

Efficiency Considerations for Scalable Information Retrieval Servers

Ophir Frieder *
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
ophir@cs.iit.edu

David A. Grossman
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
dagr@cs.iit.edu

Abdur Chowdhury
Department of Computer Science & IIT Research Institute
Illinois Institute of Technology
Chicago, IL 60616
abdur@cs.iit.edu

Gideon Frieder
Operations Research Dept.
George Washington University
Washington, DC 20052
frieder@seas.gwu.edu

Abstract

We review a variety of techniques to improve efficiency in information retrieval. Given the increasing volumes of data that are available electronically, understanding and using such techniques is critical.

We address several efficiency concerns, but our primary focus is on index processing since it dominates the computational demands of information retrieval. Given the importance of index processing, in addition to a general overview, we include some recent index maintenance results. These results demonstrate that by delaying the updating of the index when additional documents are introduced to the collection, efficiency is improved without noticeably degrading the effectiveness of information retrieval.

We conclude with an overview of parallel processing in information retrieval. Since users cannot tolerate lengthy response times, searching large text databases requires vast computational resources. Parallel processing is currently the only means to support these demands. We focus on only those approaches that are currently commercially viable.

1 Introduction

Information Retrieval (IR) refers to the processing of user requests, commonly referred to as queries, to obtain relevant information. This problem differs fundamentally from most computer science problems in that the nature of the result is not well defined. For example, the correctness of the results of a sorting algorithm, namely the sorted nature of a list, is easily judged. For an IR result set, however, it is human judgment that determines whether or not a given document is relevant to a given query. It has been shown that different people will frequently disagree on this judgment [31]. Hence, there is a need to define assessment

*This work supported in part by the National Science Foundation under contract number IRI-9357785.

measures for IR. Two commonly used measures are *effectiveness* and *efficiency*. Our primary focus in this paper is on efficiency.

Effectiveness measures the accuracy of the result set in terms of two measures: *precision* and *recall*. Precision is defined as the fraction of relevant documents retrieved to the total number retrieved. Recall is defined as the fraction of relevant documents retrieved to the total number of relevant documents collection-wide. Note that the universe of known relevant documents in a given query is very difficult to identify; hence, only a few document collections exist where recall has been estimated. These collections are commonly used for experimentation and evaluation.

Efficiency measures how fast a result is obtained. This may be computed using standard algorithmic complexity analysis (e.g., the “Big Oh” notation) or more empirically measured with statistics such as response time, disk I/O, etc.

Most research in IR has centered around effectiveness since it is well accepted that current retrieval systems exhibit mediocre accuracy. Moreover, users have become complacent in their expectation of accuracy of information retrieval systems [14]. Therefore, many information retrieval researchers focus on effectiveness and tend to ignore efficiency since, given poor effectiveness, efficiency is of secondary concern. For example, a recent compendium of seminal research papers [40] did not include a single paper whose primary focus was efficiency. The reality is, however, that commercial systems depend upon efficiency for their success. A system that takes too long to respond to a user request will simply not be used.

There are only a few basic means to improve efficiency. The first includes the preprocessing step of building an inverted index. An index is a structure that is common to all commercial IR systems that identifies, for each term, a list of documents that the term appears in. Various compression algorithms exist for inverted indexes, and we review some of them.

Another means of improving efficiency is in query processing. Some terms may be ignored, and other shortcuts may be taken based on the query. We overview some past solutions and note that a combination of such optimizations may yield greater efficiency than any technique individually. Typically, however, reduction in processing time comes at the expense of accuracy; hence, a close examination of the impact of using such techniques is necessary.

To process the ever-increasing volume of data while still providing acceptable response times, a few parallel processing algorithms specifically for IR were developed. Unfortunately, most of these approaches were machine-specific and did not achieve wide commercial adoption. One commercially available technique is the treatment of an IR system as an application of a relational database system. By using a relational database system hosted on a parallel platform, a scalable parallel IR algorithm is obtained without any special-purpose software [23]. Essentially, the database system does the parallelization, and the IR application merely takes advantage of this framework.

Relatively little research exists on distributed IR, but we focus on work done on data replication and tuning with regard to web search engines. This is relevant because every web search engine is an example of a distributed IR system.

The key algorithm most retrieval systems are implementing is called *relevance ranking*. The basic idea is to compute a measure of similarity between the query and each document. Towards implementing this basic idea, we describe various approaches that were and are being used. Initially, we focus on inverted index creation and storage. We continue with a discussion and some experimental results regarding inverted index updating frequencies. Having described inverted indexing, we turn our attention to query processing

where we describe several generic query processing algorithms that impact efficiency. Subsequently, we review several more recent parallel and distributed information retrieval approaches and conclude with our overall observations.

Note that this article focuses primarily on efficiency and does not address traditional approaches to improve accuracy. When accuracy improving utilities such as passages, proximity, relevance feedback, clustering, n-grams, etc. are used, some straight-forward modifications to the structures we describe are needed. Briefly, passage processing refers to indexing documents by sub-documents [4]. Proximity processing allows users to query on specific location within a document [37]. In doing so, a user can specify that the terms in the query must occur within some fixed window of the document. Relevance feedback refers to the process of adding terms to the query or adjusting the term weights based on documents that have been identified, either automatically or manually, as *relevant* to the user query [35]. Document clustering is used to automatically group documents into distinct clusters of related documents—once this is done a search may be directed at a particular cluster [38]. Finally, n-gram processing is used to search for fragments of terms—often useful in corrupted document collections [7].

2 Inverted Index

Typically, an IR system builds an inverted index to efficiently find terms in a document collection. An inverted index consists of two components, a list of distinct terms referred to as the *index* and a set of lists referred to as *posting lists*. Figure 1 illustrates an inverted index.

Consider a document collection in which document one contains four occurrences of *apple* and two occurrences of *pie*. Document two contains three occurrences of *apple*. The index contains the entries *apple* and *pie*. The posting list is simply a linked list that is associated with each of these terms. In our example, for simplicity, we illustrate the posting entry with only the document number and term frequency. However, the structure of a posting list entry does vary from implementation to implementation. It always includes the document number but can also include entries for term frequency, term weights, and possibly position data. Thus, we have—

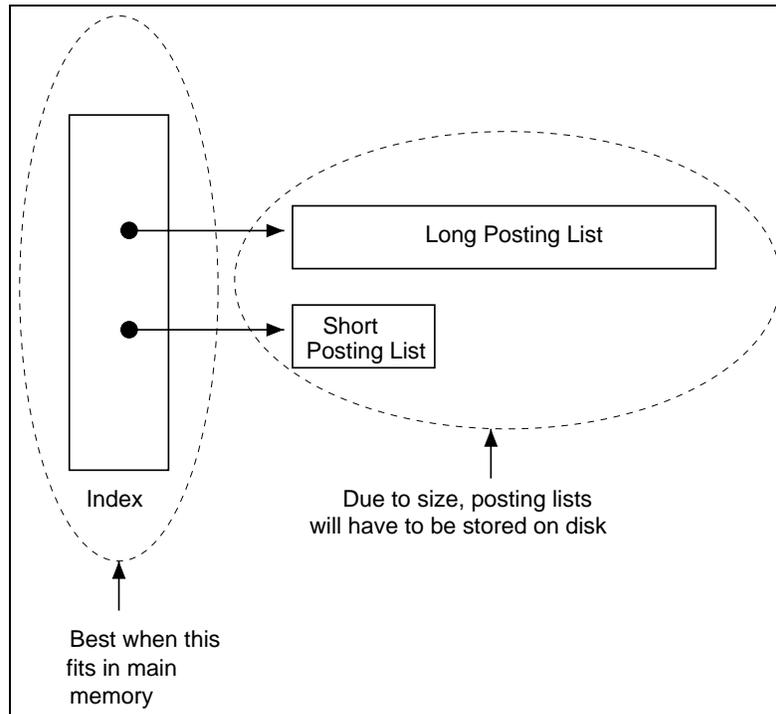
$$\begin{aligned}apple &\rightarrow (2,3) (1,4) \\pie &\rightarrow (1,2)\end{aligned}$$

The entries in the posting lists are stored in descending order by document number since it is typically more efficient to insert at the head of the list than in any other location. Clearly, the construction of this inverted index is expensive, but once built, queries can be efficiently executed.

