

Evaluating Ranked Queries in Limited Time and Memory for Information Retrieval for Distributed Digital Libraries

June Boltzis
LILAC Centre, School of Library Studies
Clyde College, Elgin, Australia

Abstract

Ranking techniques are used to evaluate natural-language queries on text databases. Text databases are an important component of digital libraries. Effective ranking can be costly in memory and time: the database may contain millions of documents and queries can contain large numbers of terms. These information retrieval systems must access large volumes of text, often divided into several collections that may be held on separate machines. In many environments, such as current desktop computers, standard CPU speeds and volumes of memory are more than adequate to rapidly resolve queries, even on databases of many gigabytes of text. Techniques for locating answers to queries must therefore consider identification of probable collections as well as identification of documents that are probable answers, to avoid the situation in which all queries must be answered in full by all servers. In other environments, however, both memory and time are limited: examples include Internet search engines, corporate data servers, online product databases, and, at the other extreme, handheld computers with PCIMIA-slot disk drives. In this paper we show that use of centralised blocked indexes, expressly designed for a multi-collection environment, can meet these objectives and simultaneously reduce overall query processing costs.

1 Introduction

The use of information retrieval systems for management of text data is widespread, and their use is likely to accelerate with the advent of the digital library. All of these techniques reduce the time or memory required to resolve a query. Newspaper archives, library catalogues, and legislation repositories all require access by record content if they are to be useful and effective. However, they do not necessarily bound it. These text databases are also quite different from the more usual key-based databases, in that the access criteria for any given record cannot be reliably determined in advance. In a practical system, however, it is important to allocate a fixed quantity of each resource to each user, to avoid problems such as thrashing. Instead, a record may be retrieved based upon any combination of words contained therein, so a relatively large amount of index information must be maintained for each record. In such contexts an important design goal may not be to seek maximum effectiveness, or the most efficient query evaluation algorithm that yields effectiveness close to that of the most effective algorithm; rather, the design goal is to maximise effectiveness once other constraints have been imposed. This facility makes text databases ideal for managing the textual component of a digital library, since the rigid structure of a conventional catalogue system need no longer restrict the access path used to locate information.

A consequence of this result is that two-level indexes may be appropriate for large collections even if all of the data is physically present on one machine. Over the last decade, new ranking techniques have allowed the resources used to evaluate a ranked query to be greatly reduced.

These techniques are largely based on inverted files, which for each distinct term includes an inverted list of the documents containing that term. Queries are resolved by fetching and

processing the inverted list for each query term, and, for each document in the list, updating an accumulator holding a partially-computed similarity for that document.

Advances in efficiency include compression of inverted lists, to reduce disk transfer costs and increase the likelihood that lists are cached in memory; internal structuring of inverted lists, to reduce decompression costs [MZ96]; heuristic limits to the number of accumulators, to reduce the memory required to resolve a query [MZ96, PZSD96]; and frequency-sorted reordering of inverted lists to bring the most significant documents to the front [Bro95, PZSD96].

In this paper we investigate query evaluation techniques for yielding good effectiveness within strict per-user memory and time bounds. Another important characteristic of text databases is the kinds of queries used to access stored documents. We focus on frequency-sorted indexes, as they appear to be the only practical mechanism for avoiding the cost of fetching entire lists. As for conventional databases, Boolean queries can be used, but it is not easy to formulate a query that is a good representation of what might be a quite imprecise information need [Sal89]. Pursuing a line of investigation originally suggested by Persin [PZSD96], we compare two strategies for computing partial similarities: term-ordered, in which each inverted list is processed in turn; and weight-ordered, in which entries in inverted lists are processed in order of decreasing weight, regardless of which list they are in.

Ranked or informal queries, which can be expressed in natural language, are the preferred mode of access to text [Sal89, vR79]; in this case, answers are ranked by their statistical similarity to the query and the best answers presented to the user. However, in other respects queries are generally simpler than for conventional databases: most queries do not, for example, involve operators such as join. Our experiments show that term-ordered evaluation yields better effectiveness in limited time than does weight-ordered evaluation.

2 Resources required for ranking

Text databases do, however, tend to accumulate non-homogenous data. A practical text database has two major components, a collection of stored documents and an index into the collection. To be truly useful, a publisher's information retrieval system should store not only the text of their quality broadsheet newspaper, but also articles from their less prestigious products. Similarly, in the Internet different newsgroups have widely differing presentation styles. The only practical form of index for a large text retrieval system is an inverted file [WMB99, ZMR98], which consists of a vocabulary containing each distinct term t and, for each t , an inverted list containing a sequence of entries. For evaluation of ranked queries, each entry is a pair d, f_d, t containing a document identifier d and a frequency f_d, t , the number of times t occurs in d . This diversity means that it is common for a single site to be used as a repository for many separate collections, either centrally managed or stored on separate machines.

