Retrieval of Partial Documents

Alistair Moffat* Ron Sacks-Davis† Ross Wilkinson‡ Justin Zobel§

Information systems usually retrieve whole documents as answers to queries. However, it may in some circumstances be more appropriate to retrieve parts of documents. These parts could be formed by arbitrary division of running text into pieces of similar length, or by considering the document's hierarchical structure. Here we consider how to break documents into parts, how to implement retrieval of parts, and the impact of division of documents on retrieval effectiveness.

1 Introduction

Provision of answers to informally phrased questions is a central part of information retrieval. These answers traditionally take the form of documents retrieved from a text database, but documents will often be unsatisfactory as answers. They may be large and unwieldy; the answer they represent may be diffuse, and therefore hard for the user to extract; and word-based retrieval systems may be misled by the breadth of vocabulary of a long document into believing it to be relevant.

Indexing and returning parts of documents addresses these problems. We have approached the problem of partial documents in two ways. The first approach is to regard documents as an unstructured series of "pages" of text of similar length, each of which can be returned as an answer to a query. We would expect, under this approach, that any bias in the retrieval mechanism towards documents of a particular length should be eliminated. By regarding an answer to be the document from which an answer page is drawn, paging can be used even in contexts where documents are required as answers, as is the case for the TREC experiments.

Breaking documents into pages, however, has implications for implementation: the growth in the number of candidate answers is such that current approaches for evaluating queries have unacceptable memory requirements and response times. We have developed new algorithms for implementing information retrieval methods on large collections, concentrating on the cosine measure with IDF term weights as a typical example. These include techniques for efficiently constructing and compressing large inverted files, and for restricting the amount of memory space and processing time required during query evaluation. The result is that we are able to identify answers in a fraction of the space and time required by previous methods. Using these techniques on a version of the TREC data in which the documents are broken into over 1.7 million pages, answers can be found more quickly than they could previously be found on the unpaged data, even though the latter has a smaller index and fewer records. Section 2 gives an overview of these techniques.

Our second approach to the problem of partial documents was to regard documents as hierarchical structures. Some of the documents in TREC are very large. It is not clear that it is desirable to return such documents as a whole, nor is it clear that these documents should be indexed as a whole. Most of the long documents in TREC are from the Federal Register collection and all have some degree of structure associated with them. We conducted a set of experiments that attempted to determine whether these documents should be indexed as single objects, and whether the documents' structure could be used in conjunction with the contents of its elements.

We also experimented with retrieval of partial documents and investigated whether context, that is the rank of the whole document, helped improve ranking of sections. The experiments with hierarchical structures are necessarily based on the small set of longer documents in the TREC

*Dept. of Computer Science, The University of Melbourne, Parkville, Victoria, Australia 3052; alistair@cs.mu.oz.au

†Collaborative Information Technology Research Institute, 723 Swanston St., Carlton, Victoria, Australia 3052; rsd@kbs.citri.edu.au

‡Dept. of Computer Science, RMIT, GPO Box 2476V, Melbourne, Victoria, Australia 3001; ross@cs.rmit.oz.au

§Dept. of Computer Science, RMIT, GPO Box 2476V, Melbourne, Victoria, Australia 3001; jz@cs.rmit.oz.au

collection. We would expect that, in a large database of such documents, the implementation techniques we developed for unstructured text could be used for structured documents almost without change.

2 Retrieval of paged text

The first issue that must be considered when accessing a set of documents as a collection of pages is the pagination strategy; one possible method for pagination is discussed in Section 2.1.

Then, given paged text, answers must be found in response to documents. In an inverted file text database system of the kind we have been developing [6, 9], the implications of paging are a large increase both in the number of accumulators used to hold the intermediate cosine values and in the length of the inverted file entries. The former means that evaluating a ranking will require more memory; while the latter implies a longer time to resolve queries. Methods for avoiding these increases are outlined in Sections 2.2 and 2.3; a full description is given elsewhere [10].

Finally, there is the impact on effectiveness of the decision to paginate the documents. Experimental results comparing document retrieval with page retrieval are described in Section 2.4.

2.1 Breaking text into pages

One way to store text in a database is as a set of records, each containing a single document. Such an organisation implies that entire documents must be retrieved in response to queries. Moreover, a query can match long documents in which its words are widely separated and could well be unrelated, so this storage strategy can result in irrelevant material being retrieved.

An alternative way of presenting text is as a series of *pages*. There are several advantages to using pages:

- they are of a more manageable size than whole documents, and the size variation between pages can be constrained to be far less than the size variation between documents;
- if a user looks for answers matching a query, it is likely that in relevant material the query's words will occur close together in the retrieved text;
- retrieving a page of text to display may be considerably cheaper than retrieving an entire document;
- in many applications it is natural to regard documents as consisting of parts rather than as a whole; for example, the nodes in a hypertext system or the sections of a book; and
- only parts of documents can, in general, fit on a screen, and people can only comprehend part of a document at a time.