As processing continues, the current posting list is stored in memory. Memory is allocated dynamically for each new posting list entry. With each memory allocation, a check is made to determine if the memory reserved for indexing has been exceeded. If it has, processing halts while all posting lists resident in memory are written to disk. Once processing continues, new posting lists are written. With each output to disk, posting list entries for the same term are chained together.

Index processing terminates when all of the terms have been processed. At this point, the inverse document frequency (*idf*) for each term is computed by scanning the entire list of unique terms. The inverse document frequencies measure the uniqueness of the terms in the collection. The higher the value, the more

Figure 1: Inverted Index



unique is the term. Once the *idf* is computed, it is possible to compute a weight for each document. This weight is used to normalize for the size of a document—the idea is that a large document should not be ranked either too high or too low simply because of the number of terms that appear in the document.

This is done by scanning the entire posting list for each term.

An entry in the list of documents may also contain the location of the term in the document (e.g., word, sentence, paragraph) to facilitate proximity search. A search that includes proximity involves locating a set of terms that are found within a text window of a given size in the document. Additionally, an entry can contain a manually or automatically assigned weight for the term in the document. This weight is often based on term frequency and is used in computations that generate a measure of relevance of the document to the query. Once this measure is computed, the retrieval algorithm identifies all the documents that are “relevant” to the query by sorting the measure of relevance called a *similarity coefficient* or *retrieval status value* (RSV) and presenting a ranked list to the user.

A similarity coefficient is a measure that estimates the relevance of a document to a given query. Many techniques exist to compute similarity coefficients (e.g., vector space model, probabilistic model, extended Boolean model, genetic algorithms, neural networks). For a comprehensive survey of these techniques, see [17].

Indexing requires additional overhead since the entire collection is scanned and substantial I/O is required to generate an efficiently represented inverted index for use in secondary storage. However, the use

Table 1: Sample Zipf Distribution

rank	frequency	constant
1	1.00	1
2	0.50	1
3	0.33	1
4	0.25	1
5	0.20	1

of indexing has been shown to dramatically reduce the amount of I/O required to satisfy an ad hoc query [45]. Upon receiving a query, the index is consulted, the corresponding posting lists are retrieved, and the documents are ranked based on the contents of the posting lists.

The size of the index is another concern. Many indexes equal or even exceed the size of the original text. This means that storage requirements are doubled due to the index. However, compression of the index typically results in a space requirement on the order of ten percent of the original text [48]. The terms or phrases stored in the index depend on the parsing algorithms that are employed.

The size of posting lists in the inverted index can be approximated by the Zipfian distribution—Zipf proposed that the term frequency distribution in a natural language is such that if all terms were ordered and assigned a rank, the product of their frequency and their rank would be constant [50]. In Table 1, we illustrate the Zipfian distribution when this constant equals one. Using $F = \frac{C}{r}$, where r is the rank, F is the frequency of occurrence, and C is the value of the constant, an estimate can be made for the number of occurrences of a given term. The constant, C , is domain-specific and equals the number of occurrences of the most frequent term.

2.1 Compressing an Inverted Index

A key objective in the development of inverted index files is to develop algorithms that reduce I/O and storage overhead. The size of the index file determines the storage overhead imposed. Since large index files demand greater I/O to read them, the size also directly affects the processing times. To reduce the size of the index file, compression techniques are used. Two primary areas in which an inverted index might be compressed are compression of the index and compression of the posting lists. We focus on posting list compression since the size of the posting list dominates the total size of the inverted index.

The King James Bible (about five megabytes) contains 9,020 distinct terms and the Text REtrieval Conference (TREC) collection (slightly over two gigabytes) contains 538,244 distinct terms [48]. The number of new terms always slightly increases as new domains are encountered, but it is reasonable to expect that it will stabilize on the order of millions of terms. With an average term length of six, a four byte document frequency counter, and a four byte pointer to the first entry in the posting list, fourteen bytes are required for each term. For a conservative estimate of eight million terms, the uncompressed index is likely to fit comfortably within 128 MB.

Given the relatively small size of an index and the ease with which it fits in memory, we do not describe a detailed discussion of techniques used to compress the index. We note that stemming and stop

Table 2: Uncompressed Encoding Example

value	uncompressed bit string
250	00000000 00000000 00000000 11111010
70	00000000 00000000 00000000 01000110
7	00000000 00000000 00000000 00000111
3	00000000 00000000 00000000 00000011
1	00000000 00000000 00000000 00000001

word processing reduce the storage size requirement, and Huffman encoding can be used in a relatively straightforward fashion to further compress the index [48].

Consider an entry for an arbitrary term, t_1 , that indicates t_1 occurs in documents 1, 3, 7, 70, and 250.

$$t_1 \rightarrow 250, 70, 7, 3, 1$$

Using no compression, the five entries in the posting list require four bytes each (this is only to store the document identifier) for a total of twenty bytes. Thus, in this example, the uncompressed posting list requires 160 bits of storage. (See Table 2).

To overview index compression algorithms, we first describe a relatively straightforward one that is referred to as Byte Aligned (BA) index compression [16]. BA compression is done within byte boundaries to improve run-time at a slight cost to the compression ratio. This algorithm is easy to implement and provides good compression (about fifteen percent of the size of an uncompressed inverted index when stop words are used). A better compression ratio is given by [48], but variable length encoding is more complex to implement.

2.1.1 Fixed Length Index Compression

As previously discussed, the entries in a posting list are stored in descending order by document identifier. Therefore, for any document identifier, only the *difference* between the current identifier and the identifier immediately preceding it is needed to represent an entry. For the case when no other document identifier exists, a compressed version of the document identifier is stored. Using this technique a high proportion of relatively low values is assured.

Since the posting list differences are generally small, they can be stored using only a small number of bits. That is, for each difference, the minimum number of bytes required to store this value is computed. Table 3 indicates the range of values that can be stored, as well as the length indicator for one, two, three, and four bytes. This indicator requires two bits. For document collections exceeding 2^{30} documents, this scheme can be extended to include a three bit length indicator which extends the range to $2^{61} - 1$.

Our example of Byte-aligned (BA) compression uses the leading two high order bits to indicate the number of bytes used to represent the value. There are four possible combinations of two bit representations; thus a two bit length indicator is used for all document identifiers. Integers are stored in either 6, 14, 22, or 30 bits. Optimally, a reduction of each individual data record size by a factor of four is obtained by this

Table 3: Index Difference Encoding

length	number of bytes required
$0 \leq x < 64$	1
$64 \leq x < 16,384$	2
$16,384 \leq x < 4,194,304$	3
$4,194,304 \leq x < 1,073,741,824$	4

Table 4: BA Encoding Example

value	compressed bit string
180	01 000000 10110100
63	00 111111
4	00 000100
2	00 000010
1	00 000001

method, since, in the best case, all values are less than $2^6 = 64$ and can be stored in a single byte. Without compression, four bytes are used for all document identifiers.

