Information Dissemination System

Tak W. Yan

Healtheon Corporation
87 Encina Avenue
Palo Alto, CA 94301

Hector Garcia-Molina

Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

Information dissemination is a powerful mechanism for finding information in wide-area environments. An information dissemination server accepts long-term user queries, collects new documents from information sources, matches the documents against the queries, and continuously updates the users with relevant information.

This paper is a retrospective of the Stanford Information Filtering Service (SIFT), a system that as of April 1996 was processing over 40,000 worldwide subscriptions and over 80,000 daily documents. The paper describes some of the indexing mechanisms that were developed for SIFT, as well as the evaluations that were conducted to select a scheme to implement. It also describes the implementation of SIFT, and experimental results for the actual system. Finally, it also discusses and experimentally evaluates techniques for distributing a service such as SIFT for added performance and availability.

Note to Referees: This paper contains material from three earlier conference publications [YGM94b, YGM95b, YGM94a]. The material has been extensively updated and integrated into this paper. This paper also includes new material and results. Our desire is to make this current paper the final SIFT publication, bringing together in an archival journal, a complete summary of the project and results.

1 Introduction

Technological advances have made wide-area information sharing commonplace. Users gain easy access to information, but at the same time, they are faced with an information overload. It is difficult to stay informed without sifting through huge amounts of incoming information. A mechanism, called *information dissemination*, helps users cope with this problem. In an information dissemination system, a user submits a long-term *profile* consisting of a number of standing queries to represent his information needs. The system then continuously collects new documents from underlying information sources, filters them against the user profile, and delivers relevant information to him.

This paper is a retrospective of one information dissemination service, the Stanford Information Filtering Tool (SIFT). SIFT was one of the first dissemination services to be used by large numbers of people on the Internet. It has gone from a small experimental service, started in February 1994, to a reasonably large one with 18,400 daily users and 40,100 long-term profiles as of April 1996, to a commercial system operated by a start-up company, Sift Inc. (`http://www.sift.com`). In April 1996, its rate of growth was approximately 680 users per month and 1,500 profiles per month. At the same time, the number of documents it processes has gone up from 30,000 initially, to 80,000 documents per day in April 96.

The SIFT system collects USENET Netnews and articles from various mailing lists, and accepts profiles from users via email or Web forms. The profiles can be expressed using two standard information retrieval query languages. For example, a user may submit the profile "gateway AND Oracle" (assuming Boolean queries are used), and any article received by the server that contains these words will be forwarded to the user. The user will receive related articles from expected sources (e.g., the database interest group in netnews), and will also receive articles that may have been missed with more limited news reading systems, perhaps an article in a hospital newsgroup where a gateway to an Oracle patient database is being evaluated.

A unique feature of SIFT is that its development has been accompanied by basic research into the efficient matching of profiles and documents. As the system became heavily used, we discovered it was taking more than 24 hours to process a day's worth of documents, clearly not an acceptable situation. At that point, we developed new indexing mechanisms for profiles, carefully analyzed them to select the best one, and implemented the improved scheme. As users demanded more flexible languages for profiles, we had to extend SIFT and its new indexing schemes to handle Vector Space Model (VSM) queries. As users started complaining abut delivery of duplicate documents, we developed and implemented mechanisms for duplicate suppression. Finally, we also explored ways to distribute the profile matching work across multiple servers. Distribution has not yet been incorporated into the operational SIFT, but we did perform some analysis and experimentation to be ready to move to a distributed scheme when the load requires it.

In this paper we describe the operational features and architecture of SIFT, some of the research that guided its development, and some of the experimental results obtained. We start in Section 2 by briefly reviewing some of the work that SIFT builds upon. In Section 3 we give an overview of SIFT and its architecture, providing a brief description of how a user interacts with the system. In Section 4 we then study the problem that is at the heart of SIFT: matching a set of standing profiles against a stream of incoming documents. We show how to build indexes on the profiles, for efficient matching. Although these are similar to the more traditional document indexes used by information retrieval (IR) systems, we show that there are important differences and new types of indexes. Although SIFT handles both Boolean and VSM profiles, due to space limitations, in this paper we only discuss VSM profiles and their indexes. (The paper [YGM94c] studies the Boolean profile case.)

Selective dissemination is a "naturally centralized" problem: all profiles have to be matched against all new documents, and this is more easily done at one central server. But as we all know, centralization may create bottlenecks and critical failure points, so eventually a service like SIFT will have to be distributed, either to handle higher loads, or to provide a more reliable service. In Section 5 we explore how to distribute the matching process. We present a number of options, and analyze their performance and reliability. We also report on some experiments with an early distributed SIFT prototype, which confirm that the analytical results are valid.

2 Related Work

The idea of information dissemination has been around for a long time in the library science and information retrieval research communities [Sal68]. The proceedings of the annual Text Retrieval Conference (TREC) [Har93, Har94, Har95] are a good resource for recent results. The main focus of those efforts was on filtering *effectiveness*, with the goal of providing fine-grained interest-matching using information retrieval (IR) [Har93, Har94, Har95, WF91], rule-based [Mal87], and artificial intelligence [She94, Ste93, BS95] techniques. Often these efforts involve a relatively small number of users and thus the need to provide *efficient* filtering is not apparent. However, some work has focused on the *efficiency* aspect, i.e., how to quickly answer queries on a large set of documents, e.g., [BL85, Per94, Bro95]. In this paper our focus is exclusively on the efficiency of dissemination. Since we must handle large number of standing queries (and not necessarily large numbers of documents at a time), the index structures we use are different than those for IR. However, our techniques are analogous to the IR ones (see especially [BL85, Per94]).

A simple kind of information dissemination service has been available on the Internet for years: mailing lists (see e.g., [Kro92]). Hundreds of mailing lists exist, covering a wide variety of topics. The user subscribes to lists of interest to him and receives messages via email. He may also send messages to the lists to reach other subscribers. A major problem with mailing lists as an information dissemination mechanism is that it provides a crude granularity of interest matching. A user whose information need does not exactly match certain lists will either receive too many irrelevant or too few relevant messages. The USENET News (or Netnews) system (see, e.g., [Kro92]), an electronic bulletin board system on the Internet, is similar in nature to mailing lists. While Netnews is extremely successful with millions of users and megabytes of daily traffic, it is not very effective as a dissemination system. Like mailing lists, the coarse classification of topics into newsgroups means that a user subscribing to certain newsgroups may not find all articles interesting, and also he will miss relevant articles posted in newsgroups that he does not subscribe to.

The Boston Community Information System [GBBL85] is an experimental information dissemination system developed at MIT. It supports IR-style, keyword-based profiles. The system broadcasts all new information via radio channel to all users, who then apply their own filters locally. This style of dissemination is appropriate where broadcasting is inexpensive, but may not be efficient in a point-to-point communication medium such as the Internet.

The Tapestry system [GNOT92] is a research prototype developed at Xerox PARC. Unique to Tapestry, it uses the relational model for matching user interests and documents; filtering computation is done not on the properties of individual documents, but rather on the entire append-only database of documents. Techniques are presented for rewriting monotonic relational queries into incremental queries. Periodically these incremental queries are evaluated against the database. Their focus is on relational queries, while we are interested in IR-style queries run against unstructured documents. Reference [Ter92] considers a distributed information filtering system in the context of Tapestry. A number of document-query replication schemes were proposed, which fit into the general framework of quorum protocols in Section 5. The schemes proposed were not evaluated in [Ter92].

Oki et al. [OPSS93] propose a Publish-subscribe model in the Information Bus architecture for building distributed systems. In this model publishers send out data and event updates in real-time to authorized subscribers. The model has been employed in commercial systems such as Tibco and Vitria. These systems focus on supporting dissemination in distributed systems using different communication protocols (point-to-point, broadcast). Our work takes on a data management perspective, aiming to devise efficient indexing and query processing techniques for dissemination.

Imielinski and Viswanathan [IVB94] considers data filtering in wireless environments, with the goal of power conservation. Acharya et al. [AFZ96] proposed Broadcast Disks, a model for disseminating data in a broadcast medium. Techniques such as propagation and prefetching are used to improve performance. Our approach focuses on a point-to-point environment, where broadcasting of data to all clients is prohibitively expensive.

A recent Data Engineering Bulletin Special Issue on Data Dissemination [Edi96] is a good source of references to unique applications of dissemination in different environments (e.g., hybrid satellite/terrestrial networks [DP96], application development [Gla96], traveler information systems [SFL96], and wireless systems [FZ96]).

Some recent commercial dissemination systems have emerged including Pointcast and Marimba. Pointcast [Poi97] is an Internet-based dissemination system that allows a user to select categories of interest from a predefined set. The user is then continuously updated with new data items that fall into the selected categories. SIFT supports full-text filtering where there are no predefined categories. Marimba [Mar97] uses the dissemination model as an application framework. New versions of application software are automatically disseminated to clients, alleviating the user of software maintenance and installation work.

3 The SIFT System

In this section we present an overview of the SIFT system. We describe here the original experimental system, whose code is in the public domain (the code is available at URL:

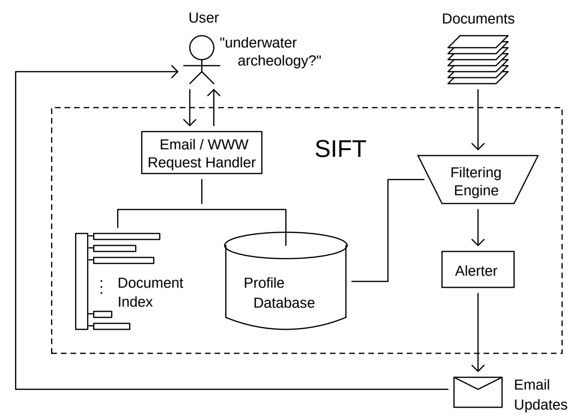

Figure 1: An Overview of SIFT

`http://db.stanford.edu/sift-1.2-netnews.tar.Z`).

SIFT was implemented in C and has been compiled on several Unix platforms: DEC Ultrix 4.2, HPUX 8.07, and SunOS 4.1. We do not describe the commercial version of SIFT, which has several proprietary enhancements. Figure 1 shows the architecture of SIFT.

3.1 User Interactions with SIFT

A user subscribes to a SIFT server with one or more profiles, one for each topic of interest. Each profile includes a query, and additional parameters to control the frequency of notification, the amount of information to receive (e.g., how many lines of matching documents to get in notification email), and the time duration for the profile. A profile is identified by the email address of the user and a profile identifier.

SIFT supports two IR models for queries: *Boolean* and *Vector Space Model (VSM)*. Using the Boolean model, the user may specify words that he wants to appear in documents he receives, and words to be excluded. For example, the boolean query "fly fishing not underwater" is for documents that contain both words "fly" and "fishing" but not the word "underwater." The SIFT Boolean model only allows conjunction and negation of words; however, the user may approximate disjunction semantics by submitting multiple profiles (though a document may match more than one profile).

With VSM queries, the user simply provides a set of words of interest. The user is then given documents that are "similar" based on the commonality of words. In Section 4 we define precisely what similarity means. The user can also specify a *relevance threshold*, which is a number between 0 and 1 indicating the strength of the desired similarity between the query and a document to be delivered. (A value of 1 indicates the highest possible similarity.) If the user does not provide a threshold, the system uses a default value (that seems to work well for general users).

To assist the user with the construction of a query, a SIFT server provides a test run facility. The user may run his initial query against an existing, representative collection of documents to test the effectiveness of his query. He may interactively change the query and (for VSM queries) adjust the threshold to the desired level. When he is satisfied with the performance of the filtering, he may then subscribe with the selected settings.

A user can deliver a SIFT profile and its associated parameters in two ways. With the email interface, users send in a message that lists the query and its parameters. The message contains a set of label-value pairs, using the format of [Coh92]. For example, if a user wishes to receive notification once every 7 days, he includes the pair "PERIOD 7" in a line. With the Web interface, the user fills in a form with the required information and submits it to SIFT. Even though forms are very popular today, a lot of SIFT users only have email access to the Internet and use the email interface.

After the user receives some periodic notifications, he may decide to modify his query or change the threshold for VSM queries. He may do this again via email of a Web form. Furthermore, for VSM queries, *relevance feedback*, a well-known technique in IR to improve retrieval effectiveness, can be used. The user simply gives SIFT the documents that he finds interesting; after examining them, the server adjusts the weights of the words in the user's standing query accordingly.

3.2 SIFT Implementation

After describing how a user interacts with SIFT, let us now present the implementation of SIFT (refer to (Figure 1). The email request handler parses incoming messages, and enters or modifies profiles in the profile database. The Web request handler is a "cgi-script" for use with the HTTP daemon released by the National Center for Supercomputing Applications. It accepts requests submitted by users using a web browser (with a form-filling graphical interface). Once the Web handler has a request, it proceeds like the email handler.