1. Set $prev \leftarrow$ *undefined* and $curr \leftarrow w_1$.
2. For each j from 2 to n,
   (a) If $curr \geq B$, emit $prev$ if it is defined, set $prev \leftarrow curr$, and set $curr \leftarrow w_j$.
   (b) Otherwise, if $w_j > prev$ then set $prev \leftarrow prev + curr$ and set $curr \leftarrow w_j$.
   (c) Otherwise, set $curr \leftarrow curr + w_j$.
3. If $curr < B$ then set $prev \leftarrow prev + curr$ and emit $prev$. Otherwise, emit $prev$ followed by $curr$.

Figure 1: Paging algorithm

For these reasons—and because of the challenge posed by the sheer size of a paged version of the TREC collection—experiments were undertaken with paged text.

The paging strategy adopted first breaks documents into minimal units likely to be useful for ranking. In most collections this unit would probably be the paragraph. The algorithm shown in Figure 1 is then used to gather paragraphs into pages, where w_i is the length of the i'th paragraph of the original document and B is the target length of each constructed page. Documents of fewer than B bytes must, of course, be allocated singleton pages, but all other pages are B bytes or more. Moreover, the aim is that only a small number of pages are significantly longer than B bytes. The algorithm requires time linear in the length of the text and a constant amount of space, and so is a small additional step during database construction. For the experiments described below, length was measured in bytes (the alternatives being words, or some more abstract length value based upon term frequencies), and $B = 1{,}000$ was used.

The pagination of the 2,055 Mb TREC collection resulted in the 742,358 documents being converted into 1,743,848 pages of average size 1,235 bytes each.

2.2 Management of accumulators

An effective way to rank documents against a query is with the cosine measure, defined by the function

$$C_d = \frac{\sum_t w_{q,t} \cdot w_{d,t}}{\sqrt{\sum_t w_{q,t}^2} \cdot \sqrt{\sum_t w_{d,t}^2}} \ ,$$

where q is the query, d is the document, and $w_{x,t}$ is the weight of word t in document or query x. We assume that $W_d = \sqrt{(\sum_t w_{d,t}^2)}$ is the length of document d, and that the top r answers are to be retrieved and returned to the user.

In our experiments term weights were calculated using the IDF rule $w_{d,t} = f_{d,t} \cdot \log(N/f_t)$, where $f_{d,t}$ is the number of appearances of term t in document d, N is the number of documents in the collection, and f_t is the number of documents that contain term t. Harman [3] gives a summary of ranking techniques and a discussion of the cosine measure and IDF weighting rule.

To support the cosine measure in an inverted file text database system, each inverted file entry contains a sequence of $\langle d, f_{d,t} \rangle$ pairs for some term t. The value f_t, the number of documents containing t, is stored in the lexicon, and from these two the cosine contribution $f_{d,t} \cdot (\log(N/f_t))^2$ of term t in document d can be calculated. A structure of accumulators is used to collect these contributions. As the inverted file entry for each query term is processed, the accumulator value for each document number in the inverted file entry is updated by adding in each partial cosine value as it is calculated, so that by the time all inverted file entries have been processed the accumulator for document d contains the value

$$\sum_t f_{d,t} \cdot (\log(N/f_t))^2 = \sum_t w_{q,t} \cdot w_{d,t}.$$

The number of accumulators required to process a query can be large. Queries were created from TREC topics 51–100 by simply extracting and stemming all of the alphanumeric strings, and stopping 601 closed-class words. The generated queries had an average of over 40 terms, and each resulted in about 75% of the pages of TREC having a non-zero accumulator value. With this ratio the most memory-efficient accumulator structure is an array. But at 32 or 64 bits per accumulator, this structure is formidably large, and dominates the retrieval-time memory requirements. Even initialising it is time-consuming.

1. Order the words in the query from highest weight to lowest.
2. Set $A \leftarrow \emptyset$; A is the current set of accumulators.
3. For each word t in the query,
   (a) Retrieve I_t, the inverted file entry for t.
   (b) For each $\langle d, f_{d,t} \rangle$ pair in I_t,
      i. If the accumulator $A_d \in A$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
      ii. Otherwise, set $A \leftarrow A + \{A_d\}$, calculate $A_d \leftarrow w_{q,t} \cdot w_{d,t}$.
   (c) If $|A| \geq L$, go to step 4.
4. For each document d such that $A_d \in A$, calculate $C_d \leftarrow A_d / W_d$.
5. Identify the r highest values of C_d using a heap.