Consider once again our earlier posting list example. In this case, the differences between entries are 180, 63, 4, 2, and 1. For the last four values, only one byte is required; for the first value, 180, two bytes are required. Recall that using no compression, a total of twenty bytes (160 bits) was needed to store the posting list. Using BA compression requires only 48 bits (see Table 4).

2.2 Variable Length Index Compression

Witten, Moffat, and Bell [48] also use the differences in the posting list, and like BA compression, capitalize on the fact that for most long posting lists, the difference between two entries is relatively small. To compress the index differences, a family of universal codes, called Elias encoding [10], is used. This code represents an integer x with $2\lfloor \log_2 x \rfloor + 1$ bits. The first $\lfloor \log_2 x \rfloor$ bits are the unary representation of $\lfloor \log_2 x \rfloor$. (Unary representation is a base one representation of integers using only the digit one. The number 5_{10} is represented as 11111_1 .) After the leading unary representation, the next bit is a single stop bit of zero. At this point, the highest power of two that does not exceed x is represented. The next $\lfloor \log_2 x \rfloor$ bits represent the remainder of $x - 2^{\lfloor \log_2 x \rfloor}$ in binary.

As an example, consider the compression of the decimal 14. First, $\lfloor \log_2 14 \rfloor = 3$ is represented in unary as 111. Next, the stop bit is used. Subsequently, the remainder of $x - 2^{\lfloor \log_2 x \rfloor} = 14 - 8 = 6$ is stored in binary using $\lfloor \log_2 6 \rfloor = 2$ bits as 110. Hence, the compressed code for 14_{10} is 1110110, a seven bit representation.

Decompression is done in one pass because it is known that for a number with n bits prior to the stop

bit, there will be n bits after the stop bit.

Returning to our same example, the differences of 1, 2, 4, 63, and 180 are stored as shown in Table 5. This requires only 35 bits, thirteen less than the simple BA compression. Also, our example contained an even distribution of relatively large offsets to small ones. The real gain is that very small offsets require only a single bit.

Table 5: Elias Encoding Example

value	compressed bit string
180	11111110 0110100
63	111110 11111
4	110 00
2	10 0
1	0

2.3 Varying Compression Based on Posting List Size

The *gamma* encoding scheme varies the compression according to the posting list. Generalized, it is a coding paradigm based on a vector V containing positive integers v_i where $\sum v_i \geq N$. To code integer $x \geq 1$ relative to V , find k such that—

$$\sum_{j=1}^{k-1} v_j < x \leq \sum_{j=1}^k v_j$$

In other words, find the first component of V such that the sum of all preceding components is greater than or equal to the value, x , to be encoded. Assume that x equals seven. Using a vector V of $\langle 1, 2, 4, 8, 16, 32 \rangle$, we find the first three components (1,2,4) that are needed to equal or exceed seven. So, k is equal to three. Now k can be encoded in some representation (unary is typically used) followed by the difference:

$$d = x - \sum_{j=1}^{k-1} v_j - 1$$

Using this sum we have: $d = 7 - (1 + 2) - 1 = 3$ which is now coded in $\lceil \log_2 v_k \rceil = \lceil \log_2 4 \rceil = 2$ binary bits. With this generalization, the γ scheme can be seen as using the vector V composed of powers of 2 $\langle 2^0, 2^1, 2^2, \dots \rangle$ and coding k in binary.

Clearly, V can be changed to give different compression characteristics. Low values in v optimize compression for low numbers, while higher values in v provide more resilience for high numbers. A clever solution given by [51] was to vary V for each posting list such that $V = \langle b, 2b, 4b, 8b, 16b, 32b, 64b \rangle$ where b is the median offset given in the posting list.

Using our example of 1, 2, 4, 63, 180, the median, b equals four, results in the vector $V = \langle 4, 8, 16, 32, 64, 128, 256 \rangle$ (see Table 6). This requires thirty-five bits as well, and we can see that, for this example, the use of the median was not such a good choice as there was wide skew in the numbers. A more typical posting list in which numbers are uniformly closer to the median frequently results in better compression.

Table 6: Difference-Specific Encoding Example

value	compressed bit string
180	1111110 01001100
63	11110 000100
4	0 11
2	0 01
1	0 00

2.4 Updating Automatically Assigned Term Weights

The inverse document frequency (*idf*) of a given term t is defined as $\log \frac{N}{df_t}$ where N is the total number of documents in the collection and df_t is the number of documents that contain term t . The logarithm is used to scale down the collection frequency so that a single term does not skew the results of an entire query. Clearly, as shown in the above definition, the addition of a document requires the re-computation of the *idf* for *each term* in the collection.

Since the *idf* is simply an estimate, it is reasonable to expect that it does not need to be updated with every new document. This premise was initially demonstrated in [47] for a small document collection, but its validity for larger collections was previously not investigated. Experimentation using only very small collections is known to be not necessarily indicative for larger collections [2].

Researchers demonstrated that it is not necessary to update the *idf* for every new document [13]. In an unpublished study, the *idf* update interval was evaluated by using a training collection of fifty percent of the document collection. Sequences of text were added to the collection, and the *idfs* were updated at different intervals.

Consider a collection with one document that contains terms a , b , and c . Assume that the *idf*'s for each of these terms are computed. A new document with a , b , and d requires an update to the *idfs*. However, it may not be necessary to update the *idfs* if they are not significantly changed by the new document. This would be fine except a term such as d would now not be present in the inverted index even though its document has been added to the system. A user who searches for d would not find this document even though it was just added. Hence, even if it is not necessary to update the *idfs* very often, the risk is that a single term could appear between the updates of the *idfs* and that term could be extremely useful for obtaining accurate results.

In the study, a 320 file document collection was used. Files contained multiple documents from within the Wall Street Journal portion of the TIPSTER collection [20]. Each file was roughly one megabyte in size.

Fifty TREC queries were used. Initially, the training set consisted of 160 files. That is, files numbered 1-160 were used as the training set, and files numbered 161-320 were incrementally added to the collection. The *idfs* were initially computed using a training set. After training, *idfs* were only updated every u files, where $u = 10, 20, 40, 80,$ and 160 files. (160 files is equivalent to no updates of the *idf* except for the training set.) The effect on the average precision was measured every twenty files to determine the impact of not updating the *idf*.

Precision can be computed at various points of recall. Average precision refers to an average of precision at various points of recall. Figure 2 illustrates the average precision for various update frequencies. The five different lines shown are not significantly different from each other. The average precision scores for each update interval are shown in Table 7. It can be seen that the average precision does not vary by more than 0.3 percent.

Table 7: Training Set 1 (First 160 files)

Update Frequency	Total Number of Files								
	160	180	200	220	240	260	280	300	320
160	17.49	16.88	16.21	16.37	15.81	15.28	15.55	15.57	16.14
80	17.49	16.88	16.21	16.37	16.11	15.58	15.85	15.88	16.32
40	17.49	16.88	16.24	16.37	16.11	15.58	15.85	15.88	16.32
20	17.49	16.71	16.24	16.39	16.11	15.58	15.85	15.89	16.32
10	17.49	16.62	16.04	16.02	15.81	15.28	15.55	15.57	16.14

