Efficient Passage Ranking for Document Databases

Marcin Kaszkiel Justin Zobel Ron Sacks-Davis
RMIT University, Melbourne, Australia

Abstract

Queries to text collections are resolved by ranking the documents in the collection and returning the highest-scoring documents to the user. An alternative retrieval method is to rank passages, that is, short fragments of documents, a strategy that can improve effectiveness and identify relevant material in documents that are too large for users to consider as a whole. However, ranking of passages can considerably increase retrieval costs. In this paper we explore alternative query evaluation techniques, and develop new techniques for evaluating queries on passages. We show experimentally that, appropriately implemented, effective passage retrieval is practical in limited memory on a desktop machine. Compared to passage ranking with adaptations of current document ranking algorithms, our new "DO-TOS" passage ranking algorithm requires only a fraction of the resources, at the cost of a small loss of effectiveness.

1 Introduction

Techniques for retrieval of documents from large text collections are well developed, and in current systems typical queries can be resolved in a fraction of a second. These techniques are used in many applications, ranging from queries of one or two words posed to Internet search engines by naive users to complex queries posed to special-purpose databases by information search specialists. In current systems the use of traditional Boolean queries has been substantially replaced by the use of ranked queries, in which each document in the collection is heuristically assigned a score representing its similarity to the query and the most similar documents are returned to the user. To compute these heuristics many similarity measures have been developed, the best of which have been shown in large-scale experiments to be effective in practice at identifying documents that satisfy users' needs [27, 37].

With the advent of intranets, the Internet, and online document authoring systems, there is rapid growth in both the number of users accessing text databases and in the volume of stored text. This pressure has led to the development of new, efficient algorithms for indexing and ranked query evaluation [20, 23, 35] that allow queries to be resolved much more rapidly and with fewer resources than with the best techniques of only a few years ago. These techniques include the use of compression, to reduce the size of index and text; heuristics that drastically reduce the number of documents to be considered as candidate answers to each query; and restructuring of the index to reduce the volume of index information processed as part of query evaluation.

Another pressure on document retrieval system technology is change in the kinds of documents that are being stored. Traditional text retrieval systems, such as those used in libraries, indexed only limited documents such as abstracts of papers. Current systems are used for indexing a great variety of documents, ranging, for example, from short documents such as abstracts, newspaper articles, and web pages to long documents such as journal articles, books, court transcripts, and legislation. In some cases individual documents can be megabytes or tens of megabytes long. For example, an Australian database of legislation includes one document of 53 Mb and another of 23 Mb; most of the principal sections of the latter document are tens or hundreds of kilobytes, but one section is 14 Mb. For such documents, standard ranking is not of value.

These applications present a serious challenge to document retrieval systems. The heuristics used for evaluating similarity of document and query are not always reliable when documents of

widely varying length are involved, and erroneously fetching a long document carries a significant performance penalty. Also, these heuristics are simply less effective over long documents, because it is difficult to weight for proximity of the query terms within documents. Moreover, in modern systems it is not always clear what to retrieve. In traditional text retrieval systems, the stored data was divided into individual documents, such as abstracts or articles, used as units of retrieval. In contrast, in some cases there is no clear division of the stored data: for documents marked up in a hierarchical language such as SGML, different applications may require different components of the text; the logical divisions may be unhelpful, as in the case of court transcripts where each session yields a single document; or the divisions or markup used in one document may be absent in another.

A technique that addresses these problems is *passage retrieval*, in which the unit of retrieval is blocks of text from the stored documents. Passages can be based on logical components of documents such as sections or paragraphs, but earlier work, in particular our own experiments, shows that the most effective and reliable form of passage is a fixed-length sequence of words occurring anywhere in the document [7, 17]. In this model of query evaluation, each document is (in principle) regarded as containing a large number of passages, with potentially each word occurrence being the start of a fresh passage. Query evaluation proceeds by identifying the passages in the database that are the most similar to the query. Then either the documents containing the highest-ranked passages are returned to the user, or the passage is returned together with context information such as the title of the document and information about the location of the passage within the document's structure. The difficulty with using passages is that they can greatly increase the cost of query evaluation.

In this paper we begin by reviewing the principal techniques for evaluation of ranked queries, including: *document-ordered* or DO processing (also known as document-at-a-time processing, in which the inverted lists are processed simultaneously); *term-ordered* or TO processing (also known as term-at-a-time processing, in which each inverted list is processed in sequence); and some of the more successful optimisations to term-ordered query evaluation, in particular *skipping* or TOS strategies based on limiting the number of candidate documents considered during query evaluation. For ranking of whole documents, a previous analysis has suggested that in limited memory DO evaluation is more efficient than TO [30]. Using the MG prototype text database system we compare TO, TOS, and DO experimentally, and show that for short queries DO is more efficient than TO, but for other queries—and even in circumstances that are ideal for DO evaluation—TO evaluation is faster, and that as query length increases the cost of DO evaluation rises much more rapidly than that of TO evaluation. For document ranking, both of these strategies are slower than TOS evaluation.

We then examine the suitability of each of these techniques for passage retrieval and propose a range of options for passage retrieval in practice. We experimentally compare these techniques, considering both effectiveness and efficiency. These experiments show that passage ranking is practical, with short queries to a large collection evaluated in limited memory in one or two seconds. With sufficient memory, TOS and DO evaluation are both effective, and that for the most common retrieval task—fetching a list of ten or twenty documents—the memory requirements need not be excessive. The experiments also show that, as for whole-document ranking, as query length grows the cost of DO query evaluation rises much more rapidly than that of TOS query evaluation.

For shorter queries on passages, however, DO evaluation is the more efficient strategy. We propose a combined DO-TOS strategy, where DO processing is used to determine a set of likely documents based on the rarest of the query terms and TOS processing is used to reorder this set, and show that this approach is often more efficient than either strategy used alone. For passage retrieval, DO evaluation allows heuristic simplification of some of the in-document computation; the combined approach is not likely to yield significant gains for document retrieval.

2 Text retrieval

In statistically-based retrieval systems, documents are ranked according to a heuristic similarity function that estimates the degree of similarity of document and query. In principle, evaluation proceeds by computing the similarity of each document to the query, then returning the k documents with the highest similarity values. Many similarity functions have been proposed; a typical, effective formulation is the cosine measure [11, 24, 37]:

$$C(q,d) \;=\; \frac{\sum_{t \in q \wedge d}(w_{q,t} \cdot w_{d,t})}{W_d}$$

where q is a query, d is a document,

$$\begin{aligned} W_d &= \sqrt{\sum_{t \in d} w_{d,t}^2}\,, \\ w_{d,t} &= \log_e(f_{d,t}+1)\,, \\ w_{q,t} &= \log_e(f_{q,t}+1) \cdot \log_e(N/f_t+1)\,, \end{aligned}$$

the value $f_{x,t}$ is the number of occurrences or *frequency* of term t in x; there are N documents; f_t is the number of distinct documents containing t; and the expression $\log_e(N/f_t+1)$ is the "inverse document frequency", a representation of the rareness of t in the collection. Compared to cosine formulations with refinements such as pivoted document length normalisation [28], we have found that, on the test data used in this paper, the formulation above achieves about the same effectiveness.

The cosine measure exemplifies typical features of the more effective similarity functions. It gives high weight to documents that contain many of the query terms; these weights are increased if the terms are common in the document; and terms that are relatively rare in the collection attract a higher weight. To counteract the statistical tendency of longer documents to contain more terms (and thus have higher weight), document length is used for normalisation. This particular formulation has the practical advantage that document length is independent of collection-wide statistics.

The performance of similarity measures can be improved in several ways. One is to use phrases as well as individual terms; rather than simply retrieving documents containing both "oil" and "well", for example, the system could favour documents in which these terms are adjacent. More generally, documents in which the query terms are proximate are, intuitively, more likely to be relevant than documents in which they are widely spaced.

Another kind of improvement is the use of relevance feedback, where users are presented with a small number of answers and indicate which are relevant; the system can then heuristically choose further terms from these documents and issue an augmented query. Intuitively, this technique is successful because the same concept can be expressed in different ways, and relevant documents are likely to contain new expressions of the concepts of interest to the user. An alternative intuition is that the user's query is an initial link into a text collection; the documents thus located can be used to link to further material on the same topic. Perhaps surprisingly, the user-feedback element of relevance feedback can be omitted: since, statistically, a reasonable proportion of highly ranked documents are relevant, these can be used to expand the query without interaction with the user. Experiments have consistently shown that query expansion leads to improved performance [1, 18, 36]. In practice, however, query expansion has a significant disadvantage: it greatly increases the number of query terms, thus increasing the costs of query evaluation. A query evaluation technique that was efficient for small queries only—such as the two-word to five-word queries typically presented to internet search engines—would not be satisfactory.

Passage retrieval

An alternative approach to text retrieval is the use of passages, or small subparts of documents. Ranking can proceed by either fetching documents according to the best passage they contain,

or by actually fetching passages (and any context necessary to the user's understanding of the passage, such as the title and structure of the containing document). Despite the fact that retrieval based on passages considers only part of each document, it has several significant advantages: in passages, query terms must be proximate; normalisation problems are avoided if passages are of similar lengths; passages are pointers to relevant parts of long documents, which may also contain much irrelevant material; and passages are convenient for presentation to the user.

Passages can be used in different ways. One approach is to return only the highly ranked passages, providing a concise response with a limited number of bytes to transport. Such passage retrieval may, for example, provide a good basis for a question-and-answer style of information retrieval. Another approach, as in the experiments in this paper, is to use passages as proxies for documents; that is, documents are ranked according to similarities computed for their passages. In this approach, documents are still returned in response to queries, providing context for the passages identified as answers. (In the context of research this approach has the pragmatic advantage that the effectiveness of passage-based retrieval can be measured with document-level relevance judgements.) However, these issues are independent of the merits of passage retrieval as a query evaluation mechanism.