Figure 2: *Quit* algorithm for computing cosine using approximately L accumulators

A simple strategy for restricting the number of accumulators is to order query terms by decreasing weight, and only process terms until some designated termination condition is met. One such condition is to impose an *a priori* bound L on the number of non-zero accumulators, and to stop processing terms when this bound is reached. Such a ranking process, which we call *quit*, is illustrated in Figure 2. Other possible termination conditions would be to place a limit on the number of terms considered or on the total number of pointers decoded; or to place an upper bound on the term frequency f_t, and only process terms that appear in fewer than x% of the documents, for some predetermined value x. Buckley and Lewit [1]; Lucarella [8]; and Wong and Lee [13] have also considered various stopping rules, and derive conditions under which inverted file entries can be discarded without affecting the r top documents (although their order within the ranking may be different). Here we are prepared to allow approximate as well as exact rules, acknowledging that the cosine measure is itself a heuristic.

Quitting has the advantage of only processing the inverted file entries of a subset of the query terms, and hence of faster ranking, but at the possible expense of poor retrieval performance, depending upon how discriminating the low weighted terms are.

An alternative is to continue the processing of

1. Order the words in the query from highest weight to lowest.
2. Set $A \leftarrow \emptyset$.
3. For each word t in the query,
    (a) Retrieve I_t.
    (b) For each $\langle d, f_{d,t} \rangle$ pair in I_t,
        i. If $A_d \in A$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
        ii. Otherwise, set $A \leftarrow A + \{A_d\}$, calculate $A_d \leftarrow w_{q,t} \cdot w_{d,t}$.
    (c) If $|A| \geq L$, go to step 4.
4. For each remaining word t in the query,
    (a) Retrieve I_t.
    (b) For each d such that $A_d \in A$, if $\langle d, f_{d,t} \rangle \in I_t$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
5. For each document d such that $A_d \in A$, calculate $C_d \leftarrow A_d / W_d$.
6. Identify the r highest values of C_d.

Figure 3: *Continue* algorithm for using approximately L accumulators

inverted file entries after the bound on the number of accumulators is reached, but allow no new documents into the accumulator set. This *continue* algorithm is illustrated in Figure 3. The algorithm has two distinct phases. In the first phase, accumulators are added freely, as in the *quit* algorithm. In the second stage, existing accumulator values are updated but no new accumulators are added. Both *quit* and *continue* generate the same set of approximately L candidate answers, but in a different permutation, so when the top r documents are extracted from this set and returned, different retrieval effectiveness can be expected, particularly if $r \ll L$.

A similar strategy to that of *continue* is described by Harman and Candela [3, 4], although their motivation is somewhat different—they desire a small number of accumulators to reduce the sorting time required for a total ranking of the collection, whereas we assume that only a small fraction of the collection is to be presented. In this latter case, to find the top r documents a heap is the appropriate data structure, and $O(N + r \log N)$ time is required, a small fraction of query processing time.

Figure 4 shows retrieval effectiveness, measured as an 11pt average precision, as a function of $k = |A|$, the number of accumulators actually used. The small numbers beside the *continue* curve show the average number of terms processed in phase one of each query. For example, only 8.2 terms are needed to generate 27,000 accumulators. The difference between *quit* and *continue* is marked, and, perhaps surprisingly, even the mid to low weight terms appear to contribute to the effectiveness of the cosine rule. Ignoring them leads to significantly poorer retrieval.

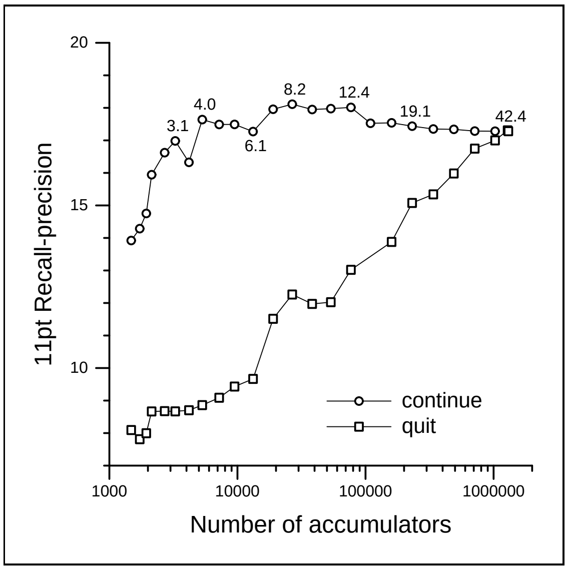

Figure 4: Effectiveness of *quit* and *continue*

In these experiments direct assessment of the relevance of pages was not possible because the relevance judgements were for whole documents. To measure effectiveness, it was assumed that if a document was relevant then each page from it was relevant, but only the most highly ranked page from each document was returned and counted. We believe this method to be fair, since one way in which pages might be used would be to rank on pages but return whole documents, with perhaps some highlighting of the top-ranked pages. Other strategies for using parts of documents to select whole documents are discussed in Section 3.