If the user requests a test run, then SIFT evaluates the query against a "test" index. The test-run index is built overnight using USENET news articles collected from our department's news host. SIFT uses the WAIS search engine to build the test index. In our experience it is a very robust implementation. However, since we use a different scoring scheme than WAIS, we made slight modifications to the WAIS code to make it compatible with our filtering engine (i.e., giving the same results).

SIFT collects new articles on a daily basis. Our source of USENET News is from our department's news host (whose disk is mounted on our SIFT server through NFS). For a time, SIFT was also collecting articles from thousands of mailing lists. SIFT itself worked fine in this case, but unfortunately, the amount of traffic flowing through our group's mail server was so high that the server crashed several times and our local users lost some of their mail. Given the "uproar" we decided to turn off the mailing lists. (A separate instance of SIFT collected bibliographic entries for new Computer Science technical reports; this was part of the DARPA-funded CS-TR Project. See `http://www.ncstrl.org`.)

Everyday SIFT obtains a list of newly arrived articles (in the form of Unix path names). The filtering engine (Figure 1) reads the articles one by one, screening out binary files, and and matches the remaining text articles against the stored queries. This matching uses some of the index structures that will be studied in Section 4. The filtering engine outputs two files of matchings (a matching is essentially a pair of user email address and article path name). One file (the "daily" file) contains matchings for profiles that require daily delivery, and the other "non-daily" file contains all the other matchings. (Most of the profiles are daily ones.)

After the filtering is completed, the "daily" matching file is sorted by users and profile identifiers. This is necessary because SIFT sends out updates on a per profile basis. After sorting, the alerter sends out the messages one by one. Included in the messages are excerpts from the matched documents (a few lines from the beginning of the document).

The "non-daily" file is merged with a master file that accumulates non-daily matchings. The file is sorted by users and profile identifiers. If a profile reaches its notification period, then SIFT will send out its matchings and delete them from the master file.

A more recent version of SIFT can also deliver matchings by preparing a "personal" Web page of matches for each user. With this option, the matchings for a user are not mailed out. Instead they are saved in a file for a number of days. When the user connects to SIFT, the system reads his matchings and dynamically prepares a Web page listing them. For each matching, the URL of the USENET article is given, so the user can read it if interested.

In Section 4.3 we evaluate how SIFT performs in practice. Before that, however, we discuss the profile-document matching process, and index structures that can be used to make it efficient. Due to space limitations, we focus exclusively on one of the query models SIFT handles, i.e., on VSM queries. We first study the VSM matching problem in a general setting, and then in Section 4.3 describe and evaluate the scheme that was actually implemented.

4 Indexing Vector Space Queries

The central component of SIFT must match a collection of profiles or standing queries against new incoming documents. Under the Vector Space Model (VSM), which is widely used in commercial search engines, documents and queries are conceptually represented as vectors. If m distinct terms are available for content identification, a document D is represented as an m-dimensional vector, $D = \langle w_1, ..., w_m \rangle$, where w_i is the "weight" assigned to the i-th term and is 0 for terms not present in D. For example, the document $D_1 = \langle 0.5, 0, 0.3, ..., \rangle$ contains the first word in the vocabulary (say by alphabetical order) with weight 0.5, does not contain the second word, and so on. The weight for a document term indicates how statistically important it is. One common way to compute it is to multiply the term 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*.

Queries in the VSM model are also represented as vectors over the term space, $Q = \langle z_1, ..., z_m \rangle$, where each entry indicates the importance of the term in the search. In SIFT, as in most VSM systems, queries are written by a user in natural language. In this case, z_i is the product of number of times the i-th term appears in the query string times the *idf* factor for the term. (The *idf* factors are the same as for documents.)

Sometimes we follow the convention of writing a document or query vector as a vector of (term, weight) pairs; those terms not listed have weights equal to 0. Thus, a query Q with k non-zero weighted terms can be written as $Q = \langle (y_1, z_1), ..., (y_k, z_k) \rangle$. For instance, in the query $Q = \langle (\text{"queue"}, 0.93), (\text{"system"}, 0.37) \rangle$, term "queue" has a weight 0.93, "system" has 0.37, and all other terms have a zero weight.

We can measure the degree of similarity between a document-query pair based on the weights of the corresponding matching terms. The cosine measure has been used for this purpose; given a document $D = \langle w_1, ..., w_m \rangle$ and a query $Q = \langle z_1, ..., z_m \rangle$, the cosine similarity measure is:

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

(The value $||D||$ is the norm of the vector.) Notice that by definition similarity values range between zero and one, inclusive. Below we assume that the document and query vectors are normalized by their lengths; thus the above simplifies to:

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

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 similarity between the query and the documents. In an information dissemination setting, a query 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 [FD92], 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 [FD92] indeed reports such drawbacks.

An alternative, as suggested in [FD92], 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, documents can be processed one at a time, as soon as they are received. Also, the precision and recall of the filtering are independent of when it is performed. Interestingly, such a relevance threshold can also be used in conventional information retrieval; [Sal91] describes such an experiment. We sum up this discussion with the following definition.

Definition 1: For a query Q and relevance threshold θ, a document D is *relevant* if $sim(D, Q) > \theta$. □

Another important difference between an IR setting and a dissemination one is what constitutes the *idf* factor for terms. In IR, *idf*'s are defined in terms of the collection of documents we are searching. For instance, the *idf* of term x can be computed as a function of the number of documents in the collection that contain x. In a dissemination setting, however, there is no fixed document collection. One solution is to use as the reference collection some set of recently processed documents. Note that it is possible to receive a new document with a term for which we do not have an *idf*. In that case, we can assign it a high value, equivalent to the highest known *idf*, for example.

SIFT uses the threshold-based definition of relevance given above. As discussed in Section 3, SIFT users can provide their relevance threshold when they define VSM queries, and they can also use relevance feedback for query refinement. It is important to note that the indexing schemes we describe next do not depend on whether relevance feedback is used, nor on the specific ways term weights for queries and documents are computed, nor on how the relevance threshold is derived.

4.1 Indexing Vector Space Queries

In this section we present three query indexing methods for VSM queries. The first, the *Brute Force* Method, uses no index structures; it is mainly included for comparison, although it is used by some dissemination systems. The second scheme (QI) is the one implemented in SIFT, while the third (SQI) is an enhanced scheme that was not implemented. After introducing the schemes, we describe the analysis we performed that lead to our selection for SIFT.

4.1.1 Brute Force (BF) Method

Suppose we store long-term queries sequentially on disk without any index structures. In that case, all queries must be evaluated when a new document is received. We call this the Brute Force (BF) Method.

To process the incoming document, we first compute its vector representation. We then examine each query in turn. For each (term, weight) pair (x, z) in a query, we find x's weight w in the document vector, and calculate the product $w \times z$. The sum of such products is the cosine similarity measure. The document is relevant to a query if the cosine measure is greater than the relevance threshold associated with the query.

The only disk data structure used by the BF method is a sequential file of queries. In this file, each query is stored as a variable-length *record* with these fields: the query identifier, the length – i.e., the number of terms in the query, the (term, weight) pairs, and finally the relevance threshold.

4.1.2 Query Indexing (QI) Method

To reduce the number of queries that must be examined, we can build an index for the long-term queries. We call this the *Query Indexing (QI)* method. This is analogous to what traditional IR systems do, except that here we are indexing the queries and not the documents. Processing with this index is also analogous to IR processing, with the role of queries and documents reversed. However, there are two differences: (1) We now have to deal with query-specific thresholds, and (2) Performance can be quite different because what we are comparing against the index (documents) typically contains many more terms that what an IR system compares (queries).

Under QI, for each term x, we collect all the queries that contain it and build an inverted list. The list is made up of *postings*; each contains the identifier of a query involving x and the weight of x in it. Thus, a query with k terms will be found in k postings; each posting in a different list. When processing a document D, we only need to examine those queries in the inverted lists of the terms that are in D. We also construct a *directory*: this is a data structure that, given a term x, gives us a pointer to the posting list for x.

To match a document against these queries, 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 queries the system handles. A query has an entry in each of the arrays: the THRESHOLD entry stores the relevance threshold, and the SCORE entry is used to keep the score of the query.

When a document D arrives, we initialize the SCORE array to all 0's. (The THRESHOLD array only

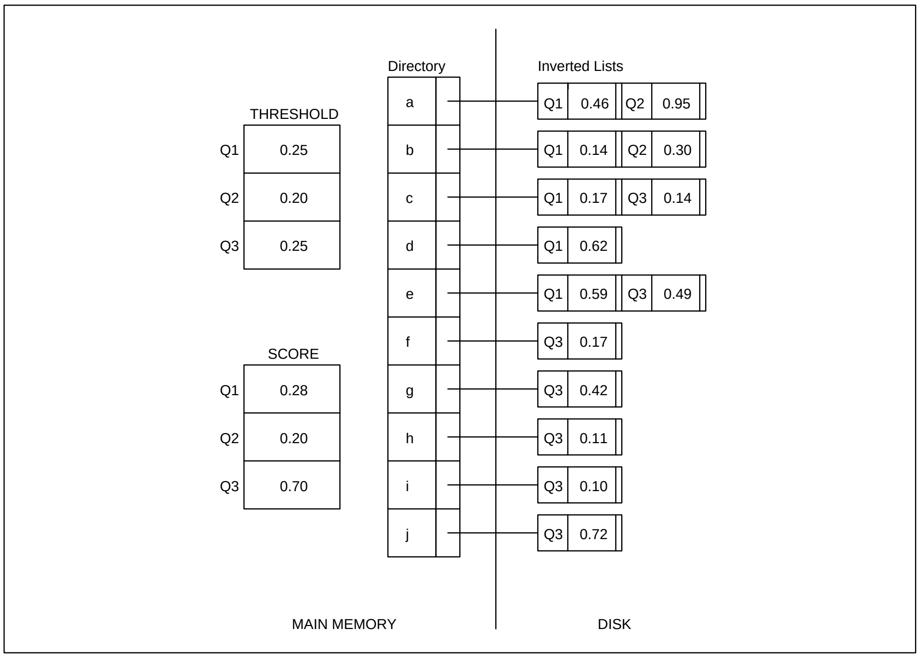

Figure 2: Data Structures for the QI Method

needs to be initialized at system start-up.) For each term x with weight w in the document, we use the directory to retrieve x's inverted list. Then we process each query Q in the list. That is, if the weight of x in Q is z, we increment SCORE[Q] by the product of $w \times z$. After all document terms are processed, a query whose SCORE entry is greater than the THRESHOLD entry matches the document.

To illustrate, consider three queries:

$$\left| \begin{array}{l} Q_1 = \langle (a, 0.46), (b, 0.14), (c, 0.17), (d, 0.62), (e, 0.59) \rangle \\ Q_2 = \langle (a, 0.95), (b, 0.30) \rangle \\ Q_3 = \langle (c, 0.14), (e, 0.49), (f, 0.17), (g, 0.42), (h, 0.11), (i, 0.10), (j, 0.72) \rangle \end{array} \right| \begin{array}{l} \theta_1 = 0.25 \\ \theta_2 = 0.20 \\ \theta_3 = 0.25 \end{array} \right|$$

The inverted index for these queries is shown in the right-hand side of Figure 2. For example, the a list contains the postings for Q_1 and Q_2. The 0.46 value in the first entry in this list is the weight of a in Q_1. Now suppose this document arrives:

$$D = \langle (a, 0.17), (b, 0.15), (d, 0.32), (f, 0.21), (h, 0.14), (j, 0.90) \rangle.$$

Suppose we read the a list. We increment the SCORE entries of Q_1 and Q_2 by $0.17 \times 0.46 = 0.0782$ and $0.17 \times 0.95 = 0.1615$ respectively. The lists of b, 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 Q_1 and Q_3.

4.1.3 Selective Query Indexing (SQI) Method

In the QI method, we index a query by all its terms. In this subsection we investigate an alternative in which we only select a number of terms for indexing. We call this the *Selective Query Indexing (SQI)* method. This is a new scheme that is not used in traditional IR systems.

The QI method indexes every term in a query without regard to its weight. The SQI method chooses to index only the more "significant" terms (which are typically terms that appear less frequently in documents), so that the processing of a document may become less expensive. This is done with no sacrifice in accuracy when computing the relevancy of a document to a query.