An example of this multi-database approach can be seen in the data, a large text collection used for information retrieval research [Har92]. By fetching and processing the inverted list for each term t in query q , the entries can be used to compute the similarity of each document to the query. The collection contains 3 gigabytes of documents drawn from newspapers and news-wire services. A typical similarity formula is the cosine measure; an effective variant of this formula is [ZM98]:

$$\text{cosine}(d, q) = \sum_{t \in d \cdot q} \log(N/f_t) \log(f_{d,t} + 1) / W_d$$

in which N is the total number of documents, f_t is the number of documents containing t (equivalently, the number of entries in t 's inverted list), and W_d is a measure of the length of d . Patent Office; and from other U.S. Federal Government sources.

In standard ranked query evaluation, processing is initialised by creation of a structure of accumulators, each representing the similarity of a distinct document. In the 2 gigabyte subset that we have been using in our research work there are nine subcollections, ranging in size from 175 Mb (56,920 documents) to 267 Mb (99,111 documents). In such environments queries may be posed to the set of collections as well as to individuals in the set. The inverted lists for the

terms in the query are then processed in turn, in order of increasing length (that is, decreasing rarity); in this processing, each d, f_d, t is extracted, the accumulator for d is located, and the value $d(t) = \log(N/f_t) \log(f_{d,t} + 1)$ is added to it.

In a shared system, the principal environment we consider in this paper, the main resource requirements for such ranked query evaluation are as follows. In this paper we consider how ranked queries to a distributed text database, such as the set of subcollections of , might be evaluated; and in particular we examine the impact of choice of evaluation method on retrieval effectiveness—the ability of the system to identify answers. First, an array of W_d values must be held in memory. The distributed text database itself might, by analogy with a distributed relational database, be a collection of remote sites, each with its own data; it might be a set of distinct text collections stored at the one site; or it might be a looser arrangement, such as is typified by the collection of text repositories available worldwide via Internet. Second, a structure of accumulators is required, one per query being simultaneously evaluated. We explore an arrangement in which a central site controls query processing by dispatching requests to the other sites and coordinating their responses. Third, buffer space is needed for inverted lists as they are read into memory. We show that the central site requires an index of the contents of the documents stored at all sites, but that the size of this index need not be onerous.

Fourth, time is required to locate and fetch uncached inverted lists from disk.

Fifth, CPU cycles are required for list processing and accumulator update. Sixth, additional CPU cycles are required once list processing is complete, to divide all accumulators by the appropriate W_d value and identify the highest similarities. Last, the highly-ranked documents must be fetched; this cost is common to all processing strategies and we therefore omit it from our analysis.

Boolean queries

There are two important access modes for document databases: boolean queries and ranked queries. In this section and the next we consider the implementation of these access strategies for conventional single-collection databases.

A boolean query consists of a set of terms connected by the operators and, or, and not. Some of the costs of ranking are shared between users. A document is an answer to the query if it contains the specified combination of terms. The same array of W_d values is used for all queries, and moreover each value can be effectively represented in 8 bits or less [MZSD94]. For example, the query “(dog or puppy or cat or kitten) and (food or eat)” might be constructed to retrieve documents that discuss the feeding habits of domestic pets. The memory required to store inverted lists in memory is also to some extent a shared cost; as experience with Internet search engines illustrates, some terms are much more commonly used than are others, and thus simple disk caching schemes such as LRU can help keep lists in memory for potential re-use.

This kind of query is very different from the queries posed to traditional database systems through the use of languages such as SQL, because there is no unique-key access, and it is impossible to anticipate what form or content any future query might have.

The single most important cost in current systems is the time required to seek to and fetch inverted lists from disk [ZWK00]. It is for this reason that compression is valuable. The simplest mechanism for processing such a query is to sequentially examine every document in the collection, checking for the presence (or absence) of the specified query terms. A compressed index is stored in a smaller region of the disk, reducing seek times; compressed inverted lists can be fetched much more rapidly than when uncompressed; and many more inverted lists can be held in memory simultaneously, increasing the benefit from caching. For collections of a few megabytes—a user’s personal mail directory, for example—this may well be an attractive approach. For larger collections, however, the scale of the problem quickly makes such an approach unwieldy. With current ratios between processor speeds and disk access times, the cost of decompressing the lists is more than offset by the other savings. At a disk transfer rate of 2 megabytes per second a one gigabyte collection takes about 10 minutes to simply transfer from disk. Identification of further improvements in caching was a specific aim of our research.