Many definitions of passage have been proposed in the literature, including passages based on: document markup [8, 10, 25, 26, 32, 33, 39] such as sections, paragraphs, and groups of sentences; boundaries at which topic shifts can be inferred [16, 19]; sequences of paragraphs of similar aggregate length [39]; and fixed-length sequences of words [7, 16, 17, 29], which can be either disjoint or overlapping. Several of these definitions of passage rely on semantic properties such as sentence boundaries, and intuitively it is reasonable to suppose that passages based on natural blocks of text should provide the basis for retrieval. However, in earlier work by ourselves [17], in which we compared many of these techniques over the same data, we observed that the techniques that made less use of semantics were in general the most effective. Confirming our suppositions and the results of Callan [7], these results showed that the greatest retrieval effectiveness is achieved with passages of fixed-length sequences of words that can start at any point in a document. Our results strongly suggest that taking into account even elementary structure such as paragraph or sentence boundaries degrades effectiveness; use of such structure assumes that it can be reliably identified, and reintroduces problems such as length normalisation.

For the FR subcollection of TREC [14], which contains documents of greatly varying length, we showed that use of overlapping fixed-length passages of 150 to 300 words markedly improves retrieval effectiveness, by up to 37% over full document ranking [17]. Smaller but consistent improvements were observed for the full TREC collection, and have since been confirmed by our contributions to the 1997 and 1998 rounds of TREC [12, 13]: as a first-stage retrieval mechanism, these results and those of Walker et al. [32] show that passages are superior to the current best whole-document similarity measures. Our results showed that the highest effectiveness is achieved when passages are allowed to start at any point in the text, but if they are restricted to commencing at, say, 25 word intervals, the degradation is small. This method is similar to the approach described by Stanfill and Waltz [29], in which documents are segmented into short sections and the best sections in each document are used to rank documents. Up to two adjacent sections can be merged by summing their absolute similarity to the query. However, in our experiments we observed that use of disjoint passages, or passages with only limited overlap, significantly degrades effectiveness.

Similar performance has been achieved for short queries with an approach based on markup. Clarke et al. [8] use Boolean queries to identify segments of documents that satisfy the Boolean condition. Document similarity is based on the shortest possible segment that matched the Boolean query, scoring text segments on their absolute length. No collection statistics are required, but, in the initial version of this work, manual effort was required to generate the Boolean queries and all query terms are treated equivalently. An approach that avoids the manual effort was used in the 1998 round of TREC [10]. A query expressed in natural language is used to generate a set of Boolean queries ordered by decreasing number of terms: the first is the conjunction of all query words; the next query is a conjunction of all query words except the word with highest IDF; and so on until the last query is a disjunction of all query words. Queries are matched in turn against

text segments, and documents are ranked by query length and shortness of matching segment. While it appears that this approach is only effective if the queries are short [9], it is still under development.

In our experiments, the similarity assigned to a document was that of its highest-ranked passage, but there are alternative strategies. Hearst and Plaunt [16] have shown that combining similarities from the best passages in a document can be more effective. We have not explored this possibility, having already achieved good gains in effectiveness and because it is not obvious how scores from overlapping passages might be combined.

Passages do have one serious drawback: naively implemented, the cost of ranking passages is high. The number of candidate passages in a collection is much larger than the number of candidate documents—particularly for highly overlapping passages—so ranking is more expensive, and potentially impractical. However, as discussed above in some circumstances there is no logical division of text into documents, and for collections of long documents passages provide substantially better effectiveness than the alternatives. For these reasons it is worth exploring options for efficient passage retrieval.

Measuring text retrieval systems

Text retrieval systems can be measured in two ways: by their efficiency, that is, their use of resources such as disk, memory, and time; and by their effectiveness, that is, their ability to retrieve answers that satisfy users' information needs. Judgement of effectiveness requires three components. First is a standard text collection containing a reasonably large number of documents. Second is a standard set of queries. Third is, for each query, a set of human relevance judgements—manual decisions as to which of the documents in the collection satisfy the information need. In this paper we use the TREC test collection [14], which consists of several gigabytes of text data, several hundred queries, and several thousand relevance judgements for each query. We focus on the data generated in the 1996 round of TREC, specifically the 2 gigabytes of data on TREC disks 2 and 4 and queries 251–300.

Given a set of test data, evaluation of each test query with a similarity measure yields a ranked set of answers, in which a proportion—the precision—are relevant, and containing a proportion—the recall—of the known relevant answers. Averaging precision at fixed numbers of documents returned yields overall average-precision, a standard metric for measuring effectiveness.

Efficiency and effectiveness are interdependent: achievement of efficiency often involves some sacrifice of effectiveness. Some of the evaluation mechanisms described below use heuristics that are designed to have minimal impact on effectiveness while significantly reducing costs.

3 Evaluation of ranked queries

Evaluation of a query by exhaustive comparison of it to each document is almost always prohibitive; efficient query evaluation requires some form of index. Many studies have shown that effective retrieval requires consideration of all terms occurring in documents, other than stopwords (content-free terms such as "the" and "furthermore"); attempts to reduce the volume of index terms to a smaller number of descriptors have not been successful. The standard structure for indexing documents for ranking is an inverted file, the only practical index structure for this task [4, 35, 38], which consists of two components: a vocabulary containing each distinct term in the collection and, for each term, an inverted list. The list must include the identifier of each document containing the term, the in-document frequency of the term in the document, and may also include information such as the positions of the word in the document to allow proximity to be determined.

For efficient processing inverted lists must be sorted by increasing document identifier; for efficient retrieval they must be stored contiguously (or, for long lists, in large blocks). Although this approach has some effect on update costs and disk fragmentation, the impact is surprisingly moderate, and has significant benefits for query evaluation [38]. Throughout this paper we assume that inverted lists are stored efficiently, using compression techniques based on variable-length

1. Create an array of accumulators, one for each document in the collection.

2. Process the inverted list of each query term t in turn, so that each inverted list is fully processed before the next is begun. For each document identifier d and in-document frequency $f_{d,t}$ in the inverted list for t, update d's accumulator by adding

$$w_{q,t} \cdot w_{d,t} = \log_e(f_{d,t} + 1) \cdot \log_e(f_{q,t} + 1) \cdot \log_e(N/f_t + 1)$$

   to its current value.

3. Pass through the array of accumulators, dividing each by W_d and identifying the k greatest values; the corresponding k documents are fetched and returned to the user.

Figure 1: Term-order or TO processing for document ranking.

codes. Compared to lists with fixed-length fields, disk transfer costs are reduced by a factor of about three to six, and seek costs are somewhat reduced (because of the smaller number of disk tracks occupied by inverted lists) [20]. The trade-off is the requirement for decoding during query evaluation; we consider the impact of this decoding below.

Using an inverted list there are in broad terms two standard strategies for evaluating a ranked query, term-ordered query evaluation and document-ordered query evaluation. We now describe these strategies and some refinements, and discuss the major evaluation costs.

Term-ordered query evaluation

The query evaluation strategy most often described in the literature is *term-ordered* or TO processing. In this strategy, the inverted list for each term is fetched, processed, and discarded before the next list is considered. The lists should be considered in order of decreasing rarity, so that the information about the rarest term is processed in full before the information about the next term is fetched. Each entry in each list contains information about the occurrence of a term in a document; to assemble this information into similarity values, it is necessary to maintain a set of *accumulators* containing partial similarities computed so far for each document. In TO processing, shown in outline in Figure 1, this set is represented as an array with one element per document in the collection. Note that, even for queries of just a few terms, a high proportion of the documents in the collection can have non-zero accumulators.

The resource costs of query evaluation include disk, memory, and CPU cycles, and vary depending on query and database statistics. These costs cannot easily be combined: any model that integrated them into a single value would be based on narrow assumptions for parameters that vary dramatically between machines and applications, such as the relationship between the cost of fetching a bit from disk, the cost of processing an addition, collection size, the length of a typical query, and the distribution of query terms. What is significant is the characteristics of these costs: their asymptotic behaviour and how they vary as the properties of query and database change. Analysis of these characteristics allows us to make predictions about the relative behaviour of different query evaluation techniques.

For TO evaluation, the main resource costs are as follows; we show below how they combine in practice.

F^D_{TO}: Fetching of the inverted lists from disk. Term-ordered processing allows the whole of each inverted list to be fetched in a single operation, or a small number of operations for lists whose length exceeds the buffer size. A single read is the most efficient way of getting lists into memory. It is for this reason that, as discussed above, lists should be stored contiguously on disk, a strategy that makes update somewhat more expensive but greatly simplifies query evaluation [35, 38], which is typically the dominant cost in text databases.

This cost is asymptotically linear in the size of the collection, assuming that the rarity of query terms (that is, the proportion of documents in which each query term occurs) does not change, but in practice this cost grows more slowly because seek costs are significant and the number of seeks is roughly constant. This cost is also linear in the number of query terms.

A single fixed-size buffer is required to store the current inverted list; this cost is independent of query length.

P^D_{TO}: Processing inverted lists. This cost has two components. The first is the time to decompress each list in full, which on current machines is somewhat less than the time saved in disk transfer costs, giving a net saving overall [20]; indeed, trends in hardware are further favouring the use of compression for indexes [34]. The second component is, for each document identifier and in-document frequency, computation of $w_{q,t} \cdot w_{d,t}$ and update of the appropriate accumulator. This cost can be reduced somewhat by, for each term, precomputing the $w_{q,t} \cdot w_{d,t}$ values corresponding to small in-document frequencies, which account for the vast majority of inverted list entries.

These costs are linear in the number of documents in the collection and in the number of query terms.

A^D_{TO}: Storing and processing the array of accumulators. The accumulators consume memory, for storage, and CPU cycles, for normalisation and for search for high similarity values. Both of these costs are linear in the number of documents in the collection, but are independent of query length.