Also surprising is that the *continue* strategy, with restricted numbers of accumulators, is capable of *better* retrieval performance than the original cosine method, in which all pages are permitted accumulators. It would appear that the mid to low weight terms, while contributing to retrieval effectiveness, should not be permitted to collaborate and select documents that contain none of the more highly weighted terms.

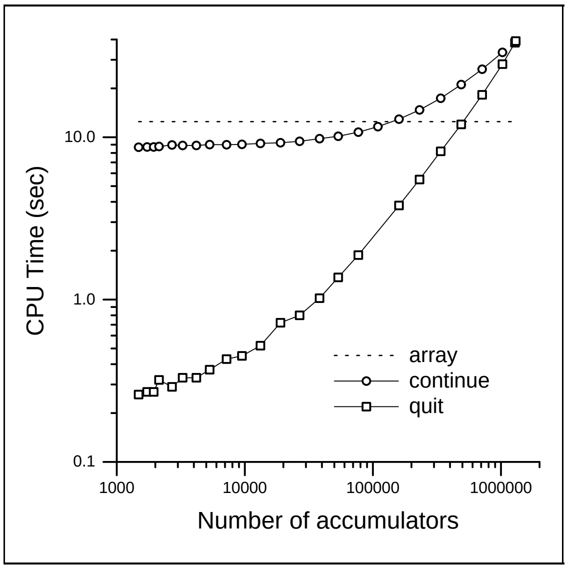

Figure 5: CPU time of *quit* and *continue*

Whilst the results for retrieval effectiveness are encouraging, the results for time are not. Figure 5 shows the cost of these two strategies in terms of cpu time, with the set A of accumulators stored as a hash table. The times shown include all processing required to identify the top $r = 200$ ranked documents, but do not include the cost of actually retrieving those documents, which, in a system such as ours, in which the text is also compressed, involves further decoding effort. All timings are for a lightly loaded Sun SPARCstation Model 512 using a single processor; programs were written in C and compiled using gcc. While for values of L less than about 100,000 the *continue* method takes no longer than the simple cosine measure when implemented using an array of accumulators, it is clearly unsatisfactory compared with the *quit* technique. In the next section we discuss methods by which this situation can be improved.

2.3 Processing long inverted file entries

Compression can reduce index size by a factor of six; for example, for the paged form of TREC the size reduces from 1,120 Mb to 184 Mb, an irresistible saving. Compression, however, prohibits random access into inverted file entries, so that the whole of each inverted file entry must be decoded, even though not every $\langle d, f_{d,t} \rangle$ pair is required. This is the reason that the *continue* strategy is slow.

The need to decompress an inverted file entry in full can be avoided by including a series of synchronisation points at which decoding can commence [10]. These can be arranged as a series of pointers, or *skips*, as illustrated in Figure 6.

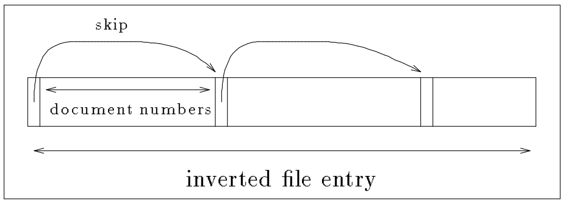

Figure 6: Adding skips

The skips divide the inverted file entry into a series of blocks, and to access a number in the entry, only the skips and the block containing the number need to be decoded. Appropriately coded, such skips increase the size of the compressed inverted file by only a few percent, but can drastically reduce the amount of decoding required. Decode time for a skipped inverted file entry is given by $T = t_d\,(2s + Lp/2s)$, where t_d is the time needed to decode one $\langle d, f_{d,t} \rangle$ pair, p is the number of such pairs in the inverted file entry, L is the number of accumulators, and s is the number of skips. This time is minimised at $s = \sqrt{Lp}/2$. For 10,000 accumulators and an inverted file entry of length 100,000 (a common figure for queries to the paged TREC), total time including reading from disk on a typical system drops from 0.30 seconds to 0.22 seconds [10]. Moreover, under the same assumptions an uncompressed inverted file entry of this length would take 0.30 seconds to read, and so the skipped compressed inverted file provides faster ranking than even an uncompressed inverted file.

To test skipping, inverted files were constructed for several different fixed values of L and then, for each inverted file, the cpu time for inverted file processing measured for a range of actual L values. To ensure that inverted files did not become too large, a minimum block size was imposed, requiring that every skip cover at least four $\langle d, f_{d,t} \rangle$ pairs. The results of these experiments are shown in Figure 7. As can be seen, substantially faster query processing is the result when the number of accumulators is less than about 100,000. As predicted by the analysis, the index constructed assuming that $L = 1{,}000$ gives the best performance when the number of accumulators (the variable L in Figure 3) is small.