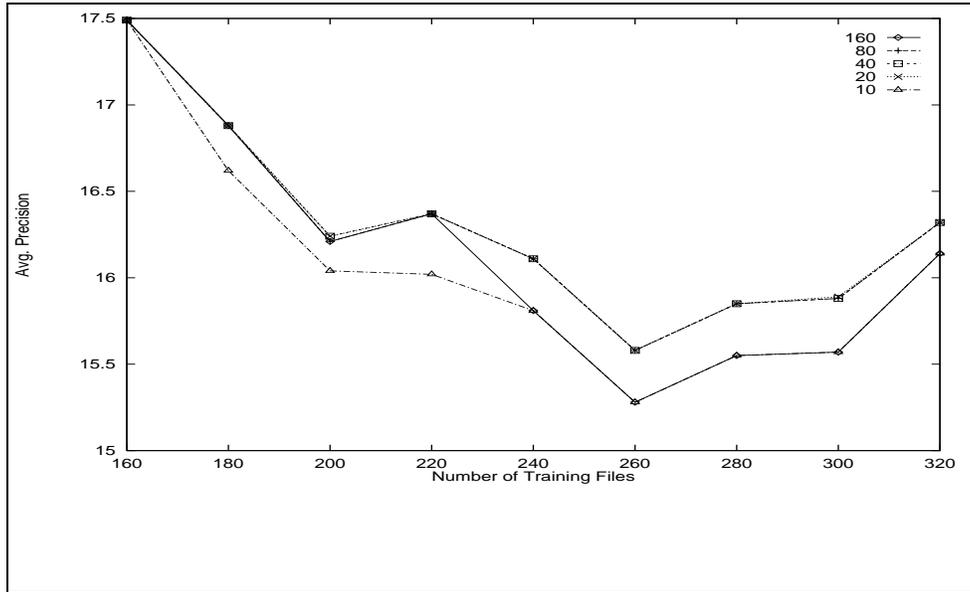
In the initial experimentation, no significant degradation in accuracy due to delayed *idf* updating was noticed. However, the order of appearance of new terms (order of document insertion into the collection), could significantly affect these results (i.e., order). Consider a case where a term that was not seen in the training set appears between *idf* updates and occurs in a relevant document. Hence, changing the order of the input files could result in such a term appearing in the training set and subsequently being found in a relevant document.

To test the impact of the input order, the authors reversed the training set and the document collection. In Figure 3, the results obtained when reversing the order are presented (see also Table 8). That is, files numbered 161-320 were used as a training set, and files 1-160 were incrementally added to the collection. Somewhat surprisingly, the order did not significantly affect average precision measurements. Hence, for a relatively large training set, the update of *idfs* has a negligible effect on accuracy.

The curves across Figures 2 and 3 should not be compared since the documents used in the training sets differ. Instead, what should be noticed is that in both sets of results the frequency of the update intervals of the *idfs* does not significantly affect the average precision as the precision numbers of the different update frequencies are roughly equivalent.

Having concluded that given a sufficiently large training set, the order of document insertion had little significance, the effects of the size of the training set were investigated. The training set size for the results presented in Figures 2 and 3 encompassed half of the document collection. The hypothesis was that the reason for the failure to detect any effect on average precision due to *idf* update frequency was because the

Figure 2: Training Set 1



training set was relatively large. To measure the effect of training set size on accuracy, training collections of $t = 80, 40, 20,$ and 10 files were tested.

In Figure 4, the results for these smaller training sets are shown (see also Table 9). In the study, the authors used the first t files, $t = 80, 40, 20,$ and 10 as a training set. A significant difference in the average precision measurements occurs when the training set drops below 20 files. At roughly 7% of the document collection size, the training set is small enough to illustrate the impact of extending the interval of the *idf* update. One reason for this effect is that there are far fewer distinct terms in a training set of size ten files than one of 160 files. In Table 10, the number of distinct terms for different training set sizes is shown. For a training set size of ten files, only 34,953 distinct terms were observed. The training set size of 160 files had nearly four times the number of distinct terms, thereby dramatically increasing the chance that a query term will appear in the training set.

The conclusion of the study was that given a sufficiently large (rich in the number of unique terms) training collection, the *idfs* need not be updated frequently to support high average precision measures. Thus, efficiency can be enhanced by initially collecting a sufficiently rich set of terms, computing their *idfs* prior to future document insertion, and then, only infrequently, updating the collection *idfs*. It remains an open question as to how to determine what is a sufficient set of terms to serve as the basis for the collection *idfs*.

Table 8: Training Set 2 (Files 161-320)

Update Frequency	Total Number of Files								
	160	180	200	220	240	260	280	300	320
160	20.71	20.04	18.72	17.99	17.56	17.69	17.27	16.59	16.32
80	20.71	20.04	18.72	17.99	17.44	17.60	17.15	16.53	16.30
40	20.71	20.04	18.80	18.07	17.44	17.60	17.14	16.48	16.30
20	20.71	20.06	18.80	17.95	17.44	17.58	17.14	16.52	16.30
10	20.71	20.06	18.80	17.95	17.44	17.58	17.14	16.52	16.30

Table 9: Effectiveness Difference as a Variation of Training Set Size

Training Set Size	Total Number of Files								
	160	180	200	220	240	260	280	300	320
160	20.71	20.04	18.72	17.99	17.56	17.69	17.27	16.59	16.32
80	20.00	19.35	18.32	17.74	17.29	17.48	17.08	16.54	16.29
40	20.90	19.89	18.61	18.07	17.37	17.70	17.25	16.45	16.20
20	17.48	17.34	16.38	15.99	15.39	15.75	15.47	15.28	15.03
10	16.15	15.98	15.08	14.56	13.95	14.37	14.13	14.00	13.80

3 Query Processing

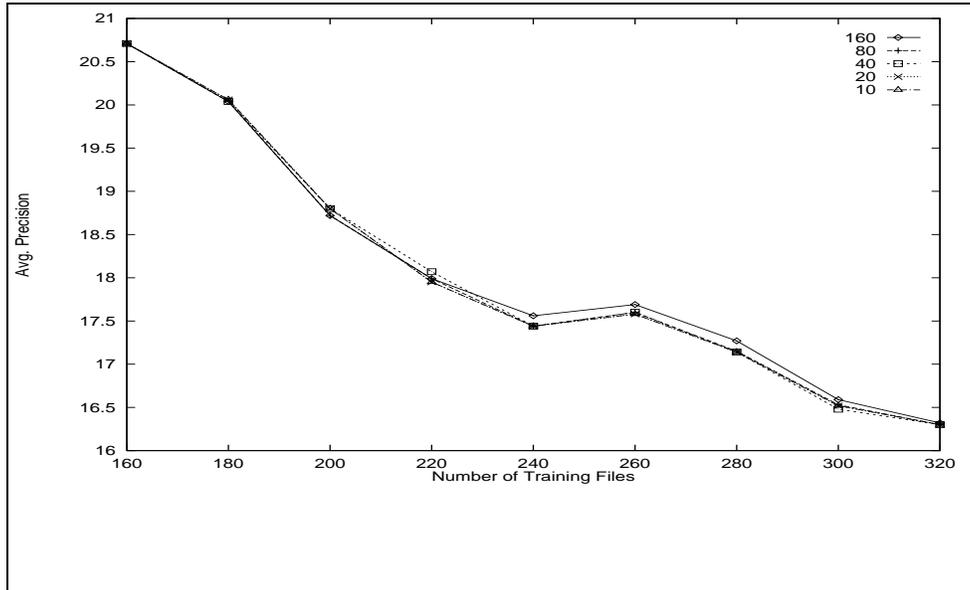
Recent work has focused on improving query run-time efficiency. Moffat and Zobel have shown that query performance can be improved by modifying the inverted index to support fast scanning of a posting list [28, 29]. Other work has shown that reasonable effectiveness can be obtained by retrieving fewer terms in the query [18]. A recent study showed that the computation can be reduced even further by eliminating some of the complexity found in the vector space model [21]. In this section, we review some representative work in improving query run-time efficiency.