In an environment with limited memory it may be necessary to store the accumulators in a compact, fixed-size structure, then, when this structure is full, write it to disk and begin again. At the end of processing of inverted lists the temporary structures must be fetched and merged. This style of processing imposes significant costs, but is probably unnecessary; it is far more efficient to simply limit the number of accumulators, as discussed below.

In order to reduce the number of accumulators, an *almost-zero* memory strategy can be used: fetch the first two inverted lists, merge them, write the intermediate result to disk, then iteratively merge this result with each subsequent list, writing the new result to disk each time [30]. If buffer space is limited, inverted lists can be fetched in smaller blocks. This strategy has the disadvantage that, for longer queries, the intermediate list will contain around one entry for each document in the database and must be read and written for each query term.

A further cost is the overhead of fetching the answer documents, which however is independent of the query evaluation technique. For small numbers of answers this cost is small, and we do not consider it in our comparisons.

Improving term-ordered evaluation

A simple improvement to term-ordered evaluation is to use stopping: stop-words are typically the most frequent words in the collection, and processing them is expensive and has little impact on effectiveness [27]. Removing further query terms, however, reduces effectiveness; Moffat and Zobel observed that effectiveness grows with number of query terms processed [20], and we observed that effectiveness is correlated with query length, as illustrated in Figure 8.

Another improvement is to limit the number of accumulators in some way. While most accumulators are typically non-zero, most accumulator values are trivially small, containing only a contribution from a less significant term. Several heuristics have been proposed for limiting of accumulators [4, 6, 15, 20, 23]. One of the most effective is to simply insist on an absolute bound to the number of accumulators [20]. In this approach, denoted TOS, as the first inverted lists (corresponding to the rarer query terms) are processed, each reference to a new document provokes

creation of an accumulator. Once the structure of accumulators is full, by which stage it is likely that processing has proceeded to terms that are not rare, existing accumulators can be updated but creation of accumulators is no longer allowed. Experiments with the TREC data showed that limiting the number of accumulators to around 5% of the number of stored documents had no impact on effectiveness. The limited-accumulator technique is of most value when the number of query terms is large, but even for queries of two terms it can result in measurable savings [20].

The main impact of this change is to reduce the cost of storing the accumulators. A more complex structure is required to hold them—the array must be replaced by a lookup structure such as a hash table that maps document identifiers to accumulators—but even so the new cost A^D_{TOS} is around one-tenth that of A^D_{TO}. The cost of accessing accumulators increases, since the lookup is less direct, but this overhead is offset by the technique described below, which allows the structure to be accessed less often.

A consequence of limiting accumulators is that, for the inverted lists corresponding to more common terms, most of the document identifiers and in-document frequencies are not used, because they correspond to documents that have not been allocated an accumulator. By adding structure to each inverted list, such as dividing each list into blocks and storing pointers (or *skips*) to the start of each block, much of the processing of unnecessary information can be avoided [20]. For the longest lists, only a small percentage of the content need be decoded.

This skipped TO processing, or TOS processing, can greatly reduce overall costs. By analogy with TO processing, the costs are F^D_{TOS} for fetching lists, P^D_{TOS} for processing lists, and A^D_{TOS} for processing accumulators.

The use of skips adds slightly to fetch costs, as lists are lengthened by the inclusion of additional pointers, with perhaps $F^D_{TOS} \approx 1.25 \times F^D_{TO}$. However, processing costs are greatly reduced; experiments with long queries have shown reduction by a factor of four and somewhat less saving for short queries [20]. Probably the most accurate statement that can be made about the relative costing is that P^D_{TOS} is always less than P^D_{TO}, and is much less for long queries. The difference between A^D_{TO} and A^D_{TOS} is not dependent on query length; the latter is typically a fifth to a tenth of the former.

Consider a typical query of 5 terms on our test data. With TO processing on our test machine, a Sparc 20, the time in seconds to fetch inverted lists is $F^D_{TO} = 0.2$, the time to process these lists is $P^D_{TO} = 0.2$, and the time required for initialisation and final processing of the accumulators is $A^D_{TO} = 0.4$ (total 0.8 seconds). For a query of 30 terms, the times are $F^D_{TO} = 1.0$, $P^D_{TO} = 1.5$, and $A^D_{TO} = 0.5$ respectively (total 3.0 seconds). By comparison, for TOS processing, for the same 5 term query the times were $F^D_{TOS} = 0.2$, $P^D_{TOS} = 0.2$, and $A^D_{TOS} = 0.1$ (total 0.5 seconds), and for the same 30 term query the times were $F^D_{TOS} = 0.9$, $P^D_{TOS} = 1.1$, and $A^D_{TOS} = 0.1$ (total 2.1 seconds). Average elapsed times per query are shown in Figure 3.

An alternative structuring strategy of hierarchical reordering of lists [2, 21] can reduce the decoding cost further, so that only the inverted list fetch cost and the accumulator update cost are significant for long queries. Briefly, in this strategy the contents of each list is turned into a balanced binary tree, which is then stored breadth-first, so that linear processing of the list yields a breadth-first tree traversal, thus allowing more efficient skipping. There are practical disadvantages: in particular, it is not clear whether word position information can be readily incorporated into inverted lists structured in this way, and update is difficult. However, for the standard task of whole-document ranking this strategy is more efficient than skipping.

Yet another approach to improving term-ordered query evaluation is to sort inverted lists by in-document frequency rather than document identifier, so that the highest $w_{q,t} \cdot w_{d,t}$ values are at the front of each list [23]. Query evaluation can then proceed by considering each list in turn until the frequencies become, by some heuristic, too small. The rest of the list can then be ignored and the next list fetched. For longer queries, typically only the first disk block of each list is required [5]. Persin et al. [23] reported that, on a 500 Mb collection, this method could reduce evaluation time by a factor of three without loss of effectiveness; the proportional savings should grow with increased collection size.

Another query evaluation strategy that uses the same index structure is to search amongst lists so that the highest $w_{q,t} \cdot w_{d,t}$ values are processed first, regardless of which term is used [22].

1. Fetch the first fragment of each inverted list and create an array of pointers, one to the start of each list. Heapify the array by the smallest document identifier in each list. Create a separate, empty heap, of capacity k, for highly-ranked documents.

2. Consume the lists as follows. For the unprocessed document d with the smallest identifier, identify every inverted list referring to that document and compute

$$\sum_{t \in q \wedge d} (w_{q,t} \cdot w_{d,t}) .$$

   Divide by W_d; if the result is larger than the current least similarity value, add d to the heap of highly-ranked documents. Update the pointers (consuming the information about d), fetch further inverted list fragments as necessary, reheapify the array of inverted lists, and continue. This strategy requires that inverted lists be sorted, which has only a small impact on update costs [35, 38].

3. The k documents in the heap of highly-ranked documents are fetched and returned to the user.

Figure 2: Document-order or DO processing for document ranking.

While this strategy is effective for document ranking, it cannot be directly applied to passages, in particular because in-passage frequencies are much smaller. Since our interest is in strategies for passage ranking, we do not consider this form of frequency sorting in this paper. In Section 8 we consider a variant form of this algorithm, but show, based on initial results, that it is unlikely to be a useful approach for passages.

Document-ordered query evaluation

The other main strategy for query evaluation is *document-ordered* or DO processing. In this strategy, the inverted lists of all query terms are processed in parallel; all the information about each document is consumed from all of the lists simultaneously. DO processing is shown in outline in Figure 2. For DO evaluation, the main resource costs are as follows.

F^D_{DO}: Fetching of inverted lists from disk. For a reasonable number of query terms, it may be infeasible to simultaneously buffer the whole of each inverted list. (Assumptions about buffer use and memory availability are considered further below.) Thus each list may have to be fetched in fragments.

The cost of fetching inverted lists is linear in the size of the collection, assuming that the rarity of query terms does not change. If there is not enough space to simultaneously buffer all of the inverted lists, the need for multiple seeks for each inverted list means that F^D_{DO} may be much greater than F^D_{TO}, and is likely to be dominated by the number of seeks. In this case, the cost is in principle linear in the number of query terms, but in practice increases sharply when buffer size is exceeded.

A single fixed-size buffer is required to store the current fragments of inverted lists; this cost is constant.

P^D_{DO}: Processing inverted lists. This cost has three components. The first is time to decompress each list in full. The second is, for each document identifier and in-document frequency, computation of $w_{q,t} \cdot w_{d,t}$. The third is, for each document identifier, the cost of searching and rebuilding the heap of inverted lists.

These costs are linear in the number of items in the collection but are $O(m \log m)$ for m query terms, because of the need at each stage to search for the query term corresponding

to the next least document identifier. For a large number of query terms this cost could in practice be dominant.

In DO as described here, there is only one accumulator, necessitating search amongst the inverted lists at each step; but without a structure of accumulators, there is no obvious way to reduce the computation cost of list merging. An alternative strategy is to allocate a small array of t accumulators, representing the next t documents. At each stage, query evaluation would proceed by cycling through the term lists, consuming information pertaining to any of these documents. These would then be searched for the highest similarity values. This avoids the search amongst lists, but reintroduces the costs of TO processing, since in effect each document is represented by an accumulator, and in addition each inverted list must be fetched in fragments. We do not consider this strategy further.

Continuing the previous example, for a query of 5 terms the times in seconds are $F^D_{DO} = 0.2$ and $P^D_{DO} = 0.6$ (total 0.8 seconds); for a query of 30 terms the times are $F^D_{DO} = 1.3$ and $P^D_{DO} = 4.8$ (total 6.1 seconds). Average elapsed times per query are shown in Figure 3.

Note that strategies such as skipping and inverted list reordering cannot be used in conjunction with DO processing, since with DO there is no way of identifying in advance that index information is not of interest.