Construction of an index allows greatly improved processing speed. Moreover, list decompression costs need not be significant. Several optimisations to query processing based on reducing these costs have been described. The preferred approach to document indexing is the use of an inverted index [FHBYL92] (we discuss elsewhere the limitations of signature files, the best-known alternative [ZMR95]). In an inverted index all of the terms that appear in the collection are listed in a vocabulary, together with a pointer to an inverted list storing document numbers that contain that term. These strategies are based on reducing the number of accumulators used during query evaluation, which has the additional benefit of reducing per-user memory requirements. For example, if “kitten” appears in the first, fifth, sixth, and tenth documents of a collection, then the term “kitten” would be included in the vocabulary of the collection, and that vocabulary record would somehow indicate the list (1, 5, 6, 10). Traditional implementations of ranking used an array of accumulators, one per stored document; however, the great majority of similarity values are very small, and thus most accumulators are unnecessary.

Similarly, if the list (1, 4, 9, 10, 15) is associated with the term “cat”, then the “cat or kitten” part of the example query matches documents (1, 4, 5, 6, 10, 15). That is, answers can be selected by executing set operations on the corresponding inverted lists: intersection for and, union for or, and complement for not. The problem is then to decide which documents are allocated an accumulator and how many accumulators to use.

A simple technique is to fix the number L of accumulators based on either some heuristic such as “ L is 5% of the number of documents” [MZ96] or on the basis of the volume of working memory allocated to each user. Provided that the stored inverted lists are sorted by document number these set operations can be carried out quickly. Terms are processed by decreasing rarity; newly-referenced documents are allocated an accumulator until they run out, after which existing accumulators can be updated but no new accumulators can be created. By this stage, typically, query processing has proceeded from rare terms to more common terms. Query evaluation consists of parsing the query into terms, and building an expression tree capturing their relationships; searching the vocabulary for each term that appears; fetching the corresponding inverted lists; and performing the required set operations in a sequential manner. For typical queries only a small fraction of the index must be consulted, and only a small fraction of the collection will match the query. This approach can be effective because it is the documents containing the rarest terms that are most likely to be relevant.

In such a case the set of answers can be provided within perhaps 0.1 to 10 seconds, depending upon the exact nature of the query and the number of answers.

A naive representation might require as much space as the collection itself. Another technique for reducing the number of accumulators is to only allow an accumulator to be created for a document d if the combination of the term t ’s rarity and its frequency within the document — that is, the $d(t)$ value— is sufficiently great [PZSD96]. Better representations use compression and exploit the fact that the inverted lists are sorted, and for typical document collections a compressed inverted index of every word and number in the text (even including common words such as “the”) takes about 10% of the size of the indexed text [BMNM⁺93].

However, deciding whether to create an accumulator with this strategy requires use of an arbitrary parameter. A refinement to this technique is to not use the $d(t)$ value at all if it is too small, even if d has an accumulator, thus saving some processing time. If positional or adjacency queries—for example, information retrieval system as a phrase rather than as three unrelated words—are to be supported then indexes must also maintain word positions within documents, and, even when compressed, indexes occupy around 25% of the space required by the original text. A second arbitrary parameter is required to determine this threshold. The best current compression methods reduce English text to around 25% [BCW90], and, given that the text can be largely reconstructed from a word-level index, it is to be expected that word-level indexes too cannot be represented in much less than 25% of the size of the original text.

With this technique, it may additionally be necessary to set an explicit limit on the number of accumulators, to ensure that per-user memory requirements remain within bounds.

The advantage of this technique is that it allows dramatic improvements in the costs of accessing lists from disk. The compression of index entries does have one drawback. In conjunctive boolean

queries of the form “term₁ and term₂ and term_q”, the most efficient way to process queries is to establish a set of candidate answers by fetching the shortest inverted list; and then, for each remaining inverted list, checking each of the candidate documents against that list, removing from the set of candidates any not found in that inverted list. Since only the inverted list entries in which the terms are frequent can contribute to or initiate creation of an accumulator, by storing these entries at the start of the inverted lists only the start need be retrieved and decoded during query evaluation [Bro95, PZSD96]. That only the start of each list contain useful entries can be guaranteed by sorting the entries in each list, not by the standard method of increasing document number, but by decreasing in-document frequency. The primary operation in this evaluation strategy is checking a relatively small set of candidates for membership in a relatively long inverted list. By sorting entries with the same frequency on document number, and discarding the frequencies themselves (because they are implicit in the list structure), good compression can be achieved and index size is not affected [PZSD96]. If the lists are stored uncompressed then this checking can be performed rapidly with binary search, but if they are stored compressed using a variable length code then there is no alternative but to perform a linear search, decoding the inverted list as the search proceeds.