To motivate SQI, consider the term b in Q_1 in our running example. Suppose a document arrives and it does not contain the terms a, c, d, or e. The maximum score Q_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 query with the term b — a document that contains only b and no other terms in the query will not be relevant anyway. However, a document that contains b and another term in the query may be relevant; so we need to duplicate $(b, 0.14)$ in the postings of the other terms in their respective lists. (If 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. In the analysis that follows, we consider a scenario where the inverted lists do not fit in memory, and hence we assume that pairs are duplicated.)

Similarly, consider the subvector $\langle(h, 0.11), (i, 0.10)\rangle$ in Q_3. Suppose a document arrives that does not have the other terms in Q_3. Then an upper bound to the similarity between Q_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 query in the inverted lists of h and i and duplicate the pairs in the postings of the other terms in the query. These observations lead us to this definition.

Definition 2: Given a query vector $Q = \langle(y_1, z_1), ..., (y_p, z_p)\rangle$, a subvector $Q_s = \langle(y_{i_1}, z_{i_1}), ..., (y_{i_s}, z_{i_s})\rangle$, $1 \leq i_1 < ... < i_s \leq p$, is *insignificant* at a threshold of θ if for any document D, $sim(D, Q_s) \leq \theta$. □

Given a query like Q_3, there may be several insignificant subvectors, e.g., $\langle\ (h,\ 0.11),\ (i,\ 0.10)\rangle$ is one, $\langle\ (c,\ 0.14),\ (i,\ 0.10)\ \rangle$ 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 query vector $Q = \langle(y_1, z_1), ..., (y_p, z_p)\rangle$, a subvector $Q_s = \langle(y_{i_1}, z_{i_1}), ..., (y_{i_s}, z_{i_s})\rangle$,

$1 \le i_1 < ... < i_s \le p$, is *most insignificant* at a threshold of θ if it has the largest number of terms among the insignificant subvectors at a threshold of θ. $\square$

Assuming *idf*s are distinct, a query 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 query subvector and any document vector. Intuitively, we can see that the similarity between a query subvector and any unit document vector is highest when the document vector is "in the same direction" as the query subvector. And if that happens, the similarity is given by the magnitude of the query subvector. This is formally stated and proved as follows.

Theorem 1: For any Q and any D, $||D|| \le 1$, $sim(D,Q) \le ||Q||$.

Proof: This follows easily from the Cauchy-Schwarz Inequality [FIS89]:

$$sim(D,Q) = D \cdot Q \le |D \cdot Q| \le ||D|| ||Q|| \le ||Q||. \quad \square$$

To find the most insignificant subvector of a query vector, we first sort the terms by *idf*. Then we include in the subvector as many low *idf* terms as possible without having the magnitude of the subvector exceed the threshold. For example, consider Q_3 again. Let us assume that the user typed each term $c, e, f, ..$ once; in this case, the weight of each term in Q_3 happens to be the *idf* of that term. Thus, term i is the rarest in documents, h is the next rarest, and so on. As

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

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

$\langle(c, 0.14), (h, 0.11), (i, 0.10)\rangle$ is the most insignificant subvector of Q_3 at a threshold of 0.25. This also shows that Theorem 1 provides a stronger way of finding insignificant subvectors than the naive way of finding an upper bound by simply adding the weights, as done earlier.

With this knowledge, we can indeed index the queries selectively. For each query, we find the most insignificant subvector at the threshold specified. The query 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 duplicated in the lists of all the significant terms. Each posting contains the query 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 QI 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 a query Q in the x list, suppose the weight of x in Q is z, and the insignificant pairs

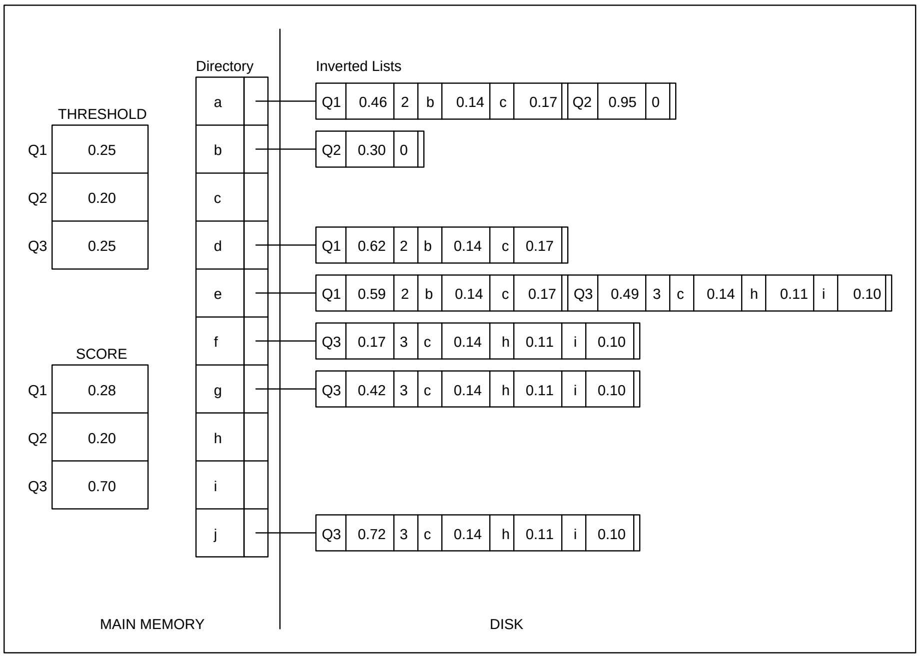

Figure 3: Data Structures for the SQI Method

are (y_{i_1}, z_{i_1}), ..., (y_{i_s}, z_{i_s}). We examine Q's SCORE entry. There are two cases: if the SCORE entry is zero, we first add the product $w \times z$. 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 $w_{i_j} \times z_{i_j}$ 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 $w \times z$. After all document terms have been processed, a query 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 pair (b, 0.15) from the document vector. The list of b has only one posting, that of Q_2. We add the product 0.15 $\times$ 0.30 = 0.045 to Q_2's SCORE entry. As there is no insignificant subvector, we are done with this posting and also with the b list. Next suppose we process the pair (d, 0.32). Only Q_1's posting is in the d list. First we add the product 0.32 $\times$ 0.62 = 0.1984 to SCORE[Q_1]. Then we process the insignificant subvector $\langle$ (b, 0.14), (c, 0.17) $\rangle$. To do this, we look up the term b in the document vector, getting a weight of 0.15. Thus we increment SCORE[Q_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.

4.2 Performance Evaluation

Given that there are multiple ways to index queries and the various tradeoffs involved, we decided to evaluate the choices before implementing one of them in SIFT. In this section we describe the framework for the evaluation; the full details can be found in [Yan95]. Then in the following two sections we present some of the results and conclusions.

4.2.1 Document and Query Models

To model documents, we assume that their terms are drawn from a vocabulary V of size v. The probability that a term appears in a document is described by Zipf's Law [Zip49]. For convenience, we represent each term in V by an integer whose rank reflects the term's probability of appearance. Thus, if $x < y$, then term x is more likely to appear in a document than term y. We model each document as a set of k_{doc} terms, each selected from V following Zipf's Law.

While the frequency distribution of terms in large collections is known to follow Zipf's Law, there is very little published about the frequency distribution of terms in queries, and even less about terms in long-term queries as used in dissemination. At the time we conducted our evaluation, SIFT did not have a large body of standing queries, so we decided to use a synthetic distribution, the same one used in [TGM93]. This query model assumes that extremely infrequent terms (in documents) typically are misspellings or typos. Thus, it assumes that queries do not use these terms. At the same time, some very frequently occurring words, called stop words, have very little content-discrimination power, and are typically dropped before processing. This is all modeled by assuming that query terms are chosen from the set $U = \{u_{start}, ..., u_{end}\}$, termed the queried vocabulary, out of the vocabulary $V = \{1, ..., v\}, 1 \leq u_{start} < u_{end} \leq v$. (Recall that we are identifying terms by their ranks.) It is further assumed that each term in U is equally likely to be chosen for a query. Hence, we assume that a query is a set of k_{query} terms chosen randomly without replacement from the queried vocabulary U. The number of queries in the system is n_{query}. (In retrospect, the queries received by SIFT do follow these general trends, and hence we believe that the model was adequate for studying the basic tradeoffs.)

The framework we have described is fairly simple, but we believe it is the right type of model for evaluating differences between implementation options. It is not the right model for predicting the actual performance of a particular implementation, but this was not our task at hand. However, even for a simple model, it is important to have realistic base parameter values, so we focused our efforts on computing parameter values that would describe the intended SIFT scenario. From these base values, we then conducted sensitivity analysis to learn how SIFT would react to a changing environment.

The base parameter values we used were based on a database of Netnews (text) articles we collected from

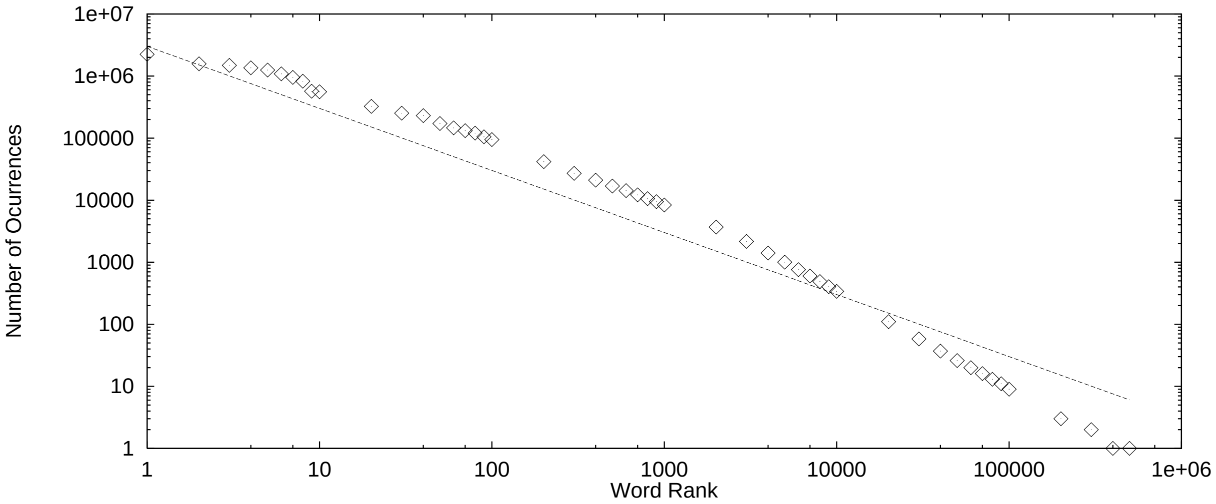

Figure 4: Term Rank vs. Term Frequency Graph for Netnews Database

our USENET News host during the period of April 22 to April 29, 1993. A total of 212,972 articles were collected, making up a 550MB database.

To study the occurrence frequency of terms in this collection, we first carried out a lexical analysis to screen out all non-alphabetical characters from the documents (i.e., articles). Then a stemming routine (Porter's algorithm [Por80]) was run to reduce the remaining words to word-stem form. Each stem thus obtained is a term. Next we measured the occurrence frequency of each term in the database, obtaining the plot shown in Figure 4 (note the log/log scale). The x-intercept (i.e., size of the term vocabulary) is found to be 521,915. The straight line in the graph was derived by curve fitting using [Wol91]. We can see the database does demonstrate Zipfian characteristics [Zip49]. Also, the average number of terms per document is found to be 323.

From these data, we set the following parameter values. We set the size of a document, k_{doc}, to 323 terms. The vocabulary size (v) was set to 521,915. For the queried vocabulary size, we assumed a stop-list made up of the top 100 words (i.e., $u_{start} = 100$). A base value of 50,000 was chosen for u_{end}, covering more than 97% of the total occurrences of terms in the Netnews database.

The vector representations of the documents and queries were computed as described earlier. The exact formulas used to compute the weight of a term x_i are from [Sal91].

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

Parameter	Base Value	Description
v	521,915	size of vocabulary
k_{doc}	323	# term occurrences per document
u_{start}	100	end of stop list
u_{end}	50,000	end of queried vocabulary
n_{query}	300,000	# queries
k_{query}	5	# terms per query
θ	0.2	relevance threshold
s_{qid}	4	# bytes for query identifier
s_{length}	2	# bytes to represent length of query
s_{term}	4	# bytes to represent a term
s_{float}	4	# bytes to represent a floating point number
s_{block}	512	# bytes in a disk block

Table 1: Model Parameters for Performance Evaluation

where f_i is the frequency of term x_i in the document. The fraction of documents with term x_i can be computed directly from the Zipf distribution.

Our last challenge was to model the relevance threshold distribution. For a user, a suitable relevance threshold for his query depends on the individual query 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?) However, in keeping with a simple model, we assumed that the relevance threshold is fixed for all queries. This allows us to study clearly its impact on the indexing methods.