Overall, for a large collection and a small number of query terms DO processing may be preferable to TO or TOS processing: buffer space may allow each inverted list to be fetched whole, the overhead of processing a large array of accumulators is absent, and skipping gives relatively little advantage. As the number of query terms rises, the penalties for DO of multiple accesses to fetch each list and of manipulating the heap of inverted lists become more significant, while skipping should yield relatively greater improvements to TOS processing. The crossover point between the two schemes depends on the size of the collection, the expected probability distribution of query terms, and the relative costs of instruction processing and disk accesses on the host machine.

Parallelism

There are two forms of parallelism in retrieval systems: the use of multiple processors and parallel disk technologies to improve system performance, and the ability to handle multiple users simultaneously. With regard to the former, extending the capacity of the hardware does not affect the trend costs of query evaluation. The impact of the latter, user parallelism, is two-fold. First, simultaneous processing of multiple queries makes it harder to schedule accesses to disk—for example, a process cannot access one block of a file then expect, some time later, that the disk head will be conveniently positioned for fetch of the subsequent block. Second, significant user parallelism reduces the buffer space available to each user; and in many current environments, such as digital libraries and internet search engines, economics dictate that systems must support hundreds of concurrent queries per CPU. Memory must be shared amongst these queries. If a query involves megabytes of inverted lists, or an evaluation technique needs megabytes of temporary structures, it is unreasonable to expect all the data for the query to be simultaneously available in memory.

The net relative impact of these considerations on TO and DO processing depends on the collection and the exact amount of space available, because they make somewhat different use of buffer space. On the one hand, it is apparent that DO processing requires that lists be fetched in smaller fragments than for TO processing, because of the need to hold multiple list fragments simultaneously. On the other hand, for the particular extreme of very short queries, huge collections, and very high user parallelism—as typified by the web search engines—there may simply not be enough space to store accumulators at all.

As the number of users on the machine rises, the memory per user falls. For TO processing, once there is insufficient space for a full array of accumulators query evaluation can proceed only by using temporary accumulator structures on disk; it follows that, if the number of users doubles, more disk accesses are required to resolve each query. However, with the almost-zero strategy (of

fetching and merging lists in pairs) virtually no memory at all is required between processing of each inverted list, so concurrent processing may be able to continue even under heavy load.

For TOS processing, as the number of users increases the accumulators per user must be reduced; once this number is too small effectiveness will become unacceptable, and the system must defer evaluation of new queries. However, as the number of accumulators is reduced, the cost per user falls, so that queries should continue to leave the system quickly. Alternatively, TOS processing can incorporate the almost-zero strategy, with, in comparison to TO processing, the advantage that there is an upper bound on the size of the temporary accumulator structure—at worst it will be comparable in size to a typical inverted list.

For DO processing, as the number of users increases, buffer space for inverted lists must be reduced. On the one hand, this reduction in space has no impact on effectiveness. On the other hand, doubling the number of users (that is, halving buffer space per user) also doubles the number of disk accesses required to resolve each query. This cost will rapidly dominate, so that doubling the number of users is likely to quadruple the load. If space is reduced too far, therefore, the system will thrash.

Turtle and Flood [30] developed a cost model for comparing DO and almost-zero TO evaluation, measuring the amount of information read and written by each strategy. The model assumes that memory is limited, and that the cost of processing inverted lists is dominant. Their analysis, motivated by investigation of algorithms that require only limited memory, suggests that the costs are lower for DO processing than for almost-zero TO processing. However, our analysis of query evaluation shows that the cost of accessing the information in each case is not the same, thus invalidating the model; and if adequate buffer space is available for a full complement of inverted lists, this buffer space could instead be used for accumulators.

With TOS evaluation, accumulator structures need not be large, thus largely removing the motivation for the almost-zero strategy. Rather than propose an abstract cost model, of uncertain value when so many aspects of the costs are variable, we experimentally tested the relative performance of TO, TOS, and DO query evaluation. Results are given in the next section.

4 Experiments with document retrieval

To guide our development of query evaluation algorithms for passage retrieval we compared the performance of the strategies discussed above for the more conventional case of whole-document retrieval. We chose to examine the relative costs of TO and DO processing experimentally, searching for trends rather than absolute behaviour. The hardware used was a Sparc 20 substantially free of other load and with 384 Mb of memory. In this environment, buffer sizes greatly exceed the total size of all structures used and retrieved during query evaluation, so that inverted lists could always be fetched in full. Between each query the memory was flushed to eliminate caching effects. The system used was the prototype text retrieval engine MG [3, 35], with modifications to allow DO processing. We are confident (after analysis of several false starts) that the implementation in each case is of good quality.

The database used was disks 2 and 4 of TREC [31], which together contain about 530,000 documents. The queries used were the full text of topics 251-300; after stemming, casefolding, and elimination of duplicate terms, their average length is approximately 30 terms. To simulate the effect of increasing query length, we generated 1-word queries by choosing the first word of each topic (that is, the first word of the title); 2-word queries by choosing the first two words of each topic; and so on. Topics of less than k words were not used to produce k-word queries, so that as k increased (past 14, the length of the shortest topic) average performance was over a decreasing number of queries. The different query evaluation methods are compared in Table 1, showing average effectiveness and average recall (or number of relevant documents) in the top 1000 documents retrieved, with the full queries and the TREC-5 data.

We used this data to measure the efficiency of the whole-document query evaluation mechanisms discussed above. Results for elapsed time are shown in Figure 3. (In all our experiments the same behaviour is shown by CPU time as by elapsed time. For this reason we do not show

Table 1: Retrieval effectiveness for document ranking, on the TREC-5 data. Each "k" figure represents '000s of accumulators.

	Recall	Precision at N documents			Avg.
		5	10	20	prec.
TO, DO	48.5	0.3680	0.3440	0.3070	0.1836
TOS, 10k	41.7	0.3760	0.3300	0.2970	0.1710

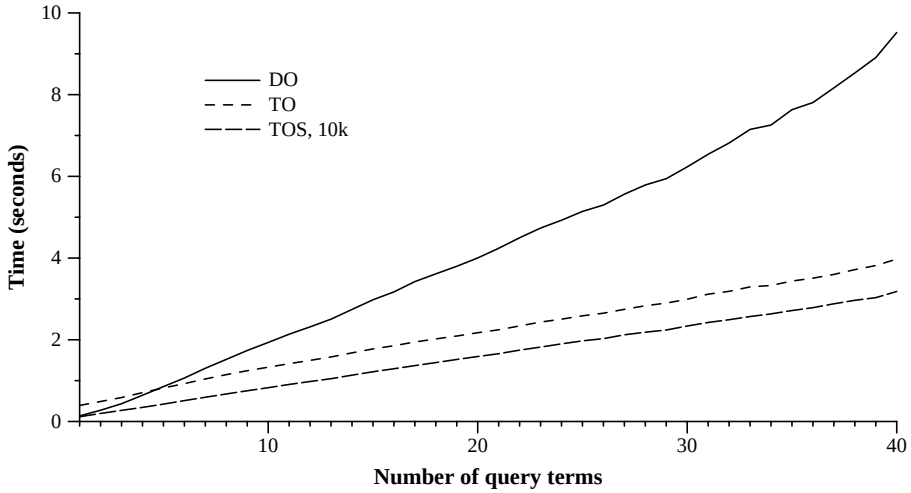

Figure 3: Document ranking: Average elapsed time for TO, TOS, and DO query evaluation on TREC-5 data. Each "k" figure represents '000s of accumulators.

results for CPU time.) For TOS processing there were 10,000 (or 10k) accumulators for index building and querying, or about 2% of the total number of documents, a figure that gives effectiveness indistinguishable from that of ordinary TO processing. As can be seen, TOS processing is always faster than TO or DO. TO processing is penalised by the initial cost of processing the full array of accumulators, but is otherwise only slightly slower than TOS—for this data, the cost of additional decompression is almost offset by simpler testing at each document entry and faster access to each document's accumulator. Note that Moffat and Zobel [20] observed bigger gains due to skipping, because in our case the queries are shorter, there are fewer documents in the collection, the processor we used is faster relative to the disk (so that the processing gains yielded by skipping are less significant), and some inefficiencies have been removed from TO processing in MG. However, the overall trends are similar.

For queries of more than a few terms, DO processing is much slower than even conventional TO processing. We believe that the reason is that the cost of searching and maintaining the heap of inverted lists rapidly becomes dominant as query length grows. These results confirm our supposition that, as query length increases, DO processing becomes relatively more expensive. For long queries TO or TOS evaluation is clearly preferable, as it is for use with techniques such as relevance feedback or query expansion [1, 18, 36], which greatly increase the number of query terms.

To confirm our observations we built a "document" collection of short pieces of text of 50–500 bytes each, from the same data; these pieces of text were identified by using paragraph breaks where possible, but introducing breaks or coalescing paragraphs where they were outside the 50–500 byte range. This gave a collection of around 7.7 million small documents. The intention was to explore behaviour on a large collection and magnify the difference between the different methods of evaluation.

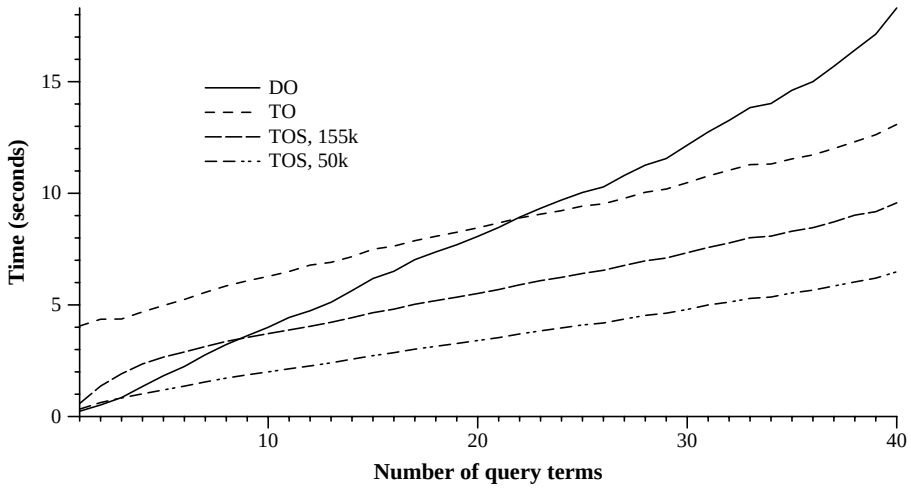

Figure 4: Document ranking: Average elapsed time for TO, TOS with 155k accumulators, TOS with 50k accumulators, and DO query evaluation on the TREC-5 "50–500" collection. Each "k" figure represents '000s of accumulators.