These techniques address all of the resource costs listed above: reducing the number of accumulators cuts both memory requirements and the time required to normalise and search the accumulators after inverted list processing is complete; both hierarchical structuring of lists and frequency-sorting greatly reduce the processing cost of decoding compressed inverted list entries; and frequency-sorting greatly reduces the volume of lists fetched from disk as well as increasing the benefit of caching. Moreover, for limits on the number of accumulators down to 2% of the number of indexed documents, they do not appear to degrade effectiveness.

3 Ranked queries

In a ranked query each document in the collection is assigned a numeric similarity to the query, and answers are presented to the user in decreasing similarity order.

Crucial to the success of a ranking system is the similarity measure. Frequency-sorting reduces the volume of data fetched from disk, but requires that two thresholds be chosen, which affect but do not limit the total number of accumulators. Moreover, neither frequency-sorting nor the other limited-accumulator schemes directly address the question of how best to process queries in a fixed period of time. There have been many heuristics discussed in the literature [FBY92, SB88, ZM95], which however all share several common features.

The first feature is that they weight rare terms more highly than common terms: appearance of “kitten” in both document and query is considered to be a more useful indication than occurrence of “the” in both. However, frequency-sorted indexes can be used by a quite different interleaved query evaluation algorithm that does lend itself to limited-time evaluation, first described by Persin in a thesis [PZSD96] but not to our knowledge otherwise published. The second common feature is that a large number of appearances of a term within a document is regarded as being a stronger indication of relevance than a small number of occurrences, irrespective of the number of times the word appears in the collection as a whole.

The third common feature is that relevance is usually weighted inversely by the “length” of the document in question, where length might mean physical length in bytes, or might be some calculated quantity based upon the weightings assigned to the terms contained in the document.

Interleaved query evaluation is based on the observation that the greatest $d(t)$ value for each term t is at the front of t 's inverted list. Rather than consume the whole of one inverted list before proceeding to the next, it is instead possible to consume just the greatest $d(t)$ value from any inverted list. To allow resolution of ranked queries the inverted file index must be augmented in two ways. First, within-document frequencies $f_{d,t}$ must be inserted for every stored document number, so that each inverted list becomes a sequence of “document number, frequency” pairs. Once this value has been consumed the lists are re-sorted on the basis of their $d(t)$ values, the next-greatest $d(t)$ value can be used, and so on.

The frequencies can also be compressed, and, relative to the original text, add just a few percent to the size of the inverted file [BMNM⁺93].

It is straightforward to incorporate limits to both accumulators and time into interleaved processing. First, since inverted list entries are processed in order of decreasing $d(t)$, the first documents referenced are those with the highest partial similarity, potentially an excellent heuristic for selecting documents that are likely to be relevant. The second additional structure required is of document weights $W_d = ({}_t d w_d, t^2)$. However, whether it is superior to allowing rare terms to select which documents have accumulators can only be determined by experiment. These are query invariant and so should be precalculated and stored. Second, query evaluation can simply be terminated after some fixed time, and it is guaranteed that all unprocessed $d(t)$ values are lower than those that have been used. Then, to process a query, the quantity $A(q, d) = \sum_{t \in q \cdot d} w_{q,t} w_{d,t}$ is evaluated by sequentially examining each inverted list, building up similarity contributions in an array of accumulators A .

When all of the terms have been processed the accumulators are each normalised by dividing by the corresponding W_d value, and the documents associated with the r largest similarities fetched and presented, for some user defined value r .

There are potential drawbacks to interleaved processing. As mentioned above, it is possible that it will be less effective than other techniques at choosing which documents are allocated accumulators. Ranked queries are fundamentally disjunctive (that is, based upon the or operator), and so the skipping technique described above is not, at face value, useful. However, if the number of potential answers is restricted to some a priori limit then the common query terms can be processed in a conjunctive manner, and for queries with the same number of terms and the same number of answers ranked query evaluation can be as fast as boolean query evaluation [MZ96]. It also has an additional cost, of searching amongst inverted lists after each list entry has been consumed. Another possible drawback is the initial cost of fetching the start of each inverted list into a buffer, to be available for searching. Imposing a limit on the number of candidates also has the useful effect of limiting the amount of memory required to store the accumulator data structure. (Fetching the start of each list on demand is likely to be even more costly, because if the lists are fetched at once the disk accesses can be sorted, reducing the seek cost that—for small volumes of data such as the first disk block of an inverted list—is the vast bulk of the disk access cost.) In the absence of such a limit the accumulator structure can be expensive to store if the database contains a large number of records and queries contain many terms.

Nor are the implications of these results clear; they are based on a relatively small collection and a single set of longer queries with a large number of relevant documents.