3.1 Inverted Index Modifications

Witten, Moffat, and Bell show how an inverted index can be segmented so as to allow for a quick search of a posting list to see if a particular document is found [48]. The typical ranking algorithm scans the entire posting list for each term in the query. An array of document scores is updated for each entry in the posting list. Witten et. al. suggest that the least frequent terms should be processed first.

The premise is that less frequent terms carry the most meaning and probably have the most significant contribution to a high-ranking document. The entire posting lists for these terms are processed. Some algorithms suggest that processing should stop after d documents are assigned a non-zero score. The premise is that at this point, the high-frequency terms in the query will simply be generating scores for documents

Figure 3: Training Set 2



that will not end up in the final top t documents, where t is the number of documents that are displayed to the user.

A suggested improvement is to continue processing all the terms in the query, but only update the weights associated with the terms found in the d documents. In other words, after some threshold of d scores has been reached, the remaining query terms become part of an AND (they only increment documents who contain another term in the query) instead of the usual vector space OR. At this point, it is cheaper to reverse the order of the nested loop that is used to increment scores. Prior to reaching d scores, the basic algorithm is—

For each term t in the query Q

 Obtain the posting list entries for t

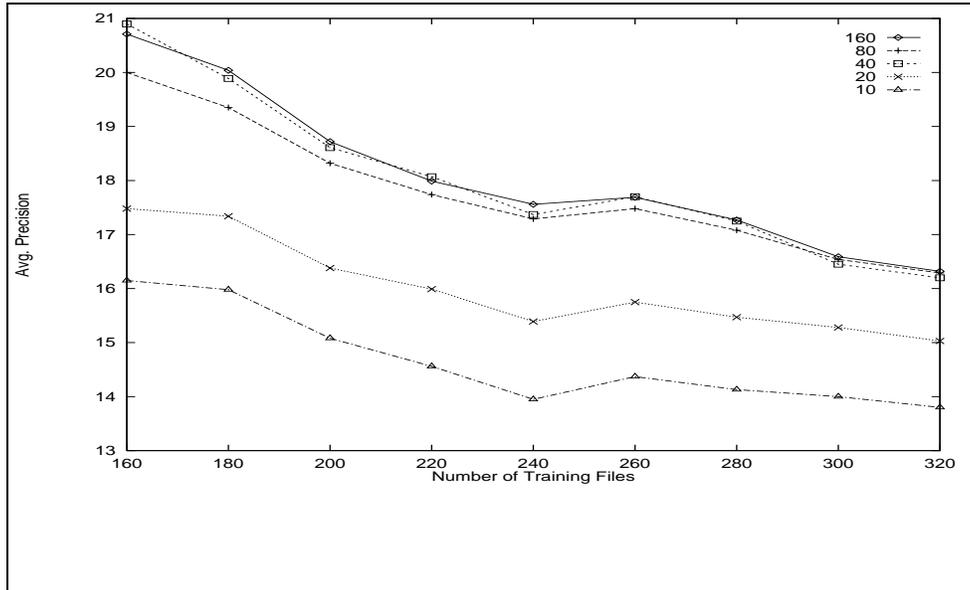
 For each posting list entry that indicates t is in document i

 Update score for document i

For query terms with small posting lists, the outer loop is small; however, when terms that are very frequent are examined, extremely long posting lists are prevalent. Also, after d documents are accessed, there is no need to update the score for every document. It is only necessary to update the score for those documents that have a non-zero score.

To avoid scanning very long posting lists, the algorithm is modified to be—

Figure 4: Effectiveness Difference as a Variation of Training Set Size



```

For each term  $t$  in the query  $Q$ 
  Obtain posting list,  $p$ , for documents that contain  $t$ 
  For each document  $x$  in the reserved list of  $d$  documents
    Scan posting list  $p$  for  $x$ 
    if  $x$  exists
      update score for document  $x$ 
    
```

The key here is that the inverted index must be changed to allow quick access to a posting list entry. It is assumed that the entries in the posting list are sorted by a document identifier. As a new document is encountered, its entry can be appended to the existing posting list. The authors propose to change the posting list by partitioning it and adding pointers to each partition. The posting list can quickly be scanned by checking the first partition pointer (which contains the document identifier of the highest document in the partition and a pointer to the next partition). This check indicates whether or not a jump should be made to the next partition or if the current partition should be scanned. The process continues until the partition is found and the document desired is matched against the elements of the partition. A partition size of about 1,000 resulted in the best CPU time for a set of TREC queries against the TREC data [29].

3.2 Partial Result Set Retrieval

Another way to improve run-time performance is to stop processing after some threshold of computational resources has been expended. One approach has been to count disk I/O and stop after a threshold has been

Table 10: Training Sets vs Number of Unique Terms

training set size	number of unique terms
160	140078
80	97408
40	69651
20	48730
10	34953

reached [49]. The key to this approach is to sort the terms in the query based on some indicator of term *goodness* and process the terms in this order. By doing this, query processing stops after the important terms have been processed. Sorting the terms is really analogous to sorting their posting lists. Three measures used to characterize a posting list are now described.

3.2.1 Cutoff Based on Document Frequency

The simplest measure of term quality is to rely on document frequency. This was described in [18, 19] which showed that using between twenty-five to seventy-five percent of the query terms after they were sorted by document frequency resulted in almost no degradation in effectiveness for the TREC-4 document collection. In some cases, effectiveness improves with fewer terms because lower ranked terms are sometimes noise terms, e.g., *good, nice, useful*. These terms have long posting lists that result in scoring thousands of documents and do little to improve the quality of the result. Using term frequency is a means of implementing a dynamic stop word list in which high-frequency terms are eliminated without using a static set of stop words.

3.2.2 Cutoff Based on Maximum Estimated Weight

Two other measures of sorting the query terms are described in [49]. The first computes the maximum term frequency of a given query term as tf_{max} and uses the following as a means of sorting the query. The tf_{max} is computed as the highest term frequency for the query term in any of the documents that contain this term.

$$tf_{max} \times idf$$

The idea is that a term that appears frequently in all the documents in which it appears is probably of more importance than a term that appears infrequently in the documents that it appears in. The assumption is that the maximum value is a good indicator of how often the term appears in a document.

3.2.3 Cutoff Based on the Weight of a Disk Page in the Posting List

The cutoffs based on term weights can be used to characterize posting lists and choose which posting list to process first. The problem is that a posting list can be quite long and may have substantial skew. To avoid this problem, a new measure sorts disk pages within a posting list instead of the entire posting list. At index

creation time, the posting lists are sorted in decreasing order by term frequency, and instead of just a pointer that points to the first entry in the posting list, the index contains an entry for each page of the posting list. The entry indicates the maximum term frequency on a given page. The posting list pages are then sorted by—

$$tf_{max} \times idf \times f(I)$$

where $f(I)$ is a function that indicates the number of entries on page I. This is necessary since some pages will not be full, and a normalization is needed such that they are not sorted in exactly the same way as a full page.

Unfortunately, this measure requires an entry in the index for each page in the posting list. However, results show (for a variety of query sizes) that only about forty percent of the disk pages need to be retrieved to obtain eighty percent of the documents that would be found if all one hundred percent of the pages were accessed. Note that the scalability of these results may be limited since these tests were performed using small document collections.

3.3 Query Processing in Web Search Engines

For web search engines, some additional problems occur with query processing. First, some documents are extremely long. These take up substantial space in the inverted index and also take more time to add to the inverted index. Every web search engine has some threshold for the size of a document, such that after that cutoff they no longer index the contents of the document. As of December 1997, web search engines were found to index only a third of the available document collection [39].