Results are shown in Figure 4, and illustrate the weaknesses of all three approaches to query evaluation. For short queries DO was the fastest, but grew in cost the most rapidly as query length increased. As for the collection discussed above, TO was penalised by the need to initialise and later normalise large numbers of accumulators. The behaviour of TOS was more complex. For short queries no advantage can be taken of skipping, and the cost rises rapidly while the number of distinct documents in the inverted lists is less than the number of accumulators, presumably because of the greater complexity of index decompression and accumulator update. We ran two separate sets of experiments with TOS, one based on 155,000 accumulators, or 2% of the total number of documents [20], a figure that is likely to give good overall effectiveness; and another based on 50,000 accumulators, a figure that is likely to give good precision in the top 10 or 20 documents retrieved, the most common retrieval task. As can be seen, reducing the number of accumulators has a dramatic impact on costs.

Turtle and Flood's comparison of processing techniques [30] assumed that only a limited amount of memory is available to store intermediate results. In our experimental work, we have assumed that intermediate results (either partial similarities or the complete set of inverted lists used to evaluate a query) can fit in memory. With stricter memory limits, elapsed time for DO and TO processing rises as discussed above, while elapsed time for TOS processing falls. Thus these results represent a best case for DO and TO processing, and overall TOS is preferable for document retrieval.

However, for passage retrieval the relative costs are different, giving DO processing a significant advantage. In the remainder of this paper we explore query evaluation for passages.

5 Evaluation of passage ranking

The aim of passage ranking is to identify short blocks of contiguous text as likely answers; each passage can be returned directly to the user, or in a production system returned with context information such as a description of the document from which it was drawn. However, as discussed earlier there are many possible definitions of passage; the method for identifying passages in text can significantly affect the performance of passage retrieval.

As discussed earlier, document retrieval based on fixed-length passages has been shown to be more reliable than retrieval based on passages defined in terms of logical or semantic properties [7, 17]. Based on these results we define a passage to be any fixed-length contiguous sequence of words

1. Create a set of accumulators A_p, each set to 0.
2. Let Q be the set of unique terms in the query.
3. For each term t in Q:
    (a) Seek to and fetch inverted list I_t.
    (b) While I_t is not empty:
        i. Get the next document d_t from I_t.
        ii. For each position offset $l_t \in d_t$ from I_t, determine the range of passages containing l_t.
        iii. Update the partial similarity of each passage.
4. Initialise a min-heap h of size k, for top-scoring passages.
5. For each passage accumulator a_p in A_p:
    (a) Add a_p to h if a_p is sufficiently large.
    (b) Rebuild h.
6. Retrieve the k documents containing the top passages.
7. Destroy A_p.

Figure 5: Term-order or TO processing for passage ranking.

from a document. That is, any word in a document, other than words too close to the end, can be the start of a passage. Passage lengths of 150–500 words consistently give good effectiveness, for the task of using highly ranked passages to identify relevant documents [17]. (Passage ranking can of course be used for identification of passages themselves, but measurement of effectiveness for this task is not supported by the standard test collections.) Also, with fixed-size passages there is no need for document length normalisation, since each passage has the same number of terms and similar cosine-based length.

A naive implementation of passage ranking would be expensive, primarily because the number of distinct items to be ranked is so large. Potentially there are many distinct passages containing each occurrence of each query term anywhere in the database, that is, each term occurrence is in many passages. As a consequence, with TO processing vast numbers of partial similarities need to be maintained, and even within a single document hundreds of passages must be considered. Yet to some degree this cost appears to be unavoidable: if passages do not overlap effectiveness declines [17].

However, as we show with careful algorithm design both TO and DO processing strategies can be used in practice for passage ranking. We now describe our proposals for applying these strategies to passage ranking.

Term-ordered passage query evaluation

Passage ranking with TO processing is similar in outline to TO document ranking; in the description below we focus on the differences between the two cases. We assume that a full word-position index is used, that is, that each inverted list contains, in addition to document identifiers, the ordinal location of each word in each document. Word-position indexes are the norm in practical information retrieval systems, which must support features such as proximity queries and query-term highlighting in addition to conventional ranking. An elementary algorithm for TO processing for passage ranking is shown in detail in Figure 5. The costs are as follows.

F^P_{TO}: Each inverted list is fetched fully before it is processed. The inverted lists are longer than for document ranking since word positions are included, and thus larger buffers are needed.

P^P_{TO}: The cost of index decompression is increased due to longer inverted lists.

Passages are not themselves indexed. Instead the word-level index is used to identify the positions at which passages occur. Every query term occurrence contributes to multiple passages. Calculation of frequencies for terms in passages is a little complex; for each candidate passage covering each term occurrence it is necessary to determine how often the term occurs in the passage.

A^P_{TO}: Since terms in queries are processed one at a time, it is not possible to predict where the best passage will occur. Thus each passage needs an accumulator, managed (naively) as a large array with one element per distinct passage in the database, or as a dynamic structure of accumulators for the passages that contain a query term.

The major cost in TO processing is maintenance of accumulators. For instance, in TREC 5 the number of passages is around 250 million. Clearly the requirement of one accumulator for every passage is not sustainable, and even one accumulator per passage containing a query term is unlikely to be tenable for a large collection. One way of reducing accumulator requirements is to observe that neighbouring passages have identical similarity scores. Thus an accumulator for a passage need only be created when a query term appears at the centre of it. This approach does have a cost—additional processing to compute partial similarities for new passages that overlap with existing ones, and temporary space for the locations of query terms that have been already processed—and it is not obvious what data structures would be suitable.

Another way of reducing the set of accumulators is to apply a limit (such as 2% of the total number of passages), thus allowing the use of skipping and giving new costs F^P_{TOS}, P^P_{TOS}, and A^P_{TOS} as for document ranking with skipping. The expected gains from skipping are higher than for document ranking since there are more units to be ranked.

Document-ordered passage query evaluation

Passage ranking with DO processing follows the same steps as for document ranking. DO passage ranking is shown in detail in Figure 6. The costs are as follows.

F^P_{DO}: As for document queries, but greater because inverted lists for passage ranking are longer due to the inclusion of word position information.

P^P_{DO}: As for document queries, but greater due to longer inverted lists.

Partial similarities are computed per passage, that is, every passage is evaluated fully before the next one is considered.

Passage-based DO ranking differs from document-based DO ranking primarily in the processing required within each document (step 5 of the algorithm), to identify the highly-ranked passages within each document, in set L. After step 5(c) in Figure 6, L contains all the positions of the query terms, ordered by their appearance in the current document being processed (d_{curr}). Step 5(d) is a linear process for computing similarities, which proceeds as follows. Starting with the first entry in L, which determines the start of the initial passage, query term occurrences in L are processed sequentially until an occurrence outside the current passage is found; at this stage the passage similarity can be computed. Then, within-passage frequencies for query terms are reinitialised and processing continues with the second entry in L, which determines the starting point for the next passage. This process continues until all the entries in L are consumed.

The differences between DO and TO evaluation are more significant in passage ranking than in document ranking. The costs for short queries using TO are expected to be higher than for DO: the costs of DO evaluation are primarily affected by the number of terms in the query and

1. Let Q be the set of unique terms in the query.
2. Create a min-heap H of size $|Q|$ for the inverted lists, ordered by least document identifier and least term position.
3. Create a min-heap h of size k for documents to be retrieved. At the end of the algorithm, h will contain k most highly ranked documents.
4. For each term t in query Q, initialise processing of inverted lists:
   (a) Allocate memory for inverted list I_t, then fetch I_t.
   (b) Get the first entry from I_t and add it to H as $\langle t, d, \mathit{positions}\rangle$, where *positions* is a list of term t occurances in d.
5. While H is not empty, process inverted lists in document order:
   (a) Let d_{curr} be the document identifier from the first entry in H.
   (b) Create an empty occurrence list L, for occurrences of terms in d_{curr}.
   (c) While d_{curr} is same document d as in the next entry $\langle t, d, \mathit{positions}\rangle$ in H:
      i. For each position x of t in *positions*, add $\langle t, d, x\rangle$ to L and rebuild H.
      ii. Remove $\langle t, d, \mathit{positions}\rangle$ from H.
      iii. Get the next entry from I_t and add it to H.
      iv. Restore heap properties for H (as in step 4(b) above).
   (d) Process L to evaluate the similarities of all passages in d_{curr} that have non-zero similarity with Q, keeping track of only the most similar passage in d_{curr} (d_p).
   (e) Add d_p to h.
6. Convert h to a max-heap.
7. Retrieve the top k documents.

Figure 6: Document-order or DO processing for passage ranking.

inverted list length, while collection size has direct impact on TO, since accumulator space is linear in collection size.

Also, DO ranking allows a significant optimisation: as all terms are considered at once, at most one passage need be considered for each occurrence of each query term. In contrast, with TO processing it is not known which passage will have the highest score until all query terms have been considered, so it is necessary to keep partial similarities for all likely passages. For the task of using passages to identify relevant documents, only the highest passage score in each document need be kept, but this does not lead to large cost savings. Nonetheless, the cost of DO passage ranking is expected to be greater than that of DO document ranking, because each term occurrence is considered individually.

Boundary restriction

In the discussion above we assumed that a passage could start at any word in a document, as our earlier experiments showed that this flexibility was necessary for passage retrieval to be robust [17]. However, it is also apparent that it leads to unacceptable costs. On the other hand, the economical solution of prohibiting passages from overlapping can significantly degrade effectiveness.