A reasonable base case value for the threshold was found by the following procedure. First a random document was generated. Then a query was created to contain a number of terms randomly selected from the document. The similarity between the document and the query was computed. The procedure was repeated a large number of times. For a base case query length of 5, we found that a query 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.

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. Finally, note that our model is a static one, i.e., we do not model how queries, thresholds, and documents change over time, nor the cost of updating the index structures to reflect the changes.

4.2.2 Performance Evaluation Results

The results for the base case are given in Table 2. These results, and others in this section, are obtained analytically except for the result for the SQI method. For that method, we actually simulated the construction of indexes to measure their size and performance. (See [YGM94b] for details.)

Table 2 shows the size of the disk data structures for each scheme. The "contiguous" column refers to the space required to pack all inverted lists next to each other. This is clearly more space efficient, but is harder to achieve if the indexes are updated in place. Notice that scheme QI actually use less disk space than BF. This is simply because QI (and SQI) hold the query thresholds in memory. (Of course, even for QI and SQI, the thresholds need to be on disk for fault tolerance. However, this copy of the thresholds is in a query record that holds all information for a query, such as the subscriber. We assume these records exist for all schemes, and are not taken into account in Table 2.) However, it is important to keep in mind that schemes QI and SQI do use more main memory than BF. Scheme SQI requires more space than QI because some (term, weight) pairs are replicated in lists.

The "fragmented" column refers to the case where inverted lists do not share disk blocks. If a list only partially uses a block, the rest of the block is left empty, which is useful for future growth. Since scheme BF does not use inverted lists, this option does not apply to it. The fragmented option is more reasonable for a system like SIFT, where we do not wish to rebuild the query index as new queries are submitted. Fragmentation does significantly increase space costs, (about 68% for SQI and 113% for QI). However, the important thing to notice is that QI and SQI are now roughly comparable. Scheme SQI does have fewer inverted lists, but they tend to have more data in them, so overall it is a wash. The number of I/Os for the fragmented and contiguous organizations are the same, since the same number of block reads is required to process a document no matter whether the blocks are shared or not. Scheme BF does use about half the disk space (and less main memory), but when one considers the number of I/O's needed to process one document (third column in Table 2), we see that the extra storage space is clearly an excellent investment.

The last column in Table 2 estimates the number of floating point multiplications that each method requires for processing one document. This metric is useful for comparing the overall CPU overhead, and is especially important in a CPU bound system (including the case when a large portion of the data structures can be cached in main memory). The SQI method is best in this category because frequent terms in a query are not indexed.

As we have stated, it is also important to study how performance varies as parameters evolve from our base settings. In [YGM94b] we present a number of sensitivity analyses; here we only show a subset to

Method	Size (Blocks) Fragmented	Contiguous	I/Os	Multiplications
Brute Force	–	29,297	29,297	4,314
Query Indexing	49,900	23,438	144	4,314
Selective Query Indexing	49,804	29,630	127	3,434

Table 2: Results for Base Case Performance Evaluation

illustrate sensitivity to two critical parameters, the number of standing queries, and the relevance threshold. (For the results that follow, all parameter values are as in Table tab-vsm-para unless otherwise noted.)

In Figure 5 we show how disk space grows as we vary the number of queries from 100,000 to 800,000. With continuous allocation, space requirements grow linearly with n_{query}, and the relative performance among the schemes remains constant. However, for fragmented allocation we see a curious effect: the space required is at first constant and then increases. This is because each inverted list fits in 1 block initially, but as n_{query} increases, 2 blocks are needed to hold a list. The lists grow at a faster rate in the SQI method initially (due to the replication of insignificant terms), but QI soon catches up with it. The conclusion is that the space requirement for fragmented allocation is less predictable, as it depends on how lists happen to fit into blocks. Overall, as the number of queries grows, the overhead of fragmented allocation relative to continuous decreases, so fragmented allocation becomes more desirable.

Figure 6 shows the number of disk I/Os required per document, as n_{query} grows. (Note that the vertical axis is truncated.) Results for BF are omitted since the values are extremely high (two orders of magnitude higher than the QI and SQI methods). We see there is a range of n values where SQI requires more I/Os per document; this happens when an SQI inverted list grows faster than a QI list. When the list length becomes the same in both methods, SQI again becomes better then QI. It is important to notice that even though the relative performance of QI and SQI changes, at no time is the difference large (it is always less than roughly 20%). Also notice that the IO values shown are per document; when we have many documents, the difference between schemes will be magnified.

The next parameter that we vary is k_{query}, the number of terms per query (Figure 7 and 8). For contiguous allocation, we see the total space requirement grows linearly with k_{query} for all methods. For fragmented allocation, with shorter queries, the inverted lists each fit in one block, so the size remains constant at the queried vocabulary size. With longer queries, the lists grow in length, so the total space requirement grows also. The SQI method grows at a faster rate than the QI method. For the number of disks read, the SQI method is better for small k_{query}, but with the longer lists at larger k_{query}, its performance deteriorates sharply.

Finally, let us examine the impact of the relevance threshold on performance. Although it may not