Also, it is to a web page author's advantage to have a web page show up as the result of a query *whether or not the web page is relevant to the query*. In this regard, a new problem occurs—that of an adversary actively trying to mislead a search engine. Hidden tags are often used to match queries with text that does not appear frequently in a document. We are not aware of any formal algorithms in this area, but web search engines all implement some functionality to avoid simply matching on hidden tags.

4 Modern Parallel Implementations

Partitioning the workload across a variety of processors in a logical single machine is referred to as *parallel processing*. First, we provide some background on parallel processing models and then describe approaches that parallelize access to the inverted index. For a given query, a sequential algorithm has to reference the inverted index for each term. A parallel algorithm can reference the inverted index for several terms simultaneously.

Most prior parallel IR systems were machine specific. These parallel machines, for the most part, did not achieve wide commercial acceptance. Therefore, the parallel IR systems based on these machines were not widely commercially accepted. We do discuss these machines—for an overview of work done in this area see papers including [1, 3, 6, 33, 34, 42, 43, 44].

We continue by describing the use of a parallel relational database system as a foundation for an IR system. The reliance on parallel databases is of interest since parallel database engines have achieved wide

commercial interest and are typically not machine-specific. Hence, it is reasonable to expect that such an IR approach can achieve commercial adoption.

Finally, we describe parallel algorithms for document clustering.

4.1 Background

Parallel architectures are often described based on the number of instruction and data streams, namely single and multiple data and instruction streams. A complete taxonomy of different combinations of instruction streams and data was given in [12]. To evaluate the performance delivered by these architectures on a given computation the speedup measure is typically used. *Speedup* is defined as $\frac{T_s}{T_p}$, where T_s is the time taken by the *best* sequential algorithm, and T_p is the time taken by the parallel algorithm under consideration. The higher the speedup, the better the performance. The motivation for measuring speedup is that it indicates whether or not an algorithm scales. An algorithm that has near linear speedup on sixteen processors may not exhibit similar speedup on hundreds of processors. However, an algorithm that delivers very little or no speedup on only two processors will certainly not scale to large numbers of processors.

Multiple Instruction Multiple Data (MIMD) implies that each processing element is potentially executing a different instruction stream. This is the case in most of the modern parallel engines such as the Intel Paragon and IBM SP2, as well as some of the earlier machines such as the Intel iPSC and the NCUBE/10. Synchronization is more difficult with this approach, as compared to a Single Instruction Multiple Data (SIMD) system because one processor can still be running some code while another is waiting for a message.

In SIMD architectures, all processors execute the same instruction concurrently. A controlling master processor sends an instruction to a collection of slave processors, and they all execute it at the same time on different sequences of data. In such cases, large speedups using SIMD engines are possible.

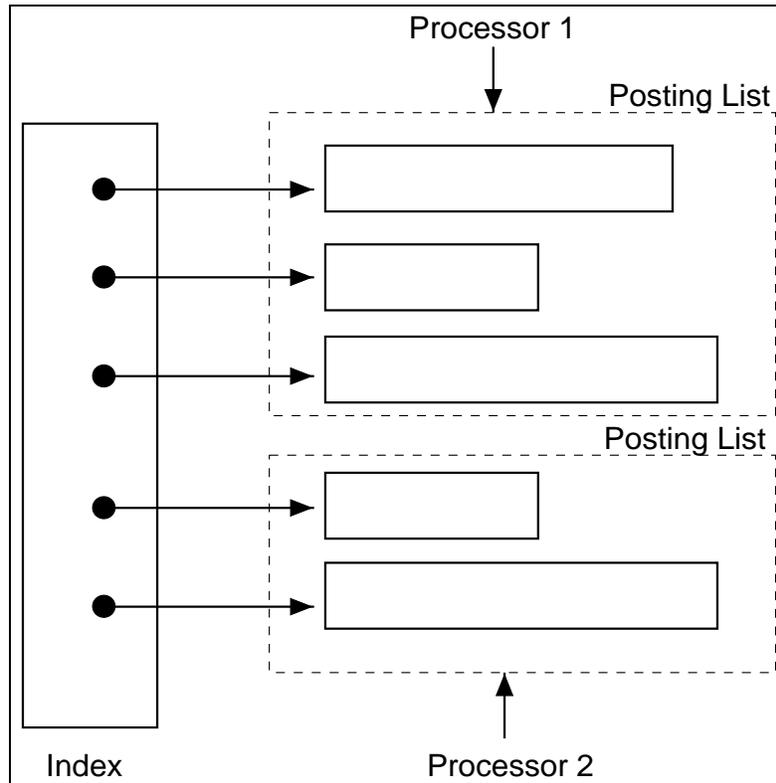
A primary approach to implement a parallel information retrieval system is to create a parallel index and distribute it across multiple processors. Figure 5 illustrates an inverted index that has been partitioned between two processors. This is intrinsically more difficult in that simply partitioning the index and sending an equal number of terms to each of the p processors does not always result in equal amounts of work. Skew in posting list size poses a difficult problem.

4.2 Partitioning a Parallel Index

An analytical model for determining the best means of partitioning an inverted index in a distributed memory/distributed I/O environment is given in [46]. Three approaches were studied. The first, referred to as the *system* approach, partitioned the index based on terms. The entire posting list for term a was placed on disk 1, the posting list for term b was placed on disk 2, etc. The posting lists were assigned to disks in a round-robin fashion.

Partitioning based on documents was referred to as the *disk* strategy. In this approach, all posting list entries corresponding to document 1 are placed on disk 1, document 2 on disk 2, etc. Documents were assigned to disks in a round-robin fashion. Hence, to retrieve an entire posting list for term a , it is necessary to retrieve the partial posting lists from each disk for term a and merge them. Although the merge takes more time than the system entry, the retrieval can take place in parallel.

Figure 5: Partitioning an Inverted Index



The *host* strategy partitioned posting list entries for each document and placed them on separate processors. Hence, document 1 is sent to processor 1, document 2 to processor 2, etc.

An analytical model was also developed by fitting a frequency distribution to some text (a more realistic approach than blindly following Zipf's Law). The results of the analytical simulation were that the *host* and *disk* strategy perform comparably, but the *system* strategy does not perform as well. This is because the *system* strategy requires sequential reading of potentially long posting lists and transmission of those lists. The *system* strategy becomes competitive when the communication costs are dramatically reduced.

4.3 Parallel IR As an Application of an RDBMS

One of the motivating features behind the development of an information retrieval engine as an application of the relational database model was the availability of commercial parallel database implementations. Executing the SQL scripts that implement the information retrieval application on a parallel relational database engine results in a parallel implementation of an information retrieval system. Given the commercial acceptance of parallel relational database technology, efficient implementations are widely available.

In [18], the feasibility of implementing a parallel information retrieval application as a parallel rela-

tional database application was demonstrated. Using a four processor NCR DBC/1012 database engine, nearly uniform processor loads and disk access rates were observed. From these findings, it was hypothesized that it was possible to develop a scalable, parallel information retrieval system using parallel relational database technology.

To validate this hypothesis, scaling experiments were conducted using a twenty-four processor NCR DBC/1012 database engine [24]. The initial findings were, however, disappointing. Using the same relational table definitions described in [18], only a forty percent processor efficiency was achieved. Further investigation revealed that the limiting factor to scalability was the non-uniform processor load on the DBC/1012.

The DBC/1012 supports automatic load balancing. The hashing scheme used to implement the load balancing is based on the index structure in the defined relational schema. In the DBC/1012 architecture used, to evenly distribute the load, a uniformly distributed set of attributes must be the input to the hashing function. In the initial implementation, the hashing function was based on terms, and thus, was nonuniform. Modifying the input to the hashing function to include document identifiers as well as terms resulted in a uniform distribution of load to the processors. In later experimentation, a balanced processor utilization of greater than 92% was demonstrated, and a speedup of roughly twenty-two using twenty-four nodes, as compared to a comparable uniprocessor implementation, was achieved.