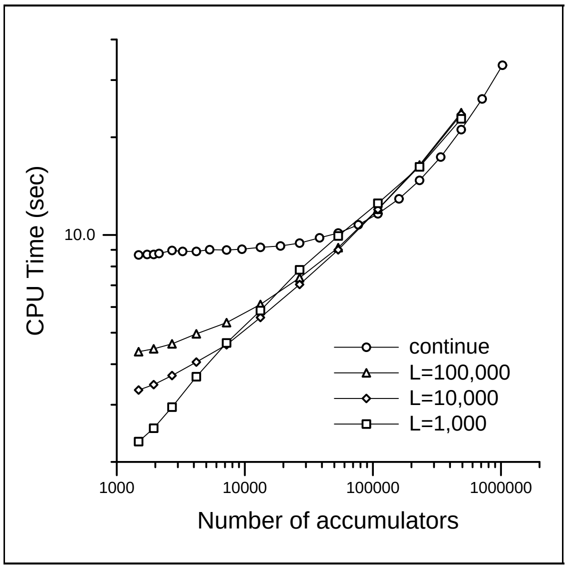

Figure 7: Time required by skipping

2.4 Retrieval experiments

The implementation techniques described above, skipping and limitation of the number of accumulators, can be applied to document retrieval as well as to page retrieval. Table 1 summarises the various resources required and performance achieved by both document and page retrieval, for various values of L, the nominal number of accumulators. The following points should be noted in connection with the data in Table 1.

File sizes Stemming was applied during inverted file construction, but no words were stopped. All alphanumeric strings were indexed, totalling 333,000,000 term occurrences of 538,244 distinct terms. The document index contained 136,000,000 $\langle d, f_{d,t} \rangle$ pointers, the paged index 196,000,000 such pairs. Each is compressed to occupy less than one byte. The variation in inverted file size is due to the insertion of skips, or, in the case of the "All" row, their absence.

The text itself was also stored compressed using a word-based model [9], allowing the 2,055 Mb to be reduced to 605 Mb. A further 27 Mb of smaller auxiliary files were also required. In total, the complete retrieval system for the paged TREC occupied about 817 Mb, or 40% of the initial unindexed text.

Retrieval time The SPARCstation 512 used for these experiments is approximately 2.2 times faster than the SPARCstation 2 used for the first round of experimentation [6, 9]. The times listed cover all activity from issue of query until a ranked list of 200 document numbers is calculated.

Fetching of the 200 answers, which occupy 226.9 Kb compressed, adds a uniform 1.9 sec per query including the cost of decompression. Decompressed, there is an average of 829.9 Kb of output text per query.

Elapsed times during the ranking process are generally about 1.6 sec greater than cpu time, in the course of which an average of 42 disk accesses are made to the lexicon; 42 disk accesses made to the inverted file itself (fetching a total of 1.65 Mb of data, containing about two million compressed $\langle d, f_{d,t} \rangle$ pairs); and 275 accesses to the combined file of document weights and addresses. All of these files except the inverted file are small, and likely to have been buffered into main memory, hence the small overhead.

Elapsed time for the presentation of documents was about 5.0 sec per query greater than cpu time, caused by the need to perform a further 200 seeks into the file containing the compressed text. This file is too large for there to have been any buffering effect.

Memory space When skipping is employed (the first three rows in each section of the table) memory space during query processing is proportional to the number of non-zero accumulators. In these experiments a hash table was used, and an average of 14 bytes per accumulator required. For the "All" experiments an array of accumulators was used, 2.8 Mb for document retrieval, and 6.6 Mb for page retrieval.

An array of 6-bit approximate document lengths was used to guide the retrieval process [11], requiring 0.5 Mb for document retrieval and 1.2 Mb for page retrieval.

Terms processed The values for "Terms processed" indicate the average number of terms processed before processing switched from the first to the second phase of *continue*. For each query a whole number of inverted file entries were processed, and this is why the "non-zero accumulators" average is greater than the target value L. Not surprisingly, with page retrieval fewer terms were re-

	Inverted file (Mb)		Retrieval Time (cpu sec)		Non-zero accumulators	
	Doc.	Page	Doc.	Page	Doc.	Page
$L = 1{,}000$	145.8	205.4	2.36	2.95	2,330	2,701
$L = 10{,}000$	156.7	220.3	4.46	5.57	12,855	13,238
$L = 100{,}000$	161.6	230.2	9.80	12.00	107,737	109,370
All (array)	128.6	184.4	7.66	12.48	582,783	1,307,115

(a) Resources required

	Terms processed		Precision at 200		Pessimal 11pt average	
	Doc.	Page	Doc.	Page	Doc.	Page
$L = 1{,}000$	3.08	2.84	0.271–0.612	0.260–0.639	0.164	0.166
$L = 10{,}000$	6.80	6.06	0.346–0.530	0.319–0.554	0.185	0.173
$L = 100{,}000$	16.84	14.26	0.333–0.446	0.326–0.504	0.175	0.175
All (array)	42.44	42.44	0.331–0.444	0.321–0.502	0.174	0.173

(b) Retrieval effectiveness

Table 1: Document vs. paragraph retrieval: (a) resources required, and (b) retrieval effectiveness

quired, on average, to consume the available accumulators.