An intermediate solution that reduces costs without significant impact on effectiveness is to restrict passage boundaries to starting only at multiples of W words. (An alternative is to define

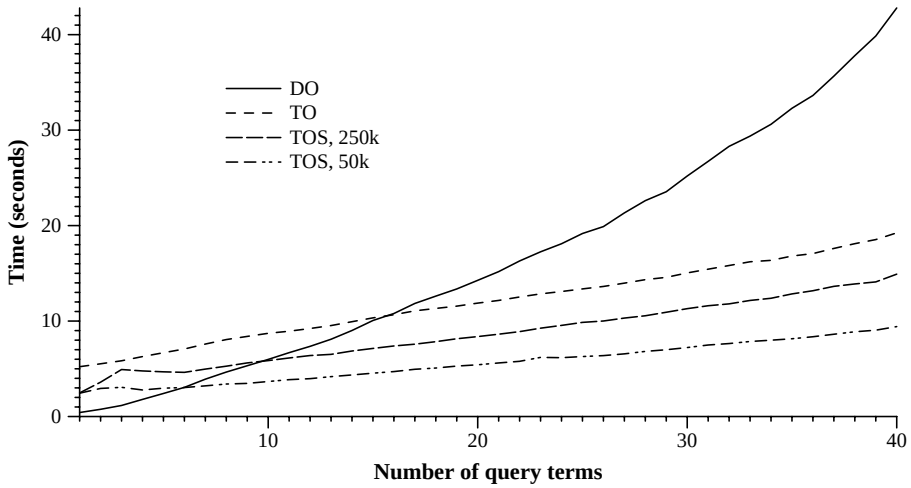

Figure 7: Passage ranking: Average elapsed time for TO, TOS with 250k accumulators, TOS with 50k accumulators, and DO query evaluation for fixed-length overlapping passages of 150 words, on TREC-5 data. Each "k" figure represents '000s of accumulators.

passage boundaries in relation to the position of query terms, as in the work of Callan [7], a solution that is similar in principle but requires different underlying structures.) Allowing passages to start only every W words reduces the number of accumulators by a factor of W. Our previous experiments [17] show that for small W (say 25), and passages of 100 words or more, retrieval effectiveness is not significantly degraded.

When passages are restricted to start at multiples of W, the index can reflect this by indexing W-word units. Each small segment can be indexed independently and be treated as segment identifier and word-frequency pair, slightly simplifying query evaluation. If W is small, most frequencies are one, potentially allowing omission of frequencies altogether. Alternatively, word-level indexes can be used for passages that start every W words, at the expense of slightly longer inverted lists and slightly more complex processing but allowing other query types such as Boolean queries and word pairs. In the experiments below we have used indexes of 25-word segments, because doing so allowed easier development of the test software. In practice, assuming that only passages at W-word intervals are to be considered, we are confident that neither index type is likely to be significantly superior.

6 Experiments with passage retrieval

We have experimentally compared TO, TOS, and DO processing for passage retrieval, for passages commencing at 25-word intervals. The passage lengths investigated were 150 words and 300 words, which are effective in the TREC environment given enough accumulators [17].

Evaluation times for varying length queries are shown in Figure 7. For short queries, DO processing is fastest; as discussed above, the simultaneous processing of terms in DO means that fewer passages need be considered. DO performance degrades rapidly as query length increases, displaying superlinear increase; this superlinear component is more significant than for document ranking because the distribution of term occurrences is more skewed towards common terms than is the distribution of in-document frequencies. In contrast, the slope for both skipped and non-skipped TO evaluation is linear. The overhead for short queries in TO is due to initialisation and final processing of accumulator structures. As for document retrieval, TOS is faster than TO, and reduction in the number of accumulators leads to a significant time saving; in the higher line there are 250,000 accumulators, or 2% of the number of passages; in the lower line there are 50,000 accumulators.

Table 2: Average per-query memory requirements for retrieval of 150-word passages, on TREC-5 data. Each "k" figure represents '000s of accumulators.

	Accums (Mb)	Buffers for inv. lists (Mb)	Other (Mb)	Total (Mb)	No. of accums
TO	48.03	0.50	—	48.53	12,591,947
TOS, 250k	6.07	0.65	—	6.72	299,298
TOS, 50k	2.63	0.61	—	3.24	74,135
DO	—	3.48	0.29	3.77	—

The rapid increase in TOS processing costs for short queries is due to the initial stage when new accumulators are created. As the number of query terms increases, so does the number of new accumulators. Once accumulator space is exhausted and inverted list entries that do not correspond to existing accumulators are skipped, times rise only slowly. However, DO processing remains an attractive option for short queries. Increasing passage size to 300 words does not change costs significantly for DO or TOS, but TO evaluation becomes more expensive since there are twice as many accumulators accessed for each document and twice as many overall.

Speed is not the only factor that distinguishes the techniques; memory consumption is also important. DO and TOS processing use less memory than TO, as shown in Table 2. Full queries have been used to record these results—shorter queries would reduce memory requirements for DO more sharply than for the other methods. The "Other" column in Table 2 shows the additional memory required to perform ranking not present in other methods; structures that are common between all algorithms, such as the dictionary, are not recorded. The accumulator space for TO processing is huge. Even with TOS and 250,000 accumulators, around 6 Mb on average is required for these queries averaging 30 terms. (Note that the projected number of accumulators is a target only, and can be exceeded in individual queries because the decision to turn off accumulator creation mode is only made between processing of individual inverted lists, not during processing of individual lists.) The use of 6 Mb for a 2 Gb collection may well be too expensive in a multi-user environment. DO processing is efficient in accumulators but requires more buffer space for lists.

The optimisations presented so far have been designed to decrease response time and reduce memory requirements, but, as system resources are reduced, retrieval effectiveness is expected to degrade. The optimisations have been intended to maintain good effectiveness: results for retrieval effectiveness are shown in Table 3. The data used is the TREC 5 collection as before, and the query set is the full version of the 50 queries. The first row shows the best results achieved using document retrieval based on the best passage, with standard TO processing. As the number of possible accumulators is reduced, the overall retrieval effectiveness degrades. However, TOS with 250,000 accumulators is almost as effective as standard TO processing, and indeed achieves higher precision for small numbers of documents retrieved, and it is unlikely that increasing the number of accumulators to be greater than 2% of the total number of documents would significantly improve effectiveness. Overall, all methods have similar precision for retrieval of up to 20 documents, the commonest retrieval task in practice. Moreover, by comparison with Table 1 it can be seen that effectiveness in each case is at least as good—and for precision over the first few documents, much better—than is obtained with whole-document ranking.

The effectiveness of 150-word passage ranking and document ranking is compared in Figure 8, over a range of query lengths. In this experiment, we used TO processing for documents, with and without pivoted length normalisation [28], and passages; and used, of the original query set, the 21 queries that had at least 30 terms, to ensure that the average effectiveness at each query length is measured over the same query set. This experiment demonstrates that restricting the number of query terms—potentially a worthwhile heuristic for reducing evaluating costs—degrades effectiveness. Even at 30 terms, none of these methods is yet close to the average effectiveness over all query terms, of 0.2239 for cosine, 0.2245 for pivoting, and 0.2441 for passages. (Note that

Table 3: Retrieval effectiveness for 150-word passages, on TREC-5 data. Each "k" figure represents '000s of accumulators. Percentage changes in average precision are shown as "deltas" (%Δ), taking TO or DO effectiveness with passages as a baseline.

	Recall	Precision at N documents			Avg.	%Δ
		5	10	20	prec.	
TO	49.4	0.4640	0.3860	0.3210	0.2081	0.0
TOS, 250k	44.2	0.4800	0.3860	0.3160	0.2022	−2.8
TOS, 50k	37.0	0.4560	0.3720	0.3000	0.1849	−11.0
DO	49.4	0.4640	0.3860	0.3210	0.2081	0.0

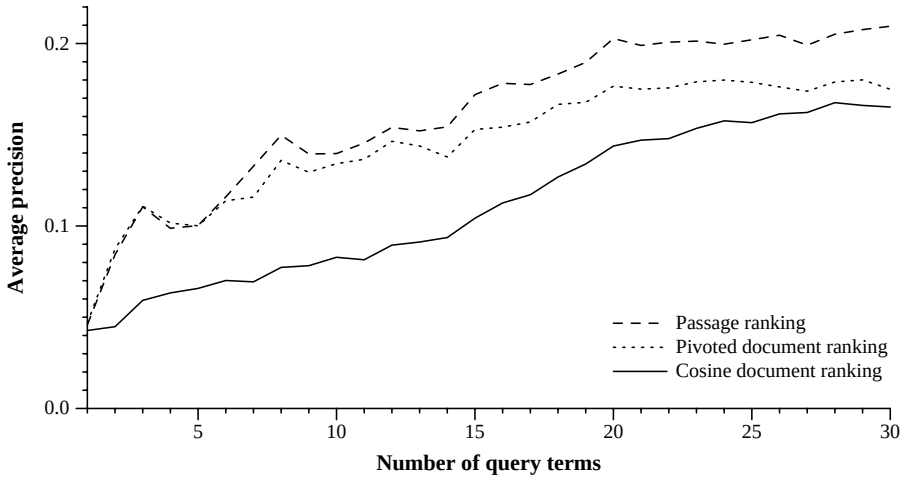

Figure 8: Effectiveness for document ranking and 150-word passage ranking on TREC-5 data, averaged over the 21 queries with 30 terms or more. Effectiveness for smaller numbers of query terms is determined by taking the first terms from the query.

these results are over a smaller query set than that used elsewhere in the paper, and with different average characteristics, so that as has happened here different average performance is likely to be observed.)

7 Combined DO and TOS evaluation

DO processing is efficient for short queries; TOS is efficient for long queries. We have therefore explored an optimisation that takes advantage of the properties of both processing methods.