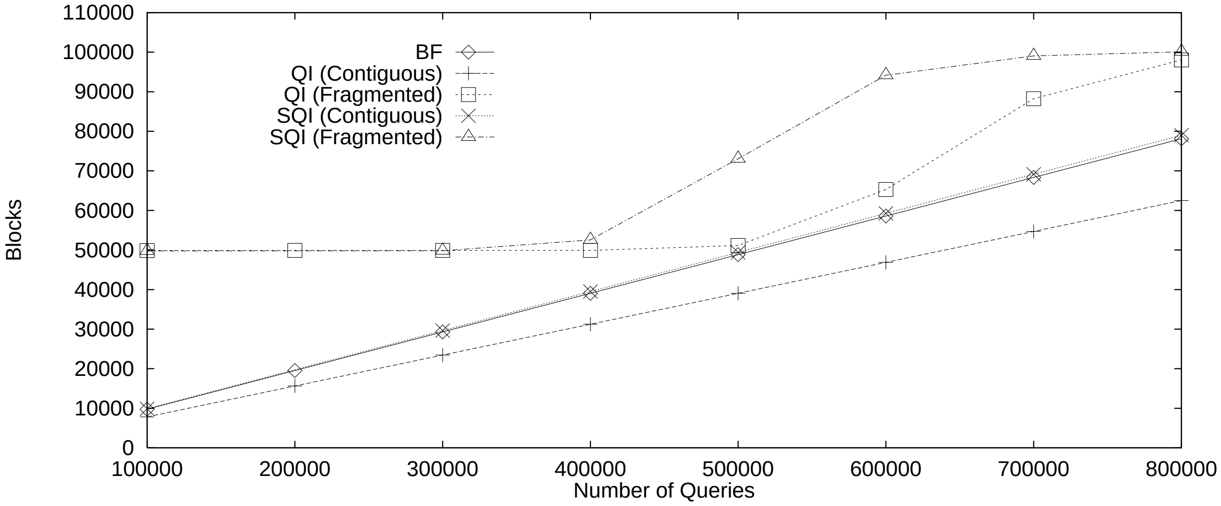

Figure 5: Total Disk Space Required vs. Number of Queries

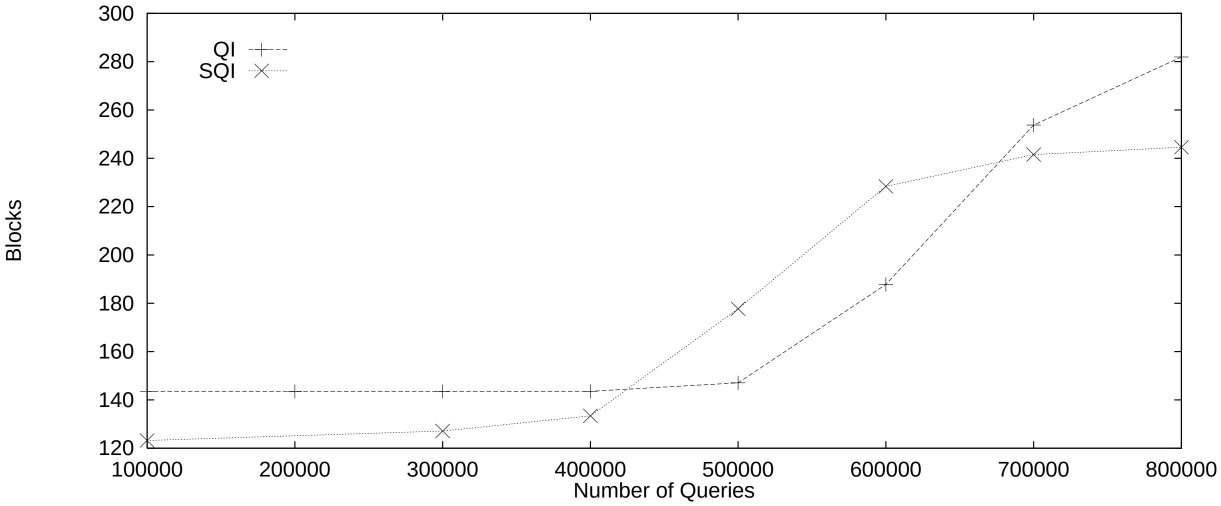

Figure 6: Number of Disk I/Os Per Document vs. Number of Queries

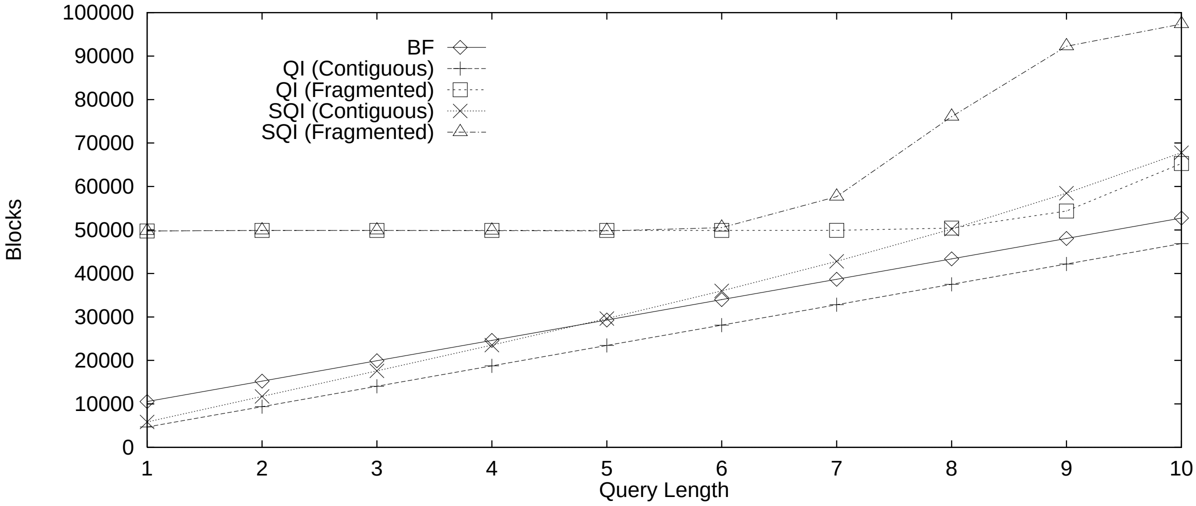

Figure 7: Total Disk Space Required vs. Query Length

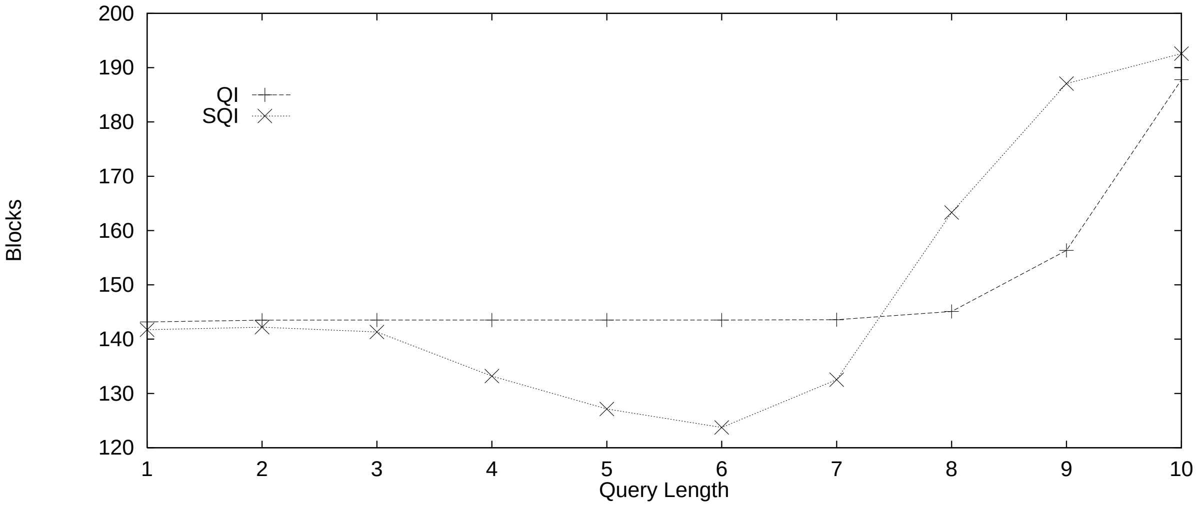

Figure 8: Number of Disk I/Os Per Document vs. Query Length

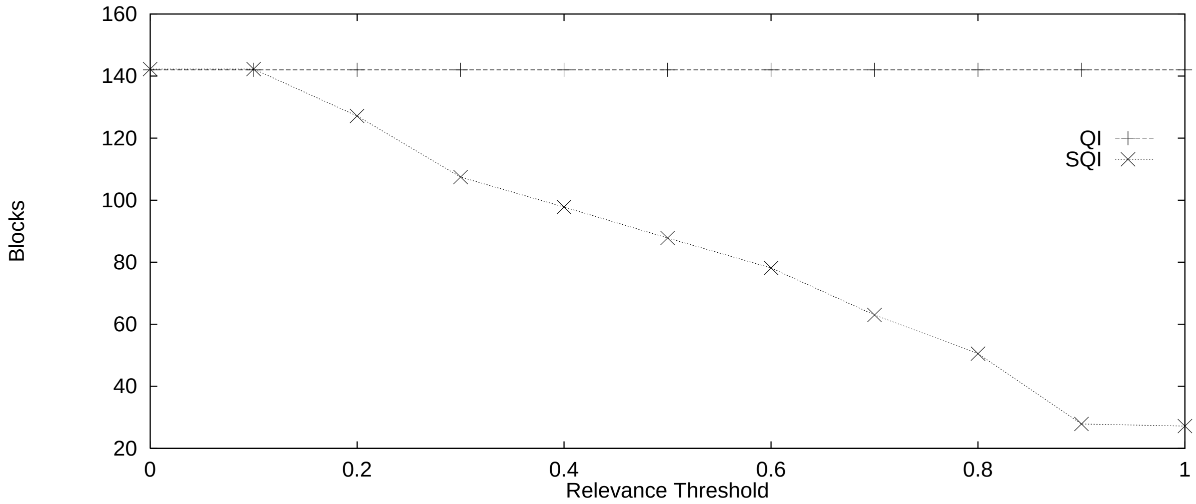

Figure 9: Number of Disk I/Os Per Document vs. Relevance Threshold

make sense to have a threshold value of 0 or 1, we study the entire range of possible values to confirm our intuition about the SQI method. The other methods are insensitive to the relevance threshold. With increasing relevance threshold, the number of I/Os required for SQI is always decreasing (Figure 9). As the relevance threshold becomes higher, more and more query terms become "insignificant." As a result fewer and fewer terms are indexed, and the indexed terms are the ones appearing very infrequently in documents. Consequently the number of I/Os required becomes less and less. Similarly, the number of multiplications decreases also.

4.3 Implementation and Evaluation of SIFT Query Indexing

From the performance evaluation results above, we can see that query indexing (QI or SQI) can reduce the amount of processing required by orders of magnitude, so clearly one of them is desirable. Scheme SQI in general performs better than QI; In terms of reduced I/O's in document processing the gains are not impressive (less than 20% in most cases) when the relevance threshold is low. On the other hand, when the relevance threshold is high, SQI outperform QI by 80% or more. Similar findings can be obtained for the amount of CPU processing,

For our SIFT implementation we selected the QI method as a compromise between ease of implementation and potential performance gains. Clearly, some type of indexing was desirable, so scheme BF was out. Since we did not have evidence that thresholds would be high, it was not clear to us that SQI would improve performance. Thus, we opted to implement the simpler QI method. SIFT also supports Boolean queries, and through an analysis similar to the one we have presented here, we selected the *Counting Method* (detailed in [YGM93]) for indexing those queries.

Since the Counting and QI method use similar index structures, in SIFT we combined them into a a single index; this way, an article needs to be run against the engine once only. Furthermore, the index was implemented fully in memory, mainly to simplify the implementation effort, but also because initially we were not expecting large numbers of queries. Also, for VSM profiles, SIFT does not use *idf* terms. This decision was made to make SIFT results consistent with the results from the "test" index. As described in Section 3, this index is implemented using WAIS, and WAIS does not use *idf* terms.

To evaluate the performance of the SIFT implementation, we conducted a series of experiments in July 1994. These experiments focused on Version 1 of SIFT, although we also evaluated Version 0 (no query index) to quantify the gains achieved by indexing. What we learned form these experiments lead to SIFT Version 2, as we will describe later.

The test data consisted of 38,000 articles received on the day of July 6, 1994 and 7,000 randomly selected profiles from the USENET News SIFT server at that time. The average article size was 269 words (a word is defined as an alphanumeric string longer than 2 characters). The actual profiles tended to be small, on average containing 1.5 query terms. The profiles were stored on a local disk, while the news articles were stored on an NFS-mounted disk (from the news host). We repeated the experiments with test data from two other days and the results were within $\pm 10\%$ of those reported below.

In the result graphs that follow, we divide SIFT processing time into these four steps:

1. *Build Time* – The query index structure is built. Other auxiliary data structures are allocated and initialized.
2. *Filtering Time* – Documents are run against the index one by one. Document-query matchings are written into a file.
3. *Sorting Time* – The document-query matching file is sorted by user email address and profile identifier, using the Unix `sort` command.
4. *Notify Time* – The sorted matching file is read. Excerpts of matchings for each profile are prepared into an email message. Unix `sendmail` is invoked to send out each message.

First we investigate the relationship between the processing time and the number of documents. Figure 10 shows the results of processing the 38,000 articles against the 7,000 profiles. (For each value on the horizontal axis, we re-ran the entire experiment from scratch, using that number of documents.) The time it takes to build the query index is very small and is as expected independent of the number of documents. The filtering time is linear with respect to the number of documents, with slope 0.21 sec./document (the value

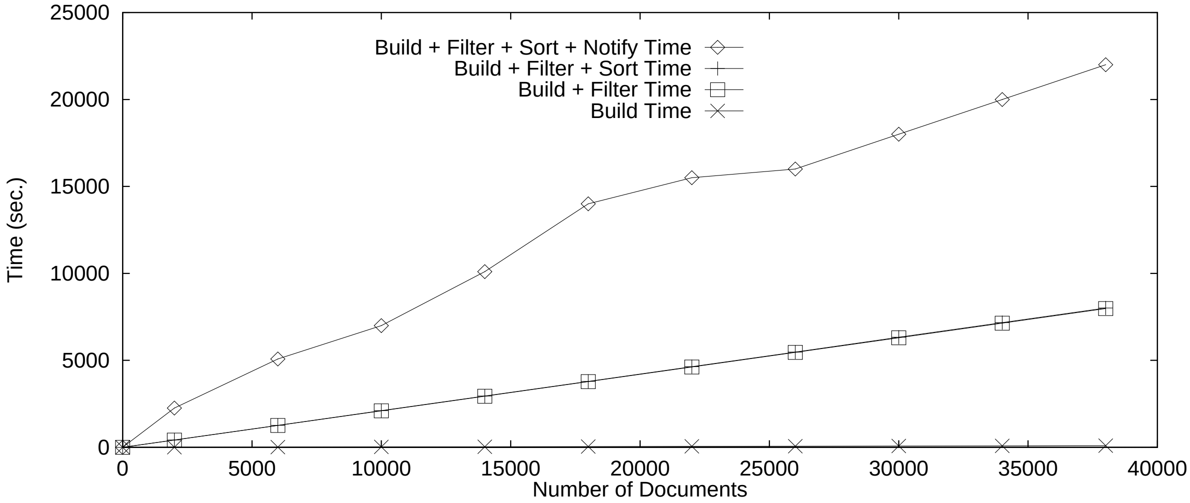

Figure 10: SIFT Performance vs. Number of Documents

is obtained by curve-fitting using Mathematica). The time it takes to sort the matching file is negligible. The proportionality constant for the total running time is 0.63 sec./document. (Recall that the articles are stored on a remote news host, so some fraction of this time is spent in reading the articles via local network. This is confirmed by looking at the CPU utilization, which is found to be 42.8%.)

The notify times shown in Figure 10 (except the rightmost data point) are interpolated results. This is because in our experiments we did not wish to actually send to our real users the same matchings repeatedly. For the interpolation, we assume that the time it takes to send out notifications is proportional to the number of matchings in the matching file. We have verified this assumption with a separate experiment and derived the proportionality constant. Then in the experiment for Figure 10, we count the number of matchings in the matching file at the data points shown. We compute the notify time as the product of the number of matchings and the proportionality constant.

The striking feature of Figure 10 is that a large fraction ($\approx$ 63% for 38,000 documents) of the total time is spent in sending out notifications to users. We return to this issue at the end of this section.

In a second experiment, we look at the relationship between the processing time and the number of queries. We repeat the process of filtering 38,000 documents against 1, 1,000, 4,000, 7,000, and 14,000 profiles. The last run is done simply by duplicating the 7,000 profiles in the database. (At that time we did not have 14,000 profiles.) Figure 11 shows the results. Again, the query index build time is negligible. For the filtering time, we find that even for one query, it takes 5,105 sec. to filter. This is the time spent reading in all the articles via NFS. The filter time itself is apparently linear in the number of queries, with slope 0.62

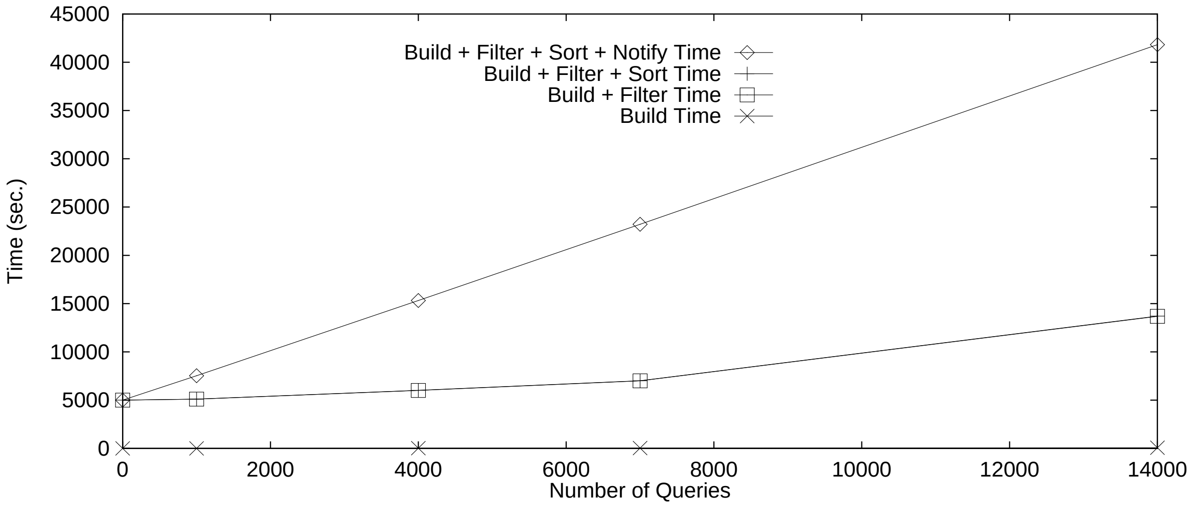

Figure 11: SIFT Performance vs. Number of Queries

sec./query. The notify time is obtained as discussed above. We find the total running time to be linear with respect to the number of queries, and the slope is 2.63 sec./query.

4.3.1 Performance Improvement

To quantify the performance improvement obtained by using the filtering engine, we compared SIFT Version 1 with the earlier SIFT Version 0. Version 0 constructed a document index for a day's worth of documents, and then simply ran each query against it. It did not use a query index, so roughly speaking, it was an implementation of the BF method of Section 4. This process consists of these three steps:

1. *Build Time* – The document index structure is built.
2. *Filter Time* – Queries are run against the document index one by one. Document-query matchings are written into a file.
3. *Notify Time* – Update messages are sent out.