4 Indexing of multiple collections

By modifying this mechanism to perform a select every k th term-frequency pair rather than at every term-frequency pair the number of these costly operations will be reduced at the expense of not processing the most valuable entry at any instant. Diverse collections of data can be stored by a single information retrieval system, and some information needs can span multiple collections. It is envisaged that this will provide a reasonable trade off in terms of performance versus effectiveness which will be examined in the next section.

This is exemplified by the data, the first 2 Gb of which contains nine distinct collections. Experimental results Some queries would properly be directed only a single collection, such as the more recent set of newspaper articles from the Wall Street Journal.

But many of the information needs in the test queries are broad and are dealt with in all of the collections.

The test collection used for these experiments was the TREC-8 collection. It is thus natural to consider the problem of multi-database text collections, and ask how best boolean and ranked queries can be supported. The document collection is approximately 2 GB in size and comprises 528,155 documents drawn from newswires (1.5GB) and the Federal Register (0.4GB).

A query is posed to the central server (the “master”), which examines the central index to

determine which parts of which collections should be examined in more detail. Step one in the communications flow occurs when the master initiates one or more “slave” processes at the sub-collection level, and informs each of the original query and of the section of the slave’s database that should be examined in detail.

The slaves then examine their own detailed indexes, and each forms a ranking of the subset of their own records they were asked to examine by the master.

When all slaves have returned their lists of candidates, the master merges the lists, perhaps adjusting the similarity values according to any global information that might be available, and then forms a global ranking. In the third communication stage the master then directly requests copies of documents from the slaves, asking each slave for exactly those documents necessary to complete the global ranking. The fourth communication step is when the slaves each return text that they have fetched from their databases.

5 Restricting memory usage

The amount of memory that a query engine uses to service a query can be controlled by the number of accumulators that it is permitted to create. The master process does not permanently store any of the text of the collection, so there is no redundant storage of documents. Accumulators must be allocated on a per query basis and are not a resource that can be shared across the evaluation of multiple queries. The master does, however, require some index information in order to be able to direct the search undertaken by the slaves. In the absence of any central index information the master has little alternative but to instruct each of the slaves to undertake the query in full, and then merge the resultant similarity lists. Therefore, each query requires its own set of accumulators with the set regularly growing towards the number of documents in the collection if their allocation is unrestricted. At the other extreme, if the master has access to a complete index describing the union of all of the subcollections, then it can completely direct the search, and the second and third communication stages become unnecessary. On a multi-user system this combination of large, mutually exclusive memory allocations can place unreasonable demands on the underlying hardware. Fortunately, these concerns can be alleviated by limiting the number of accumulators used during the processing of each query. Maintaining full index information at the master does, however, involve extra storage space.

However, this incurs the penalty of having to evaluate the query in a sub-optimal environment where not every document is allocated an accumulator. In distributed relational databases, query processing can effectively be conducted autonomously at each site [Dat83, OV91]; no master is required. In systems where accumulators are poorly allocated the effects of restricting the number of available accumulators can be manifested in very poor retrieval effectiveness. In the context of ranking, however, some central control is needed because of network costs. Therefore, it is important to minimise the effects of reducing the number of available accumulators through efficient accumulator allocation. This issue is discussed further below.

Interleaved processing and its strategy of evaluating the most highly weighted document at any instant offers the opportunity to examine the efficient allocation of memory resources. In order to explore the tradeoff between these two extremes of behaviour, we undertook a range of experiments with the data, query set, and relevance judgements. In particular, by comparing weight ordered processing with term ordered processing we shall determine whether accumulators should be allocated only with regard to term weight or whether within document frequency should also be considered.

The experimental regime was as follows. An ideal system with n accumulators to allocate, would apportion the n accumulators to the n highest scoring documents. Each collection had a separate index, listing for each word the documents containing the word and the in-document frequencies. However, it is not possible to know in advance which documents will score highly so the system must attempt to allocate accumulators to documents that it believes will score highly. The indexes also stored word frequencies, to allow computation of inverse document frequencies in each collection. A system that accurately makes this prediction will achieve high retrieval effectiveness

in bounded memory. Thus, for each collection, similarity values were computed based only upon the information within each collection, and global information was assumed to be unavailable.

That is, are we better served by allocating accumulators on the basis of term importance (term-ordered) or should we also take into account the number of times that a term appears in a document (weight-ordered). As outlined above, query evaluation with centralised indexes proceeds as follows. In order to determine which mechanism is superior in this regard, the query engines were run in unbounded time with varying limits on the number of accumulators in use. The centralised index is used to find the blocks that are highly ranked with respect to the query. The number of accumulators used were 1000, 5000, 25000 and 528155, representing approximately 0.2%, 1%, 5% and 100% of the total number of documents in the collection respectively.