Precision at 200 The range of values in this column is to show the fraction of unjudged documents that were accessed. The lower number assumes that all unjudged documents are irrelevant; the upper is calculated assuming that all are relevant.

11pt effectiveness The "11pt average" values are calculated based solely upon the top 200 ranked documents, assuming that all remaining relevant documents are ranked last, and that all unjudged documents are not relevant.

Index construction New algorithms have been developed for index construction. Using these algorithms, the index was built in under 4 hours, using a peak of 40 Mb of main memory and less than 50 Mb of temporary disk space above and beyond the final size of the inverted file. The compression of the documents took a further 4 hours, for a total database build time of under 8 hours.

Based upon these experiments we conclude that:

- use of a limited number of accumulators does not appear to impact retrieval effectiveness, and so is an extremely attractive heuristic for ranking large collections because of the dramatic savings in retrieval-time memory usage that result;
- introduction of skips to the compressed inverted file entries significantly reduces processing time in this restricted-accumulators environment;
- index compression can, in this way, become "free": if only partial decoding is required, the input time saved by compression can be more than enough to pay the cpu cost of fractional decoding;
- all non-stopped terms should be allowed to contribute to the ranking, and that the *quit* strategy is inferior to *continue*; and
- even relatively simple pagination gives retrieval not measurably inferior to document retrieval.

When dealing with pages, the retrieval system was implemented to display the entire document, but highlight the page or pages that had caused the document to be ranked highly. This decision made the paged collection particularly easy to use, and when we browsed the answers to the queries it was very satisfying to be able to find the exact text that had triggered the match. This advantage of pagination became clear even in the early stages of the project, while we were attempting to debug the implementation of the cosine method.

On the other hand, a problem that was brought out by these experiments was the difficulty of comparing retrieval effectiveness in the face of non-exhaustive relevance judgements. When precision rates are around 30%, and a further (in the $L = 1{,}000$ case) 30% of documents are unjudged, there can be no significance whatsoever attached to the difference between even 30% precision and 40% precision. Indeed, assuming that 27.1% of the unjudged documents are relevant for the "$L = 1{,}000$; Doc" combination gives a final precision of 0.411; the corresponding number for the "All; Doc" pairing is only 0.373. Thus, the precision figures of Table 1 are sufficiently imprecise that no conclusion can be drawn about the appropriate value of L that should be used, and about the merits of document vs. paged retrieval. There is clearly scope for research into other methodologies for comparing retrieval mechanisms.

3 Structured documents

Many of the documents in the TREC collection are very large and have explicit structure, and it may be possible to use this structure—rather than the statistically based pagination methods described above—to break documents into parts. In particular, many documents can be broken up into a set of sections, each section having a type. There has been relatively little work done on retrieving or ranking partial documents. However, Salton et al. [12] have demonstrated that document structure can be valuable. Sometimes this structure is explicitly available [2], and sometimes it has to be discovered [5], but the knowledge of this structure has been shown to help determine the relevance of sub-documents. In this part of the work we used a small database to investigate whether retrieval of sections helped document retrieval, and whether retrieval of documents helped section retrieval. By way of a benchmark, the paged retrieval techniques described earlier were applied to the same database.

3.1 The database

Since we needed information about the relevance of sections to queries it was not possible to use the full TREC database. Instead, we used a database consisting of 4,000 documents extracted from the Federal Register collection. These documents were selected as being the 2,000 largest documents which were relevant to at least one of topics 51–100 provided for the first TREC experiment. Another 2,000 documents were randomly selected from the Federal Register collection to provide both smaller documents and non-relevant documents. The average number of words in these documents was 3,260.

These documents were then split into sections based on their internal markup. The documents had a number of tags inserted that defined an internal structure. It appeared that only the T2 and T3 tags could be reliably used to indicate a new internal fragment. Section breaks were defined to be a blank line, or a line containing only markup, followed by a T2 or a T3 tag. This led to a database of 32,737 sections. Each of these sections had a type based on its tag. The types were ⟨purpose⟩, ⟨abstract⟩, ⟨start⟩, ⟨summary⟩, ⟨title⟩, ⟨supplementary⟩, and a general category ⟨misc⟩ that included all remaining categories.