In standard TOS query evaluation, processing consists of two stages: in the first, rare query terms are used to identify candidate passages; in the second, no new passages are identified, but the similarities of existing passages are updated. The first stage of TOS processing, which we have seen to be relatively expensive, can be replaced by a modified form of DO processing. In this DO-TOS strategy, DO processing on the rare query terms is used to identify candidate passages and TOS processing on the remaining query terms is used to refine the similarities of these passages. DO-TOS processing is shown in Figure 9; we omit the details of DO processing and TOS processing as these have already been discussed. Note that TOS processing is based on sequential processing of accumulators rather than of inverted lists, thus allowing skipping.

The DO-TOS approach is different from standard TOS processing in several ways, in particular because the set of terms to be used for identifying potential passages needs to be selected in advance, since their inverted lists are processed in parallel; in contrast, in TOS it is possible to

1. Let Q be the set of unique terms in the query.
2. Let Q_{DO} be a subset of Q, of terms selected for first-stage DO processing.
3. DO stage:
    (a) For each t in Q_{DO}: perform steps 4(a) and 4(b) from Figure 6.
    (b) Perform Step 5 from Figure 6, but keep all passages, not just the best for each document.
4. Let Q_{TOS} be the remaining terms from Q, that is, $Q - Q_{DO}$.
5. TOS stage:
    (a) Let A_p be the set of accumlators for passages obtained by DO processing.
    (b) For each t in Q_{TOS}:
        i. Seek to and fetch inverted list I_t.
        ii. For each accumulator in A_p, extract the frequency of t from I_t, updating A_p and consuming I_t.

Figure 9: DO-TOS processing for passage ranking.

move to the second stage at any time. Also, DO-TOS allows use of heuristics that have the potential to reduce costs without serious impact on effectiveness. For example, in the DO stage, instead of recording every passage in which any of the rare query terms occur, only the best passages in each document need be kept. Since high-ranking passages are likely to contain several occurrences of rare query terms, passages that are less promising can be discarded. An efficient way of implementing a strategy of this kind is to keep only the passages whose first word is a query term occurrence. Such a strategy may have some impact on effectiveness, but does yield faster evaluation. More sophisticated strategies, using for example the density of query term occurrences, would also be possible, but we have not experimented with them. Without such heuristics, DO-TOS is likely to be little faster than pure TOS processing.

In the DO-TOS approach a set of terms needs to be selected before DO evaluation begins. These terms identify the set of passages to be fully evaluated in the second stage of ranking. Since passages are shorter than documents, most of the similarity contribution is likely to be made by highly weighted terms. These terms lead to efficient DO evaluation since they are likely to have short inverted lists. Thus the size of this term set has a direct impact on the number of passages being ranked, affecting both efficiency and effectiveness.

We suggest that selection of terms for the first stage be guided by memory available for inverted lists. This size is likely to be smaller than the peak usage of TOS processing for inverted lists. For short queries, which are likely to include specific terms, any moderate buffer limit for inverted lists will accommodate an entire query. For longer queries, however, there is a trade-off between buffer size and the number of accumulators. For TREC, inverted lists of rare query terms occupy a few tens of kilobytes. A buffer space of a few hundred kilobytes should provide a good balance between buffer space for multiple inverted lists, reduced computation costs in DO stage, and overall retrieval effectiveness. Preliminary experiments showed that buffer space of around 250 Kb is acceptable for our data.

Figure 10 compares DO-TOS processing with the other passage ranking algorithms. In these experiments, a buffer of 250 Kb was used during the DO stage. For all query lengths, DO-TOS proved to be more efficient than TOS. For very short queries, DO-TOS and DO are equally efficient, but the benefits of skipping start to show as the algorithm moves to the TOS stage, when existing accumulators are only updated.

A further refinement is to explicitly limit the number of accumulators generated during the

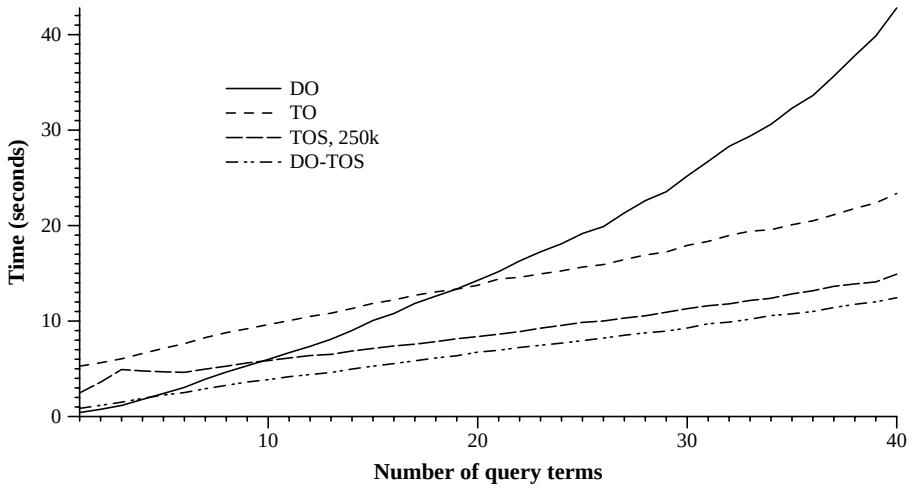

Figure 10: Passage ranking: Average elapsed time for DO-TOS evaluation for passages of 150 words, on TREC-5 data. Each "k" figure represents '000s of accumulators.

Table 4: Average per-query memory requirements for retrieval of 150-word passages with combined DO-TOS processing, on TREC-5 data. Each "k" figure represents '000s of accumulators.

	Accums (Mb)	Buffers for inv. lists (Mb)	Other (Mb)	Total (Mb)	No. of accums
DO-TOS	2.26	0.50	0.29	3.05	148,193
DO-TOS, 50k	0.76	0.50	0.29	1.55	50,000
DO-TOS, 25k	0.38	0.50	0.29	1.17	25,000

DO stage of evaluation, by creating a fixed-size heap and inserting passage accumulators into the heap as documents are processed. Elapsed times for this strategy, for limits of 25,000 and 50,000 accumulators, are shown in Figure 11. Memory requirements are shown in Table 4 and average-precision figures are shown in Table 5. Limiting accumulators reduces memory consumption but degrades retrieval effectiveness and increases processing time. DO-TOS, with or without limited accumulators, is less effective than the other techniques, but is significantly faster and more memory efficient. Note that 25,000 accumulators under DO-TOS occupy about the space needed for 10,000 accumulators under TOS, since for DO-TOS they are kept in an array rather than a dynamic structure such as a hash table.

Overall, a typical short query that required 0.5 seconds for document ranking can be evaluated in less than 2 seconds for passage ranking, even though the number of units to be ranked is twenty-five times greater. Likewise, a longer query that required 1.5 seconds for TOS document ranking can, with DO-TOS, be evaluated against passages in 6 seconds or so. These times are a considerable improvement on DO and TOS, which required 25 and 15 seconds respectively on a longer query, and show that passage ranking is indeed practical on a desktop machine.

It is interesting to note the differences between TOS with 50,000 accumulators and DO-TOS with 50,000 accumulators. As implemented here, with pure TOS evaluation more passages are generated per document, and fewer of the query terms have the opportunity to create a passage; with DO-TOS, any of the rare query terms can initiate creation of a passage. Comparing Tables 3 and 5, this strategy appears to lead to slightly greater effectiveness.

For whole-document ranking, DO-TOS processing is unlikely to yield significant performance improvements compared to TOS processing. One of the main advantages of DO-TOS passage ranking is that in the DO stage the in-document computation is somewhat simplified, with only

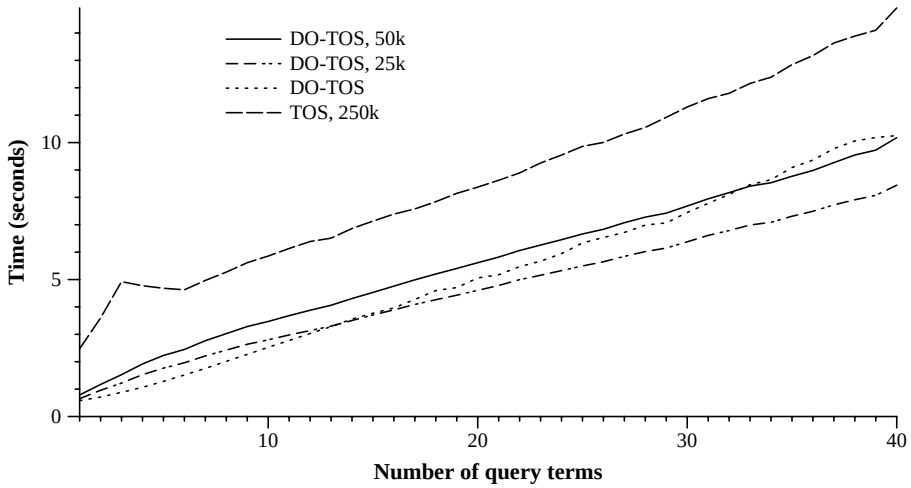

Figure 11: Passage ranking: Average elapsed time for DO-TOS query evaluation for passages of 150 words and limited accumulators, on TREC-5 data. Each "k" figure represents '000s of accumulators.

Table 5: Retrieval effectiveness for 150-word passages with combined DO-TOS processing, on TREC-5 data. Each "k" figure represents '000s of accumulators. Percentage changes in average precision are shown as "deltas" (%Δ), taking TO or DO effectiveness with passages as a baseline.

	Recall	Precision at N documents			Avg.	%Δ
		5	10	20	prec.	
TO or DO	49.5	0.4640	0.3860	0.3210	0.2081	0.0
DO-TOS	45.1	0.4680	0.3800	0.3040	0.1988	−4.5
DO-TOS, 50k	42.6	0.4600	0.3700	0.2970	0.1949	−6.3
DO-TOS, 25k	41.7	0.4400	0.3560	0.2880	0.1924	−7.5

one passage considered for each query term occurrence. It is this simplification—for which there is no equivalent in document ranking—that yields the increase in speed.