If r answers are required, at least r/B blocks must be identified; but almost certainly highly ranked blocks will contain some irrelevant documents, so more than r/B blocks should be fetched. Allowing the query engines unrestricted time in which to operate results in each inverted list being processed to completion. Even fetching r blocks does not guarantee that the top-ranked documents will be retrieved, particularly when B is large, since a highly similar record might have its strength diluted by occurring in the same block as many records containing no query terms. Exploration of the number of blocks that should be fetched is an important theme of these experiments. This means that any differences between the retrieval effectiveness of the engines is directly attributable to their accumulator allocation strategies.

Once the blocks have been ranked, the query is dispatched to each collection with matching blocks. The whole of each such collection could be ranked against the query, so that the purpose of the centralised index is to heuristically select a subset of the collections for further query evaluation.

Consider the behaviour of the centrally indexed system as B is varied. With $B = 1$, the central index is ranking documents directly, but this ranking is not the same as would be given by querying the individual collections, as the $(N/f_t + 1)$ values will vary between collections. Simply requesting the top r answers may omit documents that would be ranked highly in their collection, so fetching $r' > r$ blocks (each of one document) may yield better retrieval.

Interestingly a high k value improves the performance of interleaved processing at low levels of accumulator allocation. As B is increased further central index size and central processing costs continue to fall, but with diminishing returns. Reading k frequencies at a time makes the evaluation process more termwise in nature, with the effect becoming increasingly noticeable with small numbers of accumulators in use.

Larger blocks imply higher probability of false block match, so that, even if the number r' of blocks considered is decreased, the overall number of documents that must be ranked increases. The effectiveness of term ordered processing suggests another heuristic for increasing the query engine's memory efficiency. We would expect that, for sufficiently large B , effectively the whole of each collection would have to be ranked to produce good retrieval effectiveness, somewhat defeating the purpose of a central index.

Note that we have not in any way supposed that blocks are constructed according to relatedness of topic. If blocks consisted of similar documents then retrieval performance for a given r' should improve. For example, if a cluster-based (rather than chronological) insertion policy was adopted then the likelihood of the records in a block being "about" the same topic would increase.

6 Experimental results

The queries were drawn from the medium query set and the processing was term ordered. We have experimented with a centralised index by regarding the data as consisting of nine separate collections, ap1, ap2, doe, fr1, fr2, wsj1, wsj2, ziff1, and ziff2. The centralised index was implemented by using the mg system to index the data, but with most of the document separators deleted to mimic aggregation of documents into blocks.

Thus reindexing was required for each value of B tested. Using a hard limit of 5000 accumulators and a cutoff after the first term has been processed, it was found that retrieval effectiveness ??

topics 51–150 were transformed into mg input by removal of SGML markup and common words such as “the” and “however”. The output of each query was a list of block numbers, which were transformed into lists of document identifiers by Unix utilities. However there was a significant increase in memory efficiency with the average number of accumulators in use falling to 7. The problem with rigidly enforcing this scheme is that although it appears useful when averaged over 50 queries, there were pathological cases within the 50 queries where the scheme was extremely detrimental to retrieval effectiveness. We had already precomputed for each collection a 10,000-document list of similarity values for each query; and so final lists of answers were computed by joining the query output to these values. For example, when a query containing a very rare term is processed an insufficient number of accumulators are created, making effective retrieval impossible. That is, the process of selecting answers from collections was simulated, and while this gives exact values for retrieval effectiveness, we can only estimate the CPU time required by this phase of query evaluation.

This problem could be eliminated by the imposition of a lower limit below which the query engine must continue to allocate accumulators. A query engine organised along these lines would still have the same bounded memory efficiency as an engine without the first term cutoff however the average memory use would be considerably less, because the engine is less likely to allocate all of the accumulators that it is permitted to. As a baseline, we also indexed the full collection in the normal way—that is, as one monolithic collection—and computed retrieval effectiveness as an 11-point average assuming that 1,000 documents are returned as answers for each query. Similarly, using the first term cutoff alone without the hard limit of x accumulators is not useful in our discussion as we are trying to bound the amount of memory that the query engine can use. The result (queries 51–150) was an effectiveness of 29.4%. When time is unlimited and memory is restricted, conventional term-ordered processing uses the available memory more efficiently.

The next section of this paper considers how this situation is affected when the available processing time is also restricted.