Comparing this with the filtering engine process (Version 1), we note that although the first steps have similar names, they represent totally different work. The document index takes much longer to build since there are many more documents than queries and a document is much larger than a query. Also the sorting of matchings is not needed in this approach because the filtering is done on a per profile basis. In fact, a

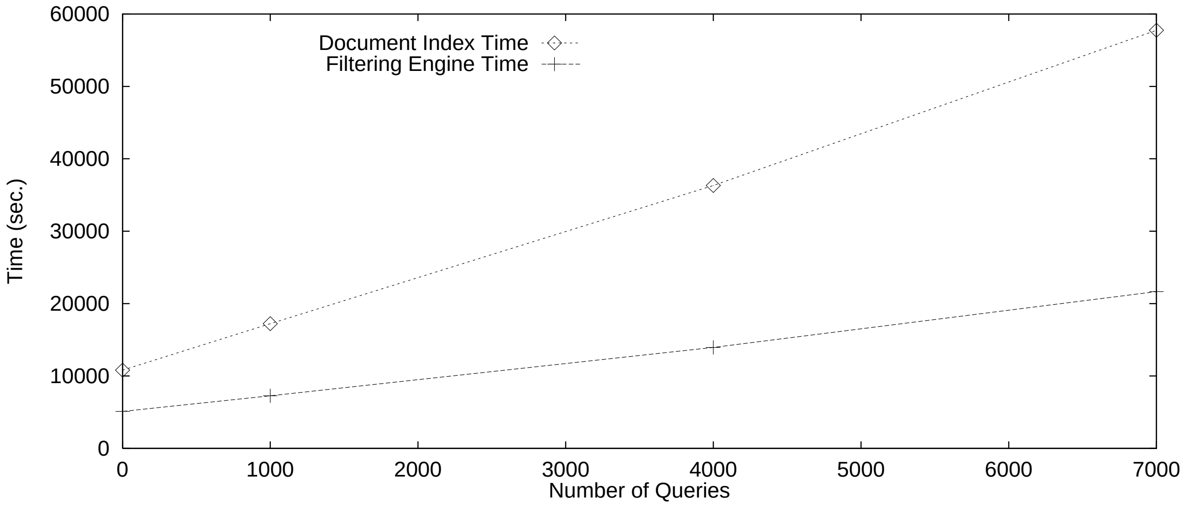

Figure 12: Comparing Performances of Document Index and Filtering Engine

notification may be sent out as soon as a profile is processed. However, for comparison purposes we opt to separate the work into the shown sequence.

First we compare the total running times of the two approaches. Figure 12 shows the results of processing different numbers of queries against 38,000 documents. The top curve is for the document index strategy (Version 0), while the lower one is for the filtering engine strategy (Version 1). The slope for the document index total running time is 6.68 sec./query, compared with 2.63 sec./query for the filtering engine. If we focus on the 7,000 query runs, the total running time with the document index is 57,745 sec., compared with 21,652 sec. with the filtering engine. Thus, the SIFT filtering engine is more than twice as fast as the document index approach.

Figure 13 shows the time breakdown for the document index approach. We see that the document index build stage makes up a significant portion (10,803 sec.) of the whole process. The majority of the time is spent in the second stage; for the 7,000 query run, the filter time is 33,345 sec. or 58% of the whole processing time. On the other hand, the notify time now takes up a smaller fraction of the running time.

It may be argued that the document index build time should also be included in computing the total running time for the filtering engine case, since it would be built anyway to provide the test run facility. However, if the document index is used strictly for test run purposes, we might simply build an index with any representative collection and not once for every batch. Even if we do include the document index build time for the filter engine strategy, the total time for 7,000 queries sums up to 32,455 sec., still just 56% of that for the document index approach.

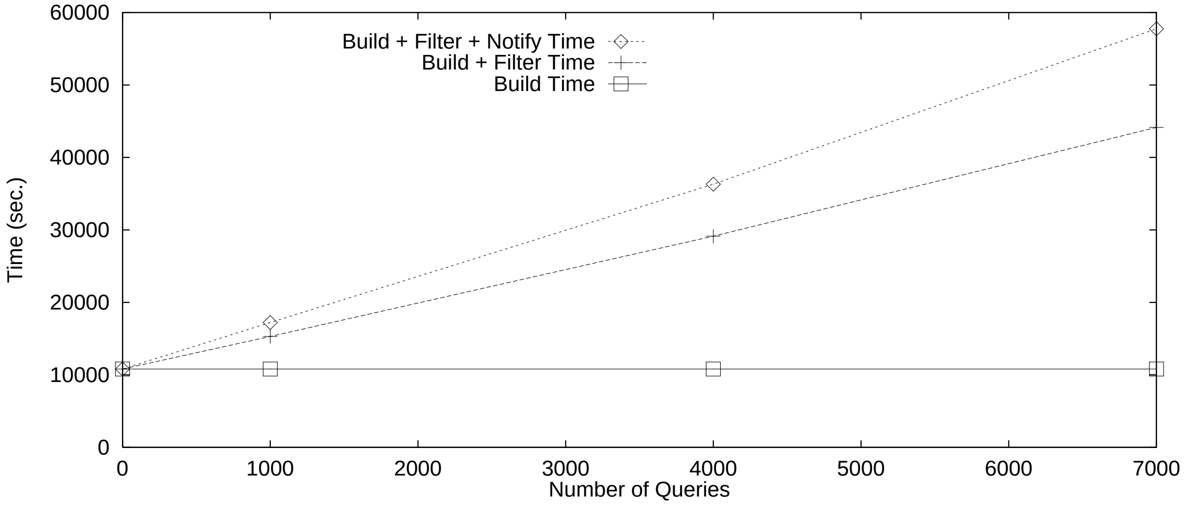

Figure 13: Breakdown of Running Time Using Document Index

In summary, the performance evaluation showed that some of the performance gains predicted by our analysis (Section 4.2) were achieved by implementing a query index. However, the results also showed that, having significantly reduced the time for matching queries and documents, the new system bottleneck was the time to mail out notifications. This lead us to improve SIFT Version 1 in a number of ways, leading the SIFT Version 2 (the last version before commercialization). The most important change was in the alerter component. In Version 2, instead of invoking Unix sendmail and thus creating a new process for every update, we send the outgoing message directly to a mail daemon, which then sends out the message. The significant overhead incurred by invoking sendmail is avoided. Further, the mail daemon is located on a separate machine, thus relieving the SIFT machine of the task of actually sending out the messages. This way, we were able to cut down the notification time significantly.

5 Distributed Information Dissemination

As the query and document loads increase, or as more reliable operation in the face of failures is needed, a service such as SIFT must be distributed. In this section we describe how SIFT or similar services could be distributed in the future, and we evaluate some of the options.

In a distributed environment, there will be multiple dissemination *servers*. A document will be examined by one or more servers, and a profile (or standing query) may be *posted* (i.e., submitted for continuous matching) at more than one server. We call the set of servers that a document is sent to a *document quorum*, and the set of servers that a profile is posted at a *profile quorum*. Given a profile quorum P and an document quorum D, one can guarantee that a profile does not miss an document by having $P \cap D \neq \emptyset$. We call this the *intersection property.* This property is similar to that required for replicated data [AA90].

One way to enforce the intersection property is with majority consensus (related to [Tho79]). Suppose we have n servers, and a document is sent to a document quorum formed by d (arbitrary) servers. A profile is posted at a profile quorum formed by p (arbitrary) servers. If we enforce the equality $p + d = n + 1$, then the intersection property is guaranteed. Note that to update a profile (including submission, modification/feedback, and deletion), p servers have to be accessed.

As an example, suppose we have eight servers with $p = 3$ and $d = 6$. Each profile is posted at any three arbitrary servers, while each document is sent to any six servers. In this case, every matching document-profile pair will be found, but a document may match the same profile at up to three servers, possibly causing duplicates to be delivered to the end user. (Duplicate documents are screened out at the user site; see [YGM95a] for a discussion.) We denote a majority consensus arrangement as M(p, d).

An alternative is to organize the servers into a grid of d rows, each having p servers (related to protocols such as [Mae85, CAA90]). A profile quorum is formed by selecting a row at random (p servers), and a document quorum is formed by selecting a (random) representative from every row (d servers). For instance, in a 2×2 grid of servers $s_{1,1}$, $s_{1,2}$, $s_{2,1}$, $s_{2,2}$, a profile is posted at either $s_{1,1}$, $s_{1,2}$, or at $s_{2,1}$, $s_{2,2}$, A new document is sent to one of $s_{1,1}$, $s_{1,2}$, and to one of $s_{2,1}$, $s_{2,2}$. We denote such an arrangement as G(p, d). With the grid protocol, there is always only one server in the intersection between any document quorum and any profile quorum. It is this server that performs the matching for a particular profile-document pair.

Another alternative is a hierarchical organization (related to protocols such as [AA90, RST92, Kum91]). To illustrate, consider 8 servers. Suppose we first split the servers into two logical units, each with four servers. Within each unit, we choose to arrange the servers into a 2×2 grid. Next we view the two units as two logical servers, and coordinate them using the majority consensus protocol. To post a profile, we select one of the units. Within that unit, we then post the profile at the servers in one of the rows. A document is sent to both units, and a server from each row is selected to receive the document. We call this scheme GM(2, 2, 1, 2) because we use a G(2, 2) protocol within units, and M(1, 2) to select units. If we focus on two-level hierarchies, we can also have MM, MG, and GG combinations. Note that we can still get duplicate document deliveries to the end user with the grid and hierarchical organizations.

5.1 Performance Evaluation

To contrast these options, we describe a simple performance model. Our results and observations are given in the following section. (For additional details, please refer to [Yan95].)

We assume that the combined document generation rate from all information sources follows a Poisson distribution, with an average of λ document/sec. The document size follows an exponential distribution with mean k_{doc} words. The query update (including submission, modification/feedback, and deletion) rate also follows a Poisson distribution, with its mean proportional (by ν) to the number of queries. We assume the size of a update message is exponentially distributed, with mean k_{update} bytes. Other model parameters are given in Table 3.

Each information dissemination server is modeled as an $M/G/1$ server, servicing two kinds of jobs: documents to be filtered and query updates. The time it takes to match an incoming document is proportional (by c_{filter} sec./query/word) to the number of queries at the server and the size of the document. Processing a query update takes time exponentially distributed with mean t_{update} sec. The availability of a server, i.e., the probability that it is operational, is a_{server}.

For the WAN network, instead of modeling a particular existing network, we opt for a simple, generic topology in which a number of switching nodes are fully connected by n_{link} links (note that if x is the number of switching nodes, $n_{link} = x(x-1)$), each of bandwidth b Kbps. This model captures communication parallelism, and is simple enough for us to derive closed-form solutions. Again for simplicity, we assume that the users, information sources, and information dissemination servers are uniformly distributed across the network, and thus the amount of traffic through each channel is the same. Each channel is modeled as an $M/G/1$ server, processing new documents and query update messages. As we assume that the sizes of documents and update messages follow exponential distributions, the respective transmission times also follow exponential distributions.

Table 3 shows a summary of the parameters in our evaluation model, together with the base values, which represent a reasonable starting point for our evaluation. For some parameters, we make use of statistics collected from our Netnews information dissemination server to estimate the base values. For the rest, we choose values that we believe are reasonable and are useful for illustrating the key tradeoffs. One notable parameter is a_{server}, which is the availability of a SIFT server. This includes hardware and software failures, as well as the reachability of the server from other hosts on the network. Reference [LCP91] reports that the availabilities of some 68,000 Internet hosts range from 0.7833 to 0.9688. We assume a value of 0.8.

5.2 Results

The fundamental tradeoff for distributed dissemination is performance versus availability. Figure 14 illustrates this tradeoff for some of the distribution schemes we have described. The vertical axis gives the document delivery delay, i.e., the total time elapsed between the time a document is generated and the time the interested user receives the document. The horizontal axis gives the availability. For this we define system failure as the event that a user is missing documents, i.e., not receiving documents that match his or her queries as soon as possible. System availability is then defined as the probability that there is no failure

Parameter	Base Value	Description
λ	0.4823	mean document generation rate (document/sec.)
k_{doc}	323	mean document size (words)
n_{query}	1,000,000	total # queries
ν	6.918×10^{-7}	proportional constant for mean query update rate (update/sec./query)
s_{update}	50	mean update message size (bytes)
n	16	# information dissemination servers
p	1 - 16	a query is replicated at p servers
d	1 - 16	a document is sent to d servers
φ	0.000294	probability that a query matches a document
c_{filter}	10^{-8}	proportionality constant for mean filtering time (sec./query/word)
t_{update}	10^{-2}	query update processing time (sec.)
a_{server}	0.8	server availability
n_{link}	72	# links in WAN model
$band$	50	bandwidth of WAN link (in Kbps)

Table 3: Model Parameters for Performance Evaluation

at a particular time. Each data point represents a particular quorum organization. For example, if we look at the graph for the grid organizations (diamond symbol), the leftmost data point is for G(1, 16), followed by those for G(2, 8), G(4, 4), G(8, 2), and G(16, 1).