Having made the document selections, only 19 of the queries 51–100 had a relevant document in the collection. Each of the sections for documents that had been judged as relevant was judged for relevance against these queries so that finer grained retrieval experiments were possible. One difficulty that arose was that quite a few documents that had been judged relevant appeared to have no relevant sections—there were relevant key terms in the documents but the documents themselves did not appear to address the information requirement. There were 145 such (query, document) pairs. To be consistent, we took these document to be irrelevant. After these alterations, only 14 queries had a relevant section, and there were an average of 23 relevant sections per query.

3.2 Structured ranking

We carried out a set of experiments on ranking documents using the retrieval of sections. We first compared simple ranking of documents against ranking sections to find relevant documents. Next, a set of formulae were devised that attempted to use the fact that one document has several sections that might be more or less highly ranked. These took into consideration the rank of the section, the number of ranked sections, and the number of sections in the document. Experiment 3 describes one of the more successful formulas.

Further trials were then performed using the type of the section. First, a set of experiments were run that determined which types were better predictors of relevance. These results were then used to devise a measure that used a weight for each type. Finally, we tried to combine these results to see if it was helpful to use the rank of the whole documents along with the rank of its component parts.

Experiment 1: Rank full documents against the queries using standard cosine measure.

Experiment 2: Split documents into sections. Measure similarity of each section against the queries using standard cosine measure. Order documents based on the highest ranked section.

Experiment 3: Split documents into sections. Measure similarity of each section against the queries using standard cosine measure. Order documents based on

$$\sum 0.5^{n-1} wt(s_n)$$

where s_n is the n^{th} highest ranked section in the document. The effect of this formula is that a document's weight is determined by a decay formula using the weight of all of a document's sections. (Values other than 0.5 were tried, but gave poorer performance.)

Experiment 4: Split documents into sections. Measure similarity of each section against the queries using standard cosine measure. Then weight each section using its type, for example $\langle$introduction$\rangle$ or $\langle$address$\rangle$. Order documents based on

$$\sum tw(type(s_n)) 0.5^{n-1} wt(s_n)$$

where s_n is the n^{th} highest ranked section in the document, $type(s_n)$ is the type of the section, and $tw(t_i)$ is the weight of the type t_i.

The weights of the types of sections were obtained by conducting a set of experiments where all types but one were given a weight of 1, and in turn, each type was given a weight of 2. Using these experiments it was determined that $\langle$purpose$\rangle$ and $\langle$summary$\rangle$ were each more helpful, and that $\langle$misc$\rangle$ was less helpful. As a result, in this experiment $tw(\langle\text{purpose}\rangle) = tw(\langle\text{summary}\rangle) = 2$, $tw(\langle\text{misc}\rangle) = 0.5$, and other weights were set to 1.

Experiment 5: Rank the documents, and rank the sections. Form a new rank based on the average rank of these two ranks.

Other experiments were carried out using best two sections, and formulas that more closely approximated the cosine measure. None of these experiments achieved better results than the ones displayed here. The obvious conclusion is that if documents are available for ranking as whole documents, then for this collection it is preferable to do so.

3.3 Section retrieval

For very long documents it may be desirable to return relevant sections rather than relevant documents. We were interested to see whether it might be useful to know about the rank of the containing document. In the first experiment documents were ranked, and sections shown in document order. This produced very poor results. Next, we still ranked sections in higher ranked documents ahead of lower ranked documents, but used section ranking for sections in the same document. This was reasonable but there were still many irrelevant sections being examined. Finally, we attempted to delete these irrelevant sections by using document ranking, and then section ranking, but this time discarding sections that had a section rank of greater than 200.

Experiment 6: Rank sections against the queries using standard cosine measure.

Experiment 7: Rank full documents against the queries using standard cosine measure. Order the sections by their appearance within documents.

Experiment 8: Rank full documents against the queries using standard cosine measure. Order sections, first by document, then by rank within documents.

Experiment 9: As in experiment 3, but then delete all but the 200 highest ranked sections.

These experiments show that ranking both documents and sections does help to find more relevant sections. This result is in contrast to the earlier investigation of finding relevant documents. We are able to find relevant sections much easier if the rank of both the sections, and the containing documents are taken into account.