For higher B , false block match rapidly becomes a significant problem. For blocks of 10 records, even fetching 1,000 blocks (that is, $Br' = 10,000$) results in a loss of retrieval effectiveness. For $B = 1,000$, the bulk of the collection must be processed to achieve good performance. The reasons for the superiority of term ordered processing over weight ordered processing are three fold. Firstly, interleaved processing requires a select and heapify operation for each unique term-frequency pair that is evaluated. These problems are to some extent due to the experimental methodology. Secondly, whereas the blocks for each vector are read sequentially during term-ordered processing they can be read randomly during weight ordered processing with dramatically increased IO costs. The procedure of fetching 1,000 records is designed to ensure high recall, so that when the answer sets of the various participating research groups are pooled there is some degree of certainty that most of the relevant documents have been located. But for ad hoc queries far fewer than 1,000 answers are typically requested. The third factor is an artefact of the observation that term ordered processing selects better accumulators. If the system selects better accumulators then it is processing more relevant index entries earlier, therefore it will have better retrieval effectiveness after processing the same number of index entries, regardless of its more efficient execution. To test the performance of a central index in this context, we reran the experiments assuming that only 10 answers are returned for each query.

But, if we equate volume of index inspected with ranking time, in the $B = 1$ case all of the central index must be examined, plus roughly $10/742,000$ 'ths of the document indexes; while in the $B = 100$ case, a central index one fifth the size must be examined, plus $10,000/742,000$ 'ths of the component indexes, a net saving in index processing time. We are currently developing a full implementation to test these hypotheses experimentally.

Restricting the amount of times that a select takes place we can significantly improve the performance of interleaved processing without sacrificing retrieval effectiveness.

The other drawback of interleaved processing is the increased IO costs generated by not reading the blocks of each vector in a sequential manner. There is a steady decrease in index size as B increases, but at a lesser rate. At $B = 100$, for example, the central index still occupies about 1% of the space of the source text. In term-ordered processing each vector is read to completion,

therefore all of its blocks are read sequentially. Conversely during weight ordered processing small reads are made from the different vectors at different times, therefore the reads are not guaranteed to be sequential. The nine subcollection indexes totalled 118.4 Mb, and it is interesting to note that for such large volumes of data markedly better compression (compared to the $B = 1$ value of 127.6 Mb) can be achieved by partitioning the indexes according to subcollection. Not reading each vector sequentially translates into additional disk seeks and reduced IO effectiveness. This in turn is indicative of quite different term frequencies across the collections—words common in some subcollections are rare in others.

These experiments have led to some interesting observations. One is that there appears to be no problem with, for a query, combining similarity values from separate collections, even though the weights for the query terms vary.

For a greater number of collections, or smaller collections, however, difficulties may arise; this potential collection fusion problem is the subject of active research [CLC95, VGJL94]. A graph showing precision at 20 documents retrieved for systems using the medium query set with 1000 accumulators and varying windows of processing time. Another observation is that tiny perturbations in weights can substantially perturb rankings, since even the 1,000th document in a ranking often has a similarity value that, numerically, is greater than 50% of the score achieved by the top ranked document.

Callan, Lu, and Croft have observed similar behaviour, and concluded that ignoring collections based upon gross evidence leads to loss of recall [CLC95]. Indeed, our results indicate that an even stronger claim can be made: the ordering of collections given by ranking of vocabularies is little better than random.

The experimental results described above have implications for two existing text retrieval systems, GLIMPSE [MW93] and GIOSS [GGM95, GGMT94]. In GLIMPSE (which is not a distributed system) documents are in effect gathered into buckets that are indexed as a single document. A graph showing precision at 20 documents retrieved for systems using the medium query set with 5000 accumulators and varying windows of processing time. Buckets that match a boolean query are retrieved in full, then scanned to identify any matching documents. Our results are for ranked queries, but we believe that similar behaviour would be observed for boolean queries: a bucket of several documents could easily contain all of a set of query terms even if no individual document in the bucket do so.

GIOSS [GGM95, GGMT94] is a component of a distributed text retrieval system; it provides an index of text databases, and is used to identify collections that are likely to contain answers to boolean or ranked queries. (It does not provide access to the collections themselves.)

The index is based on the vocabulary of each indexed database: for each database, it stores the total number of stored documents and, for each word in the database, the number of documents containing this word. This is equivalent to indexing each collection as a single block. But as we have shown, blocks of 1,000 records provide poor discrimination between collections, and indexing of whole collections as a single document is almost completely ineffective.

7 Conclusions

We have described a possible architecture for a distributed text database system, an integral component of the digital library. The primary design aim of the system was that it be able to support information retrieval on the full text of stored documents, but our proposals extend naturally to the indexing of other forms of stored data such as images, video clips, and sound bites. We have argued that, if query evaluation costs are to be kept to reasonable levels, that the system should be a master-slave arrangement.