The best organization would be the one with the highest availability and the shortest delivery delay; i.e., as close as to the bottom-right corner of the figure as possible. There are two candidates: G(2, 8) and G(4, 4), and the latter is apparently better as it provides a high availability. Using the grid, if we tolerate lower availability, we achieve shorter delivery delay; and if we desire higher availability, we have to tolerate longer delays. In fact, we can see that the grid organizations form a delay vs. availability envelope: no organization gives superior tradeoff. And in general, grid organizations with balanced document and query quorum sizes provide high system availabilities and short delivery delays. Other results obtained confirm this general conclusion that the grid organization, with balanced quorum sizes, is the best for distributed dissemination.

Figure 15 shows the tradeoff between system availability (x-axis) and network utilization (y-axis). Again, the best organization should be the one with the highest availability and the least utilization; i.e., as close as to the bottom-right corner of each figure as possible. We can see that again the grid organizations envelop the others. In general, the grid organizations use the network most efficiently (its data points are at the bottom of the figure), while the majority quorum organizations are the least efficient (often at 100% network

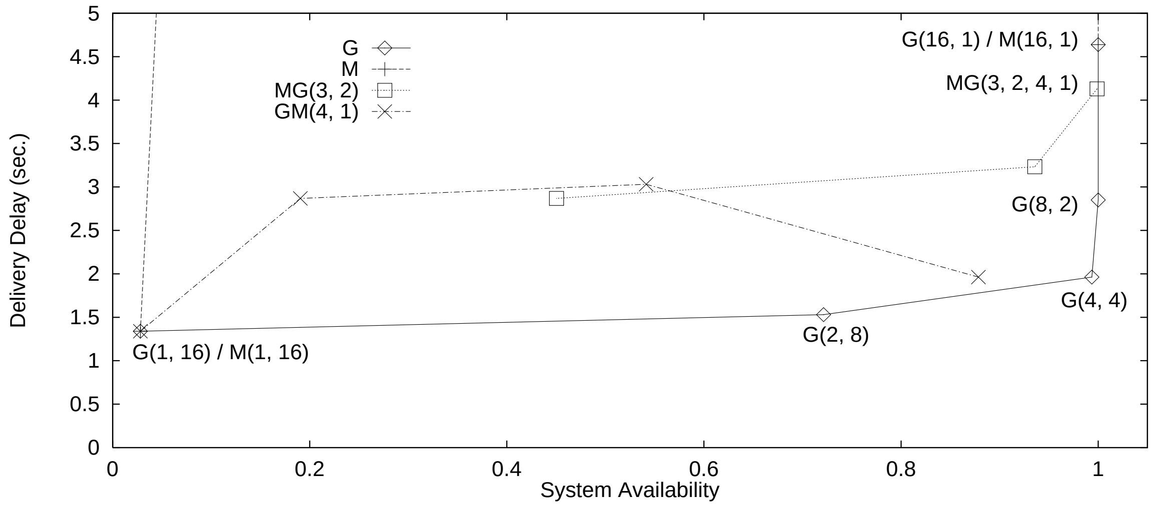

Figure 14: Document Delivery Delay vs. System Availability

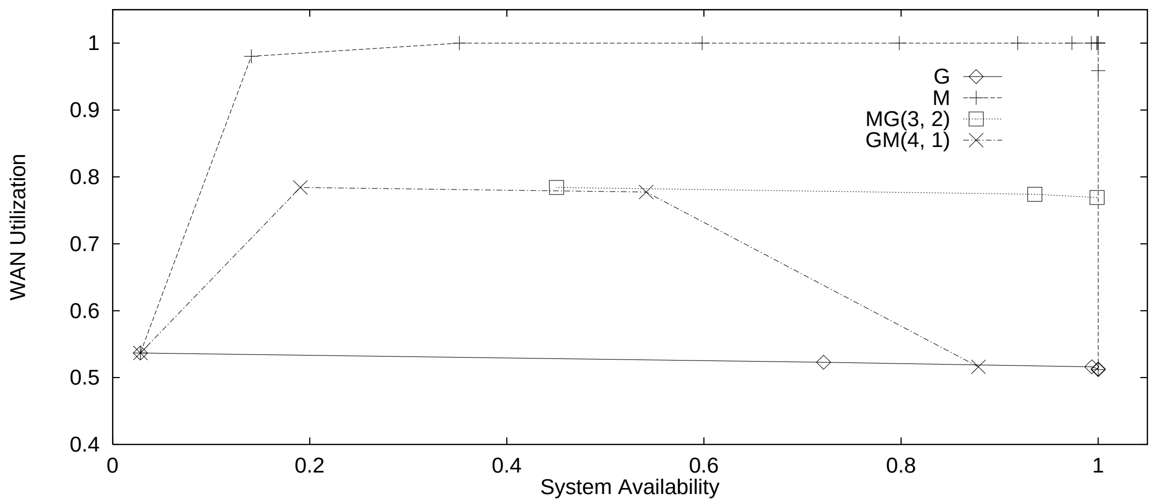

Figure 15: WAN Utilization vs. System Availability

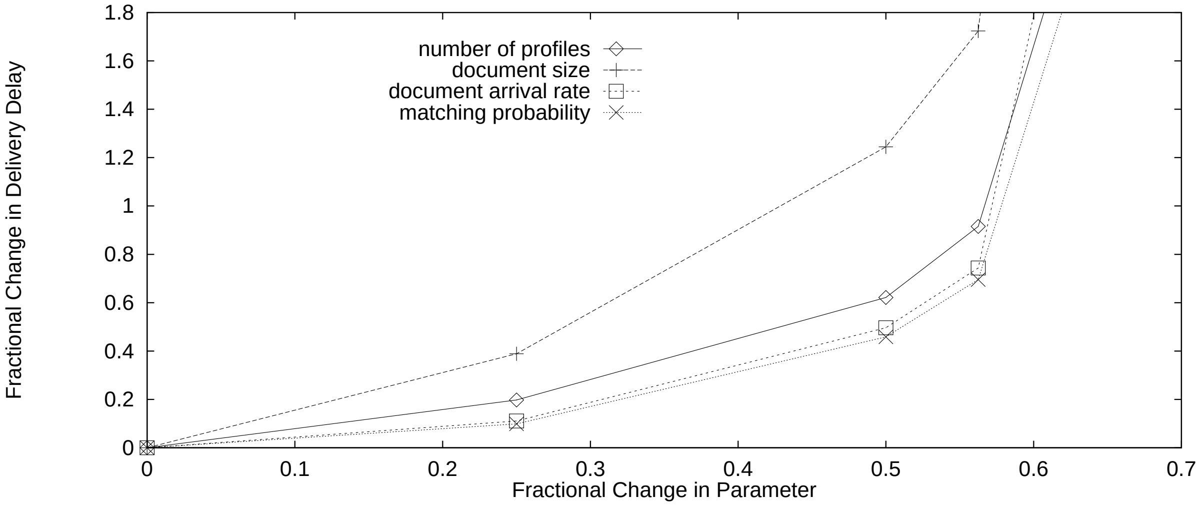

Figure 16: Sensitivity Analysis: Adding Information Dissemination Servers

utilization). Notice that in this network evaluation we assume that servers send documents to users in an individual fashion, like SIFT does. Regardless of the server organization, one can try to reduce network costs by propagating matching documents from servers to users more efficiently. For instance, if a set of users on a cluster gets common documents, the server can send the documents to some cluster controller, which then disseminates locally. (This is analogous to how Netnews articles are disseminated.) Reference [YGM94a] discusses this further.

Looking to the future, it is natural to wonder how distributed dissemination will perform as loads continue to grow. For instance, in the near future, we expect very large, e.g., multimedia, documents. If we have faster networks, does this "solve" the problem? That is, can we scale to "larger" scenarios just by adding bandwidth? Or do we also need more information dissemination servers? To answer these questions, we carry out the following experiments. In each experiment, we try a different strategy to control the growth in the delivery delay as the parameters are scaled up.

Adding Information Dissemination Servers. In our first experiment, Figure 16, we study if adding dissemination servers can handle growth. To do this, we take a critical growth parameter (e.g., document size) and increase it, while maintaining a constant ratio between the total number of servers and the studied parameter. For example, for the second data point (fractional change = 0.25) in the graph for document size ("+" symbol) in Figure 16, we assume we have 20 servers and use the G(4, 5) configuration. The fractional increase in the number of servers is (20-16)/16 = 25%, and thus we increase the document size by 25% also. The rest of the data points in that graph correspond to G(4, 6) and G(5, 5) in that order. Now, if this curve were horizontal, this would mean that we could maintain the same performance (no increase in

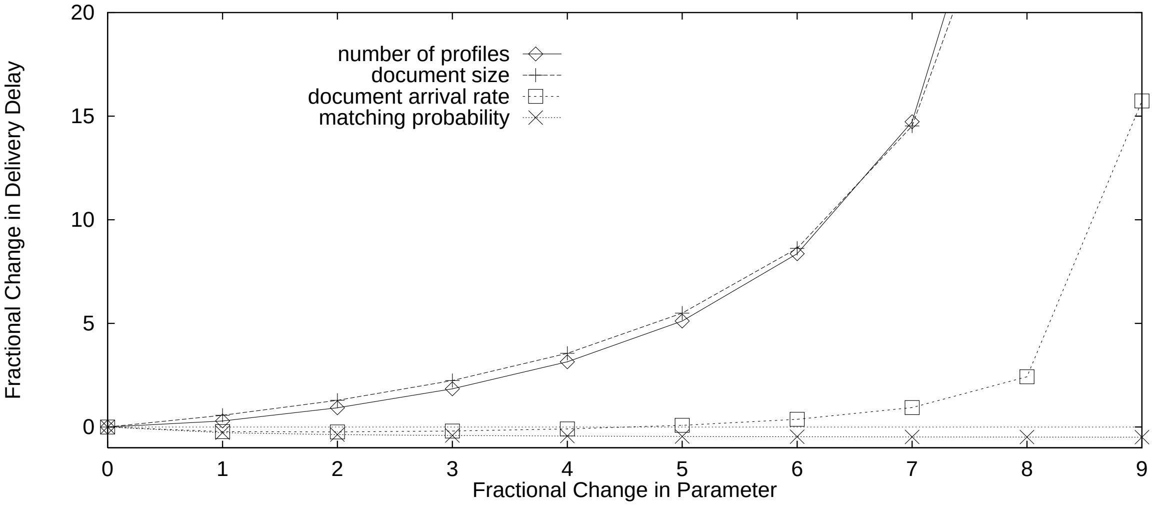

Figure 17: Sensitivity Analysis: Increasing WAN Bandwidth

delivery delay) by simply adding information dissemination servers as the documents grew in size. However, the results show adding servers hardly helps at all — when we increase the document size and the number of servers proportionally, the system is bottlenecked at the WAN (the other critical system resource), and performance degrades dramatically. The same is true for the other three parameters. This confirms our intuition that dissemination is a "naturally centralized" problem and is hard to partition.

Increasing Network Bandwidth. Next we examine if we can cope with the scaling-up of the parameters by increasing the network bandwidth alone. While increasing each studied parameter, we increase the network bandwidth to keep the ratio (parameter value)/(network bandwidth) constant. For instance, for the last data point (fractional change = 9) for document arrival rate (box symbol) in Figure 17, we increase the number of queries and network bandwidth ten times simultaneously. The results show that increasing bandwidth effectively controls the increase in query-document matching probability, and to a large extent the document arrival rate. However, when the number of queries or document size is increased, or when the document arrival rate is very high (e.g., eight times the base value), the information dissemination servers become the bottleneck of the system. In an experiment not shown here, we also observed that even if we increase bandwidth *and* the number of servers simultaneously, it is still hard to cope with large documents and with many profiles.