It is important to note that the use of the relational model for document retrieval has been proposed since the late 1970's [25, 26]. The idea was basically dropped because of the increased overhead found in the relational model. Many inverted indexes work by storing the index in memory so no I/O is required to access the index and retrieval of posting lists often only take one or two I/O instructions. Using the relational model presumably will increase I/O because of the repetition of the document identifier in each element of the posting list. Additionally, all current commercial IR systems use an inverted index.

Hence, the ideas described in this sub-section are still the subject of some controversy. Several reasons exist to seriously consider the relational approach. The first is that the relational approach does not require the index to fit into main memory and very few I/O instructions are required to find a specific term. Also, the relational approach has been shown to scale due to parallel processing. Similar attempts to parallelize inverted index algorithms are described throughout this section, but they often fail to provide good speedup. Finally, the relational approach opens up the possibility of updating the inverted index without much difficulty. Updating a single posting list entry in an inverted index is non-trivial. The index entry must be found and the entire posting list must then be scanned. In an environment where updates to documents frequently occur, the relational approach offers additional flexibility.

4.4 Summary of Parallel Indexing

Parallel processing within information retrieval is becoming more applicable as the cost of parallel I/O is reduced. Previous algorithms had problems with memory limitations and expensive communication between processors. Signature files were popular but have not been used recently due to their unnecessarily high I/O demand and the difficulty in using them to compute more sophisticated measures of relevance. Parallel inverted index algorithms are becoming more popular, and with improved compression techniques, they are becoming substantially more economical.

4.5 Parallel Implementation of Clustering and Classification

Document clustering and classification algorithms partition the document collection into separate sets of related documents. The idea is that medical documents are placed in one cluster while documents about baseball are placed in another cluster. If a user query is submitted to only the relevant sets of documents in the collection, the search space is dramatically reduced.

Recently, parallel clustering and classification implementations were developed for the Intel Paragon [36]. Using a production machine, the authors developed a parallel implementation for the single-pass clustering and single-link classification algorithms. Using the *Wall Street Journal* portion of the TREC document collection, the authors evaluated the efficiency of their approach and noted near-linear scalability for sixteen nodes.

To accurately compare the efficiency of the developed approaches, the clusters derived from both the parallel and serial implementations must be identical. Otherwise, an improvement in the efficiency of the algorithm (via parallelism) could come at the expense of accuracy.

The single-pass clustering algorithm is data, presentation, and order dependent. Namely, the order in which the data are presented as input directly affects the output produced. Thus, it was necessary to provide mechanisms in the parallel implementation that mimicked the order of the presentation of the documents as input to the algorithm. Guaranteeing the identical order of document presentation resulted in the formation of identical clusters in both the serial and parallel implementations. The authors noted that the size of the clusters varied dramatically and suggested measures to reduce the cluster size disparity. Since the size disparity is a consequence of the single-pass algorithm, no modification was made.

5 Distributed Algorithms

Another means of improving efficiency is to distribute processing across many physical machines. Work on distributed IR systems began in the late 1980s. A prototype used to investigate implementation details of a distributed IR system is described in [27]. Today, every web search engine is essentially an implementation of a distributed IR system because the actual documents are stored across the Internet. We also describe general performance improvements to distributed IR systems.

5.1 Replication

Distributed structured DBMS algorithms were first developed in the early 1980s. These algorithms, such as the two-phase commit [5, 8, 11, 32], support updates to multiple sites in a single transaction. Since the premise behind information retrieval systems is that updates occur relatively infrequently, replication algorithms specific to information retrieval systems have been developed. These algorithms improve performance by ensuring that the data that are searched are physically *close* to the user.

A recent algorithm designed to support replication of Internet archives was given in [30]. Essentially, the algorithm estimates topological information between source nodes and replicas. Groups are then replicated based on their physical topology. Those that are deemed closest (lowest communication cost based on available bandwidth and propagation delay) are automatically grouped together. The source node then has to refresh only to one replica in a group and it can then be later propagated to each group member.

Individual replicas request data from a source member. Whenever a replica is transferred, the destination sends a message to members in its group indicating that it has received a replica that has a given timestamp. When a site receives this message, it checks to see if its current replica is out of date, and if so, it requests a copy. It is not possible for a site to receive data from more than one place at a given time because each replica is only provided data upon request. Overhead associated with this approach is relatively small and is primarily based on the frequency with which communication costs are estimated. The key to this approach is that the physical location of replicas is chosen automatically (some prior work required manual administration) and the heuristic constantly tries to send data to sites that have a low communication cost.

5.2 Improving Performance of Web-based IR Systems

Using a web server to implement an information retrieval system does not dramatically affect the types of algorithms that might be used. Some work is being done to apply web server performance improvements to the development of distributed information retrieval systems [22]. In those cases, the use of pre-started processes, or *cliettes*, avoids the start-up costs of starting processes from a typical common gateway interface (CGI). This was used to implement a prototype system that provides search access to eight library collections.

Early work in the area of web-based distributed query processing was done by [9] in which a system that used the Wide Area Information Service (WAIS) only sent queries to certain servers based on an initial search of the content of those servers. The content was described by some specific fields in the documents that exist on each server such as *headline* of a news article or *subject* of an e-mail message. The use of a content index is the middle ground between sending the request to all of the servers, or providing a very detailed full-text index, and sending the request to only those servers that match the index.

More recent work done for the Glossary-of-Servers Server (GLOSS) builds a server that estimates the best server for a given query using the vector-space model [15]. The query vector is matched with a vector that characterizes each individual server. The top n servers are then ranked and searched. Several means of characterizing a server are explored. The simplest is to sum the *tf-idf* weights of each term on a given server and normalize based on the number of documents on the server. This yields a centroid vector for each server. A *tf-idf* vector space coefficient can then be used to rank the servers for a given query. Different similarity coefficient thresholds at which a server is considered a possible source and assumptions used to estimate which databases are likely to contain all of the terms in the query are also used. The index on the GLOSS server is only about two percent of the size of a full-text index.

6 Summary

We described a variety of approaches to improve efficiency in IR. Without efficiency, users will simply not take advantage of an IR system to find the information they require.

We started with a discussion of the indexing. The inverted index is the basic structure underlying every current IR algorithm. Any means of improving the efficiency in which an inverted index is used significantly improves overall IR performance. Interestingly, an inverted index may be compressed to only ten percent of its original size. We described compression algorithms that improve run-time performance by reducing disk I/O.

We then described some recent research that questions the need to continuously update the inverse document frequency. The research efforts showed that effectiveness is not significantly degraded when the *idf* is not updated continuously—only infrequent updates are needed. This clearly impacts efficiency because updates to inverse document frequencies are computationally expensive.

Subsequently, we discussed query processing. Many techniques exist to reduce the workload required by a given query. It has been shown that careful removal of some terms in the query does not degrade effectiveness and can dramatically improve efficiency. Other simple techniques were also discussed.

Query processing and compression of inverted indexes are really the two most fundamental means of improving efficiency in a sequential environment. We then turned our attention to an environment with either multiple processors in a single machine (parallel processing) or multiple machines (distributed processing).

We described some parallel IR algorithms. Few algorithms fit into this class, but research in the use of a relational database system to serve as the foundation for an IR engine (thus treating the whole problem of IR as an application of a relational database system) has shown the potential to scale quite well using general purpose processors.

We also discussed distributed IR algorithms. These are more relevant to the current computing environment because every web search engine is an example of a distributed IR system. Current web search engines allocate only a single CPU second for a query of the entire web [41]. Hence, efficiency considerations are critical when implementing real systems.