Thus one site should control query evaluation, dispatching requests to the various collections and coordinating their responses. By use of a simple protocol the network costs can be kept low.

A potential drawback of this approach is the need for a central index. Use of compression reduces index size to around 10% of the size of the stored data, but even this figure could be onerous for a distributed system of many collections. A solution is to index blocks of documents

as single entities; a blocking factor of 100 reduces index size to around 1%, for example.

But we have also shown that excessively large blocks can increase net indexing processing costs, by requiring that the slaves consider large portions of their collections in response to each query.

That is, there is a trade-off between the size of the central index and query evaluation time. Note however that we have assumed that documents are grouped into blocks effectively at random (since the order of documents in our collections is not by semantic content) and clustering on some basis—such as document similarity—could well improve performance. The major cost of weight-ordered evaluation is the need to initially load the first few kilobytes of each list, but as we show it is more costly to entirely disregard some lists, as would be required for time-limited term-ordered evaluation.

We believe that, due to caching, the cost of initial load is likely to be low in practical multi-user systems. However, the other problem, of poor choice of documents for ranking, remains. We have also shown that ranking of collections according to the contents of their vocabularies is not likely to be effective. Indeed, even use of large blocks, such as of 1,000 documents each, leads to poor performance. An interesting consequence of our experiments is that, for moderate block sizes, overall indexing costs were reduced compared to indexing of the collections as a whole. Thus the central index in effect provides a two-level indexing scheme. It may well be that, for large collections in particular, two-level indexing is an attractive approach.

References

- [BCW90] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
- [BMNM⁺93] T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Jour. of the American Society for Information Science*, 44(9):508–531, October 1993.
- [Bro95] E. W. Brown. Fast evaluation of structured queries for information retrieval. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proc. 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 30–38. ACM Press, New York, July 1995.
- [CLC95] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval*, pages 21–28, Seattle, Washington, July 1995.
- [Dat83] C. J. Date. *An Introduction to Database Systems*, volume II. Addison-Wesley, Massachusetts, 1983.
- [FBY92] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.
- [FHBYL92] E. A. Fox, D. K. Harman, R. Baeza-Yates, and W. C. Lee. Inverted files. In Frakes and Baeza-Yates [FBY92], chapter 3, pages 28–43.
- [GGM95] L. Gravano and J. H. Garcia-Molina. Generalising GLOSS to vector-space databases and broker hierarchies. In *Proc. Int. Conf. on Very Large Databases*, Zurich, Switzerland, 1995.
- [GGMT94] L. Gravano, J. H. Garcia-Molina, and A. Tomasic. The effectiveness of GLOSS for the text database discovery problem. In R. T. Snodgrass and M. Winslett, editors, *Proc. ACM-SIGMOD Int. Conf. on the Management of Data*, pages 126–137, Minneapolis, Minnesota, 1994.
- [Har92] D. K. Harman. Overview of the first Text Retrieval Conference. In D. K. Harman, editor, *Proc. TREC Text Retrieval Conference*, pages 1–20, Washington, November 1992. National Institute of Standards Special Publication 500-207.
- [MW93] U. Manber and S. Wu. GLIMPSE: a tool to search through entire file systems. Technical Report TR 93–34, Department of Computer Science, University of Arizona, October 1993.
- [MZ96] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.
- [MZSD94] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Information Processing & Management*, 30(6):733–744, November 1994.
- [OV91] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, New Jersey, 1991.
- [PZSD96] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Jour. of the American Society for Information Science*, 47(10):749–764, 1996.
- [Sal89] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, MA, 1989.

- [SB88] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. 24(5):513–523, 1988.
- [VGJL94] E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird. The collection fusion problem. In D. K. Harman, editor, *Proc. Text Retrieval Conf. (TREC)*, pages 95–104, Gaithersburg, Maryland, 1994. NIST Special Publication 500-225.
- [vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.
- [WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, California, second edition, 1999.
- [ZM95] J. Zobel and A. Moffat. Similarity measures explored. Technical Report TR-95-3, Collaborative Information Technology Research Institute, RMIT and The University of Melbourne, January 1995.
- [ZM98] J. Zobel and A. Moffat. Exploring the similarity space. *SIGIR Forum*, 32(1):18–34, Spring 1998.
- [ZMR95] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. Technical Report TR-95-5, Collaborative Information Technology Research Institute, Departments of Computer Science, RMIT and The University of Melbourne, Australia, February 1995.
- [ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.
- [ZWK00] J. Zobel, H.E. Williams, and S. Kimberley. Trends in retrieval system performance. In J. Edwards, editor, *Proc. Australasian Computer Science Conf.*, pages 241–248, Canberra, Australia, January 2000.