3.4 Paged versus section retrieval

Our results showing that retrieving documents based on section ranking was not as useful as

Documents / Experiment	5	10	15	20	25	30	50	200
1	0.286	0.271	0.248	0.236	0.234	0.229	0.206	0.102
2	0.243	0.221	0.214	0.204	0.191	0.178	0.170	0.092
3	0.271	0.250	0.229	0.221	0.206	0.202	0.184	0.094
4	0.329	0.257	0.233	0.229	0.206	0.202	0.184	0.085
5	0.343	0.264	0.238	0.236	0.234	0.231	0.209	0.099

Table 2: Comparison of ranking formula for fixed number of documents returned

Documents / Experiment	5	10	15	20	25	30	50	200
6	0.186	0.164	0.181	0.161	0.160	0.152	0.140	0.120
7	0.100	0.121	0.105	0.100	0.094	0.088	0.090	0.083
8	0.171	0.121	0.114	0.100	0.091	0.083	0.100	0.082
9	0.214	0.171	0.186	0.189	0.189	0.193	0.179	NA

Table 3: Comparison of ranking formula for fixed number of sections returned

document ranking was perhaps surprising—other studies have indicated that considering smaller fragments was helpful, although in combination with larger contexts. We wondered whether the section boundaries that had been imposed were inappropriate. We thus applied the pagination techniques described in Section 2.1 to the 4,000 document database used here. These results are shown in Table 4 as Experiment 10.

These three experiments are a little difficult to interpret. Dividing documents into sections leads to poorer retrieval performance, but dividing further into pages leads to comparable retrieval performance to ranking whole documents. It may be that the manually supplied divisions are poorer than the divisions generated by automatic techniques [5]. Experiments 2–4 show that some of this performance degradation can be ameliorated by taking documents' structure into account. However, these experiments indicate that there is no retrieval advantage in breaking the document up, should the desired unit of retrieval be whole documents.

4 Conclusions

The combination of a restricted-accumulators policy and the introduction of skips to the compressed inverted file entries allows fast query evaluation on large text collections. Moreover, the ranking can be carried out within modest amounts of main memory. For example, the paged TREC collection contains 1.7 million pages, but ranked queries of 50 or more terms can be resolved within seconds using just a few megabytes of main memory. These two techniques mean that large collections can be searched on small machines without measurable degradation in retrieval effectiveness.

In the second part of the experiment we have concentrated on large documents, breaking them into smaller units for the purposes of indexing. It is not clear that users are interested in retrieving 3 Mb documents, and these experiments were designed to allow users the option of retrieving smaller parts of such documents. The results were mixed. It appears that indexing both sections and documents is helpful in ranking sections. However, it is not clear what an appropriate indexing strategy is if only full documents are to be returned. We are continuing our investigation of partial document retrieval.

Acknowledgements

We would like to thank Daniel Lam and Neil Sharman for their assistance with various components of the experiments described here. This work was supported by the Australian Research Council, the Collaborative Information Technology Research Institute, and the Centre for Intelligent Decision Systems.

Documents / Experiment	5	10	15	20	25	30	50	200
1	0.286	0.271	0.248	0.236	0.234	0.229	0.206	0.102
2	0.243	0.221	0.214	0.204	0.191	0.178	0.170	0.092
10	0.314	0.271	0.238	0.236	0.223	0.224	0.193	0.096

Table 4: Comparison of pages, sections and whole document ranking

References

[1] C. Buckley and A.F. Lewit. Optimization of inverted vector searches. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 97–110, Montreal, Canada, June 1985. ACM Press, New York.

[2] M. Fuller, E. Mackie, R. Sacks-Davis, and R. Wilkinson. Coherent answers for a large structured document collection. In Korfhage et al. [7], pages 204–213.

[3] D. Harman. Ranking algorithms. In W.B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 14, pages 363–392. Prentice-Hall, 1992.

[4] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581–589, 1990.

[5] M.A. Hearst and C. Plaunt. Subtopic structuring for full-length document access. In Korfhage et al. [7], pages 59–68.

[6] A. Kent, A. Moffat, R. Sacks-Davis, R. Wilkinson, and J. Zobel. Compression, fast indexing, and structured queries on a gigabyte of text. In D.K. Harman, editor, *Proc. First Text REtrieval Conference (TREC-1)*, pages 229–243, Gaithersburg, Maryland, November 1992. National Institute of Standards.

[7] R. Korfhage, E. Rasmussen, and P. Willett, editors. *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, Pittsburg, June 1993. ACM Press, New York.

[8] D. Lucarella. A document retrieval system based upon nearest neighbour searching. *Journal of Information Science*, 14:25–33, 1988.

[9] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 1993. To appear.

[10] A. Moffat and J. Zobel. Fast ranking in limited space. In *Proc. IEEE International Conference on Data Engineering*, Houston, Texas, February 1994. To appear.

[11] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Information Processing & Management*. To appear.

[12] G. Salton, J. Allen, and C. Buckley. Approaches to passage retrieval in full text information systems. In Korfhage et al. [7], pages 49–58.

[13] W.Y.P. Wong and D.K. Lee. Implementations of partial document ranking using inverted files. *Information Processing & Management*, 29(5):647–669, 1993.