Although greater attention has traditionally been placed on the effectiveness of information retrieval systems, efficiency issues are critical. Failure to optimize the efficiency of an information retrieval system can result in a highly accurate system that has prohibitive execution or storage performance. As storage technology continues to improve and decrease in cost, storage constraints are becoming less critical. However, with the continued exponential growth of online data, storage constraints are still a concern and run-time performance considerations are of tantamount importance.

Acknowledgment

We thank Nazli Goharian and Don Kraft for their insightful comments that dramatically improved the quality and clarity of this paper. Clearly, any remaining errors are ours and ours alone.

References

- [1] N. Asokan, S. Ranka, and O. Frieder. A parallel free text search system with indexing. In *Proceedings of the International Conference on Databases, Parallel Architectures, and their Applications (PARBASE-90)*, pages 519–534, 1990.
- [2] D.C. Blair and M.E. Maron. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Communications of the ACM*, 28(3):289–299, 1985.
- [3] N. Bond and S. Reddaway. A massively parallel indexing engine using DAP. *Cambridge Parallel Processing. Technical Report.*, 1993.
- [4] J.P. Callan. Passage-level evidence in document retrieval. In *Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 302–310, 1994.
- [5] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1984.
- [6] J. Cringean, R. England, G. Manson, and P. Willett. Parallel text searching in serial files using a processor farm. In *Proceedings of the Thirteenth Annual ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 429–445, September 1990.
- [7] M. Damashek. Gauging similarity via n-grams: Language independent categorization of text. *Science*, 267(5199):843–848, 1995.
- [8] C. Date. *An Introduction to Database Systems*. Addison-Wesley, 1994.
- [9] A. Duda and M. Sheldon. Content routing in a network of WAIS servers. In *Proceedings of the IEEE Fourteenth International Conference on Distributed Computing Systems*, pages 124–132, 1994.
- [10] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.
- [11] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 1994.
- [12] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.
- [13] O. Frieder, D. Grossman, A. Chowdhury, and G. Frieder. Phrase processing with the relational model and IDF update effects on precision/recall. Technical report, IBM Technical Report 05.500, April 1997.
- [14] M. Gordon. It’s 10 a.m. do you know where your documents are? The nature and scope of information retrieval problems in business. *Information Processing and Management*, 33(1):107–121, 1997.
- [15] L. Gravano and H. Garcia-Molina. Generalizing GIOSS to vector-space databases and broker hierarchies. In *Proceedings of the 21st International Conference on Very Large Database Conference*, pages 78–89, 1995.

- [16] D. Grossman. *Integrating Structured Data and Text: A Relational Approach*. PhD thesis, George Mason University, 1995.
- [17] D. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Press, 1998.
- [18] D. Grossman, O. Frieder, D. Holmes, and D. Roberts. Integrating structured data and text: A relational approach. *Journal of the American Society for Information Science*, 48(2):122–132, Feb 1997.
- [19] D. Grossman, D. Holmes, and O. Frieder. A parallel DBMS approach to IR. In *Proceedings of the Third Text REtrieval Conference*, pages 279–288, 1994.
- [20] D. Harman. The TREC conferences. In *Proceedings of the Hypertext - Information Retrieval - Multimedia: Synergieeffekte Elektronischer Informationssysteme (HIM'95)*, pages 9–28, 1995.
- [21] D. Lee and L. Ren. Document ranking on weight-partitioned signature files. *ACM Transactions on Information Systems*, 14(2):109–137, April 1996.
- [22] Y. Liu, P. Dantzig, C. Wu, J. Challenger, and L. Ni. A distributed web server and its performance analysis on multiple platforms. In *Proceedings of the Sixteenth IEEE International Conference on Distributed Computing Systems*, pages 665–672, 1996.
- [23] C. Lundquist. *Relational Information Retrieval: Using Relevance Feedback and Parallelism to Improve Accuracy and Performance*. PhD thesis, George Mason University, 1997.
- [24] C. Lundquist, O. Frieder, D. Holmes, and D. Grossman. A parallel relational database management system approach to relevance feedback in information retrieval. *to appear in Journal of the American Society for Information Science*, April 1999.
- [25] I. Macleod. A relational approach to modular information retrieval systems design. In *Proceedings of the ASIS Annual Meeting*, pages 83–85, 1978.
- [26] I. Macleod. SEQUEL as a language for document retrieval. *Journal of the American Society for Information Science*, 30(5):243–249, September 1979.
- [27] T. Martin, I. Macleod, and J. Russell. A case study of caching strategies for a distributed full text retrieval system. *Information Processing and Management*, 26(2):227–247, 1990.
- [28] A. Moffat and J. Zobel. Fast ranking in limited space. In *Proceedings of the Tenth IEEE International Conference on Data Engineering*, pages 428–437, 1994.
- [29] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.
- [30] K. Obraczka, P. Dantzig, D. DeLucia, and E. Tsai. A tool for massively replicating internet archives: Design, implementation, and experience. In *Proceedings of the Sixteenth IEEE International Conference on Distributed Computing Systems*, pages 657–664, 1996.

- [31] J. O'Connor. Some independent agreements and resolved disagreements about answer-providing documents. *American Documentation*, 20(4):311–319, 1969.
- [32] T. Oszu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.
- [33] C.A. Pogue and P. Willett. Use of text signatures for document retrieval in a highly parallel environment. *Parallel Computing*, 4(3):259–268, 1987.
- [34] S. Reddaway. High speed text retrieval from large databases on a massively parallel processor. *Information Processing and Management*, 27(4):311–316, 1991.
- [35] J. J. Rocchio. *The SMART Retrieval System Experiments in Automatic Document Processing*, chapter Relevance Feedback in Information Retrieval, pages 313–323. Prentice Hall, 1971.
- [36] A. Ruocco and O. Frieder. Clustering and classification of large document bases in a parallel environment. *Journal of the American Society for Information Science*, 48(10):932–943, October 1997.
- [37] G. Salton. An evaluation of term dependence models in information retrieval. *Lecture Notes in Computer Science: Research and Development in Information Retrieval*, pages 151–165, May 1983.
- [38] H. Schutze and C. Silverstein. Projections for efficient document clustering. In *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 74–81, 1997.
- [39] Web searches fall short. *Science News*, 153:286, 1998.
- [40] K. Sparck Jones and P. Willett. *Readings in Information Retrieval*. Morgan Kaufmann Publishers, Inc., 1997.
- [41] G. Spencer, D. Cutting, and J. Xu, 1998. Personal Communication with D. Grossman.
- [42] C. Stanfill and B. Kahle. Parallel free-text search on the connection machine system. *Communications of the ACM*, 29(12):1229–1239, 1986.
- [43] C. Stanfill and R. Thau. Information retrieval on the connection machine: 1 to 8192 gigabytes. *Information Processing and Management*, 27(4):285–310, 1991.
- [44] C. Stanfill, R. Thau, and D. Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the Twelfth Annual ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 88–97, June 1989.
- [45] H. Stone. Parallel querying of large databases: A case study. *Computer*, 20(10):11–21, 1987.
- [46] A. Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

- [47] C. Viles and J. French. On the update of term weights in dynamic information retrieval systems. In *Proceedings of the Fourth Annual Conference on Information and Knowledge Management (CIKM95)*, 1995.
- [48] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, 1994.
- [49] W. Yee, P. Wong, and D. Lee. Implementations of partial document ranking using inverted files. *Information Processing and Management*, 29(5):647–689, 1993.
- [50] G.K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wessley, 1949.
- [51] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of the Eighteenth International Conference on Very Large Databases*, pages 352–362, 1992.