Increasing Network Bandwidth and Reducing Filtering Proportionality Constant. Finally, we repeat our procedure, but this time we proportionally increase the WAN bandwidth and *decrease* the proportionality constant for the mean filtering time. The latter controls the time it takes for a information dissemination

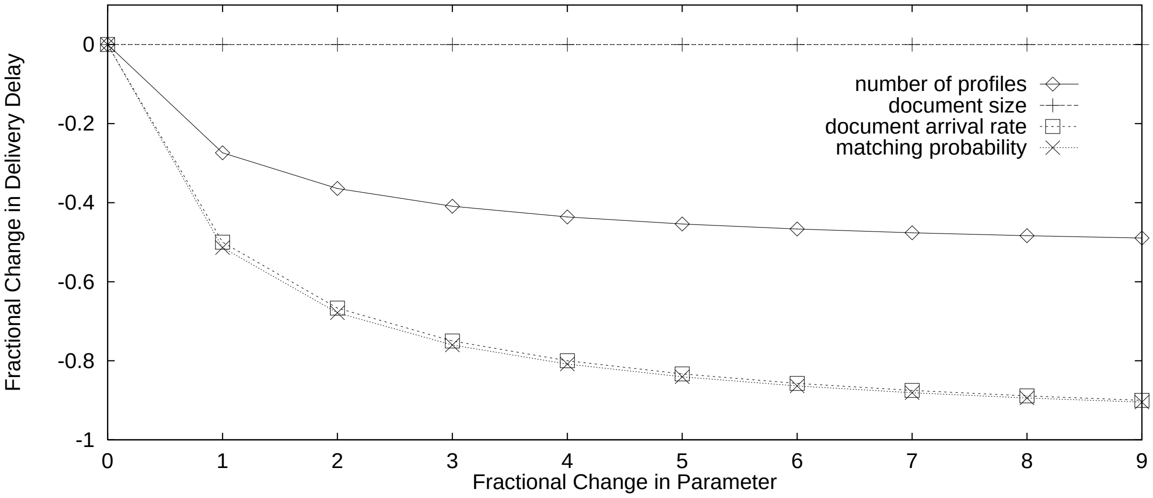

Figure 18: Sensitivity Analysis: Increasing WAN Bandwidth and Decreasing Filtering Constant

server to filter a document. We find that performance improves for *all* studied parameters (Figure 18), indicating that fast filtering at a information dissemination server is very effective, more so than adding servers, in coping with large scale scenarios. More sophisticated query indexing schemes, such as those proposed in Sections 4, are attractive. Hardware solutions (e.g., using parallel computers and RAIDs) to reduce the processing time may also be desirable.

5.3 Distributed SIFT

To validate the model and predictions of the previous section, we implemented a prototype distributed version of SIFT, using the Grid protocol. As it is difficult to coordinate an experiment involving machines on an wide-area network, in our experiment we used machines located in the same local-area network. Thus the results shed light on how the grid protocol behaves in a local-area network. We used four DEC 3000 (Alpha) machines as four SIFT servers. At the time of this experiment we did not have enough SIFT queries to justify distribution, so we used 137,037 queries from the Carnegie-Mellon University Lycos system. (These queries are not long-term profile queries, but are retrospective queries issued against an existing catalog of web pages. In our experience, user profiles have similar characteristics to one-time queries.)

We experimented with the configurations G(1,4), G(2,2), and G(4,1). For example, in the G(1,4) configuration, we partitioned the user queries into four groups, and loaded each SIFT server with one group. We then ran a number of USENET News articles against this configuration, sending each article to all four servers. The average time needed to match an article against the queries was then measured. We did not

Organization	Actual Processing Time (sec.)	Predicted Processing Time (sec.)
G(1,4)	0.031	0.019
G(2,2)	0.022	0.015
G(4,1)	0.120	0.024
Single Site	0.118	N/A

Table 4: Experimental Distribution Experiment

measure the time required to send out the notifications here, since for all configurations we need the same amount of processing to sort the matchings and send out email messages or prepare personal web pages. Table 4 shows the results in column 2.

Next we compared the actual results with performance numbers predicted by the analytical model. We used the following measured values for the parameters: $\lambda = 55$, $k_{doc} = 358$, $n_{query} = 137{,}037$, and $band = 4 \times 2^{10}$. As all machines are on one Ethernet LAN, we assumed $n_{link} = 1$. And using the results above for a single site, we calibrated c_{filter} as 3×10^{-10}. (Note that there were no update requests, so certain parameters in our model are irrelevant here.)

Feeding these numbers into our model, we computed the processing times for the grid organizations, and the results are shown in the third column of Table 4. The actual number for each configuration do not match well, but this is to be expected since our simple model was not intended to predict actual performances. However, what is important to notice is that the *relative* performance of the schemes does match relatively well, with G(2,2) being the best, and G(1,4) the next best. Thus, although our comparison between the model and the implemented distributed SIFT is limited, it does give us some confidence that the models we used in our design are useful for rough comparisons among strategies.

As pointed out earlier, distributed dissemination can deliver the same document to the end user multiple times. One way to cope with this problem is to have duplicate removal at the user site, although this still causes unnecessary network traffic. If the distribution is done on a local network, as in our experiment (say to improve performance), another option is to have an additional processing stage for duplicate removal. To illustrate, say we have a set of computers $C_1, C_2, ..., C_N$ for this processing. (They need not be separate from the servers doing the matching.) After a server discovers a matching for end user X, it hashes the identity of X to one of the N computers, and forwards the matching there. Thus, all the matchings for a given user are processes at the same C_i, and duplicates can be removed before a notification is mailed to a user. This duplicate elimination is not included in the results of Table 4. For an additional discussion of duplicate elimination, see [YGM95a].

6 Conclusions

Large-scale information dissemination is becoming increasingly important. In addition to existing services available from commercial information providers such as Dialog and traditional libraries, new services and systems have emerged on the Internet, including SIFT (InReference), InfoSeek Personal Newswire, and Mercury NewsHound. Many sources of information are covered, such as news wires, USENET News, and mailing lists. With the rapid growth of the Internet, these systems will become more and more popular, and the volume of traffic will be higher and higher.

In this paper we have studied two of the fundamental problems faced by dissemination systems: how to index queries and how to distribute the load among multiple servers. We presented a variety of indexing and distribution techniques, and through analysis showed that the right choice can significantly improve performance and availability. We also discussed how some of these techniques were implemented in SIFT, and we reported on the performance of the actual system. We believe that our analytical results and our SIFT experience can be of use to future designers.

SIFT itself is popular information dissemination system, providing a valuable service to the Internet. In addition to the SIFT service now operated by InReference, the SIFT code is being used at other sites for dissemination services. These include the EBI BioSci Filtering Service set up at the European Bioinformatics Institute to disseminate bioinformatics information, and another SIFT server set up at Hewlett-Packard Laboratories as part of the InfoServer project.

References

[AA90] D. Agrawal and A. El Abbadi. Exploiting logical structures in replicated databases. *Information Processing Letters*, 33:255–60, 1990.

[AFZ96] S. Acharya, M. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. In *Proc. Very Large Data Bases*, pages 354–365, 1996.

[BL85] C. Buckley and A. Lewit. Optimization of inverted vector searches. In *Proc. ACM SIGIR Conference*, 1985.

[Bro95] E. Brown. Fast evaluation of structured queries for information retrieval. In *Proc. ACM SIGIR Conference*, pages 30–38, 1995.

[BS95] M. Balabanovic and Y. Shoham. Learning information retrieval agents: experiments with automated web browsing. In *Proceedings of the AAAI-95 Spring Symposium on Information Gathering from Heterogenous, Distributed Environments*, 1995.

[CAA90] S.Y. Cheung, M.H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. In *Proc. International Conference on Data Engineering*, pages 438–45, 1990.

[Coh92] D. Cohen. *A Format for E-mailing Bibliographical Records (RFC-1357).* Network Information Center, SRI International, Menlo Park, California, 1992.

[DP96] S. Dao and B. Perry. Information dissemination in hybrid satellite/terrestrial networks. *Data Engineering Bulletin*, 19(3):12–19, 1996.

[Edi96] M. Franklin (Special Issue Editor). Special issue on data dissemination. *Data Engineering Bulletin*, 19(3):3–54, 1996.

[FD92] P.W. Foltz and S.T. Dumais. Personalized information delivery: an analysis of information filtering methods. *Communication of the ACM*, 35(12):29–38, 1992.

[FIS89] S.H. Friedberg, A.J. Insel, and L.E. Spence. *Linear Algebra.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[FZ96] M. Franklin and S. Zdonik. Dissemination-based information systems. *Data Engineering Bulletin*, 19(3):20–30, 1996.

[GBBL85] D. Gifford, R. Baldwin, S. Berlin, and J. Lucassen. An architecture for large scale information systems. In *Proc. Symposium on Operating System Principles*, pages 161–70, 1985.

[Gla96] D. Glance. Multicast support for data dissemination in orbixtalk. *Data Engineering Bulletin*, 19(3):31–39, 1996.

[GNOT92] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communication of the ACM*, 35(12):61–70, 1992.

[Har93] D. Harmon. *Proceedings of the 1st Text Retrieval Conference.* National Institute of Standards, Maryland, 1993.

[Har94] D. Harmon. *Proceedings of the 2nd Text Retrieval Conference.* National Institute of Standards, Maryland, 1994.

[Har95] D. Harmon. *Proceedings of the 3rd Text Retrieval Conference.* National Institute of Standards, Maryland, 1995.

[IVB94] T. Imielinski, S. Viswanathan, and B. Badrinath. Power efficient filtering of data an air. In *Proc. Extending Data Base Technology*, pages 245–258, 1994.

[Kro92] E. Krol. *The Whole Internet User's Guide & Catalog.* O'Reilly & Associates, Sebastopol, California, 1992.

[Kum91] A. Kumar. Hierarchical quorum consensus: a new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, 1991.

[LCP91] D.D.E. Long, J.L. Carroll, and C.J. Park. A study of the reliability of internet sites. In *Proc. IEEE Symposium on Reliable Distributed Systems*, pages 177–86, 1991.

[Mae85] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–59, 1985.

[Mal87] T.W. Malone. Intelligent information sharing systems. *Communications of the ACM*, 30(5):390–402, 1987.

[Mar97] Marimba. *http://www.marimba.com.* Marimba, Inc., Palo Alto, California, 1997.

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus – an architecture for extensible distributed systems. *Operating Systems Review*, 27(5):58–68, 1993.

[Per94] M. Persin. Document filtering for fast ranking. In *Proc. ACM SIGIR Conference*, pages 339–348, 1994.

[Poi97] Pointcast. *http://www.pointcast.com.* Pointcast, Inc., Sunnyvale, California, 1997.

[Por80] M.F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–7, 1980.

[RST92] S. Rangarajan, S. Setia, and S.K. Tripathi. Fault tolerant algorithms for replicated data management. In *Proc. International Conference on Data Engineering*, pages 230–7, 1992.

[Sal68] G. Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, New York, 1968.

[Sal91] G. Salton. Global text matching for information retrieval. *Science*, 253:1012–15, 1991.

[SFL96] S. Shekhar, A. Fetterer, and D. Liu. Genesis: An approach to data dissemination in advanced traveler information systems. *Data Engineering Bulletin*, 19(3):41–47, 1996.

[She94] B.D. Sheth. A learning approach to personalized information filtering. Technical Report Master Thesis, Massachusetts Institute of Technology, 1994.

[Ste93] C. Stevens. Knowledge-based assistance for handling large, poorly structured information spaces. Technical Report Ph.D. Thesis, CU-CS-640-93, University of Colorado at Boulder, 1993.

[Ter92] D. Terry. Replication in an information filtering system. In *Proc. 2nd Workshop on the Management of Replicated Data*, pages 66–7, 1992.

[TGM93] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

[Tho79] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[WF91] M.F. Wyle and H.P. Frei. Retrieval algorithm effectiveness in a wide area network information filter. In *Proc. ACM SIGIR Conference*, pages 114–22, 1991.

[Wol91] S. Wolfram. *Mathematica*. Addison Wesley, Redwood City, California, 1991.

[Yan95] T. Yan. *Efficient Techniques for Wide-Area Information Dissemination. Ph.D. Thesis.* Department of Computer Science, Stanford University, 1995.

[YGM93] T.W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. Technical Report STAN-CS-92-1454, Stanford University, 1993.

[YGM94a] T.W. Yan and H. Garcia-Molina. Distributed selective dissemination of information. In *Proc. Parallel and Distributed Information Systems*, pages 89–98, 1994.

[YGM94b] T.W. Yan and H. Garcia-Molina. Index structures for information filtering under the vector space model. In *Proc. International Conference on Data Engineering*, pages 337–47, 1994.

[YGM94c] T.W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Transactions on Database Systems*, 19(2):332–64, 1994.

[YGM95a] T. Yan and H. Garcia-Molina. Duplicate removal in information dissemination. In *Proc. Very Large Data Base Conference*, 1995.

[YGM95b] T. Yan and H. Garcia-Molina. Sift – a tool for wide-area information dissemination. In *Proc. 1995 USENIX Technical Conference*, pages 177–86, 1995.

[Zip49] G.K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, Cambridge, Massachusetts, 1949.