8 Frequency sorting for passage retrieval

Frequency-sorting is effective for whole-document ranking because each entry in each inverted list refers to a single unit of text to be ranked, and overall is the most efficient document ranking technique. In contrast, for passage ranking each entry pertains to multiple passages, and the in-passage frequency of a word can be different for each passage. Direct application of frequency-sorting to passage ranking would require an inverted list entry per passage, greatly increasing index storage requirements; and, since the distribution of in-passage frequencies is much narrower than the distribution of in-document frequencies, the benefit is likely to be limited. However, the strength of frequency-sorting for whole-document ranking means that it is worth investigating for passages.

We have explored a variation on frequency sorting that has the potential to provide some efficiency gains for passage retrieval. Heuristically, it is reasonable to suppose that documents in which a rare term is frequent will have passages in which the term is frequent. On this basis, an index that is frequency-sorted according to within-document frequency may yield efficiency gains, as the documents at the start of each inverted list should contain the best passages.

A possible query evaluation strategy is then as follows: fetch the document information for

the document with the highest partial similarity (determined by term weight and in-document frequency); extract per-passage information for that term from the document; and update the accumulators of each of these passages.

An alternative is to strictly process information in order of decreasing partial passage similarity. At each stage, we know the weight of each term and its frequency in the next document containing that term. We can also keep a heap of unprocessed passage information, ordered by decreasing partial similarity. The frequency of the term in the next document is an upper bound on that term's in-passage frequency for any remaining passage. We can then fetch the next document entry from the inverted list for that term, extract all passage information for that term, find the actual in-passage frequencies, and use these to compute partial passage similarities for that term. The partial similarities that exceed those in the heap, or those that may be found by processing further inverted list information (hence the use of the upper bound determined by in-document frequency), can be immediately used to update accumulators. The remaining partial similarities can be added to the heap.

We have explored these strategies for our data but neither appears likely to be effective. First, the correlation between in-document frequency and in-passage frequency within the document is fairly weak. Second, the distribution of in-passage frequencies is narrow, so the per-passage partial similarities are not discriminating. In both strategies the bulk of the inverted list information still has to be processed—so that query evaluation is less efficient than with skipping—and large numbers of accumulators must be maintained.

9 Conclusions

Passage retrieval is useful for collections of documents that are of varying length or are simply very long, and for text collections in which there are no clear divisions into individual documents. The more effective passage retrieval techniques require that large number of individual entities be considered for ranking at query evaluation time; in this paper we have shown that such passage ranking of large collections of documents is feasible on ordinary hardware.

Our exploration of efficient passage ranking has required that we take a fresh look at the query evaluation techniques used for document ranking. We have shown that, assuming inverted lists are fetched in full and buffered until needed, that document-ordered and skipped term-ordered document ranking are about equally efficient for queries of a couple of terms but that skipped evaluation is much more efficient for longer queries. Both skipped evaluation and document-ordered evaluation can run in strict space limits; however, overly strict limits are likely to cause document-ordered evaluation to thrash, while, as space is reduced, skipped evaluation becomes faster but less effective. These trade-offs gradually change as collection size grows, but, even with an artificial collection of around 7.7 million small documents, document-ordered processing is only more efficient for queries of a few words or less.

For passage ranking, however, different trade-offs emerge. Document-ordered passage ranking is somewhat simpler than term-ordered passage ranking, allowing considerable efficiency gains for short queries. On the other hand, for long queries the cost of document-ordered ranking grows rapidly. Our results show that for short queries document-ordered ranking is clearly preferable, given the reduced processing cost and the ability with this method to smoothly reduce the space requirements. For both evaluation methods we have proposed substantial revisions to the standard query evaluation algorithms, to allow efficient processing of overlapping passages.

These results suggested a new "DO-TOS" processing method, combining the advantages of document-ordered and term-ordered processing. The former is applied to information about rare terms to identify likely passages; the latter is used to complete the scoring of these passages using information about common terms. We have shown experimentally that this new technique is efficient for all query lengths; is as fast or faster than document-ordered or term-ordered processing and requires less memory, although it is not quite as effective; like skipped evaluation, becomes faster but less effective as buffer size is reduced; and, given its low resource requirements and good asymptotic characteristics, should scale well. On our test collections passage ranking requires

only a four-fold increase in evaluation time compared to ranking documents with comparable algorithms (but note that other, faster ranking techniques such as frequency ordering can be used for documents but not for passages). Overall, passage ranking is practical on a desktop machine, with limited-memory evaluation of typical short queries on a 2 Gb collection taking only around one second.

While we have described these techniques in terms of their impact on effectiveness, they can also be viewed in terms of more traditional query evaluation constraints such as available buffer space. All the major methods we have described—skipped term-ordered evaluation, document-ordered evaluation, and the DO-TOS combination of these—can be tuned to available system resources. Within limits, this tuning only marginally degrades retrieval effectiveness.

Acknowledgements

This work was supported by the Australian Research Council. We are grateful to the anonymous referees for their extensive comments and feedback.

References

[1] J. Allan. Relevance feedback with too much data. In E.A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the 18th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 337–343, July 1995.

[2] V.N. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In W.B. Croft, A. Moffat, C.J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proceedings of the 21st Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 290–297, Melbourne, Australia, August 1998.

[3] T.C. Bell, A. Moffat, I.H. Witten, and J. Zobel. The MG retrieval system: Compressing for space and speed. *Communications of the ACM*, 38(4):41–42, April 1995.

[4] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and B. Catania. Text databases. In *Indexing Techniques for Advanced Database Systems*, Boston, Massachusetts, 1997. Kluwer Academic Press.

[5] E. W. Brown. Fast evaluation of structured queries for information retrieval. In E.A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the 18th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 30–38, July 1995.

[6] C. Buckley and A. F. Lewit. Optimisation of inverted vector searches. In *Proceedings of the Eighth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 97–110, June 1985.

[7] J.P. Callan. Passage-retrieval evidence in document retrieval. In W.B. Croft and C.J. van Rijsbergen, editors, *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 302–310, July 1994.

[8] C. Clarke, G. Cormack, and F. Burkowski. Shortest substring ranking (MultiText experiments for TREC-4). In D.K. Harman, editor, *Proceedings of the Fourth Text REtrieval Conference (TREC-4)*, pages 295–304, November 1995.

[9] C. Clarke, G. Cormack, and E. Tudhope. Relevance ranking for one to three term queries. In *Proceedings of the Fifth RIAO Conference*, pages 388–412, June 1997.

[10] G. Cormack, C. Palmer, M. Biesbrouck, and C. Clarke. Deriving very short queries for high precision and recall (MultiText experiments for TREC-7). In E.M. Voorhees and D.K. Harman, editors, *Proceedings of the Seventh Text REtrieval Conference (TREC-7)*, 1998. To appear.

[11] W. B. Frakes and D. C. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[12] M. Fuller, M. Kaszkiel, Dongki Kim, C.L. Ng, J. Robertson, R. Wilkinson, M. Wu, and J. Zobel. TREC 7 ad hoc, speech, and interactive tracks at MDS/CSIRO. In E.M. Voorhees and D.K. Harman, editors, *Proceedings of the Seventh Text REtrieval Conference (TREC-7)*, 1998.

[13] M. Fuller, M. Kaszkiel, C.L. Ng, P. Vines, R. Wilkinson, and J. Zobel. MDS TREC 6 report. In E.M. Voorhees and D.K. Harman, editors, *Proceedings of the Sixth Text REtrieval Conference (TREC-6)*, pages 241–258, November 1997.

[14] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing & Management*, 31(3):271–289, 1995.

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

[16] M.A. Hearst and C. Plaunt. Subtopic structuring for full-length document access. In R. Korfhage, E. Rasmussen, and P. Willet, editors, *Proceedings of the Sixteenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 59–68, June 1993.

[17] M. Kaszkiel and J. Zobel. Passage retrieval revisited. In N.J. Belkin, D. Narasimhalu, and P. Willett, editors, *Proceedings of the 20th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 178–185, July 1997.

[18] M. Mitra, A. Singhal, and C. Buckley. Improving automatic query expansion. In W.B. Croft, A. Moffat, C.J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proceedings of the 21st Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 206–214, August 1998.

[19] E. Mittendorf and P. Schäuble. Document and passage retrieval based on hidden Markov models. In W.B. Croft and C.J. van Rijsbergen, editors, *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 318–327, July 1994.

[20] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[21] A. Moffat, J. Zobel, and S.T. Klein. Improved inverted file processing for large text databases. In R. Sacks-Davis and J. Zobel, editors, *Proc. 6'th Australasian Database Conference*, pages 162–171, Adelaide, January 1995.

[22] M. Persin. Efficient implementation of text retrieval techniques. Technical report, Melbourne, Australia, 1996. Thesis.

[23] 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.

[24] G. Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer*. Addison-Wesley, 1989.

[25] G. Salton, J. Allan, and C. Buckley. Automatic structuring and retrieval of large text files. *Communications of the ACM*, 37(2):97–108, 1994.

[26] G. Salton and C. Buckley. Automatic text structuring and retrieval—experiments in automatic encyclopedia searching. In G. Salton A. Bookstein, Y. Chiaramella and V.V Raghavan, editors, *Proceedings of the Fourteenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 21–30, October 1991.

[27] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[28] A. Singhal, C. Buckley, and M. Mitra. Pivoted document length normalization. In H.-P. Frei, D. Harman, P. Schäuble, and R. Wilkinson, editors, *Proceedings of the 19th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 21–29, August 1996.

[29] C. Stanfill and D.L. Waltz. Statistical methods, artificial intelligence, and information retrieval. In Paul S. Jacobs, editor, *Text-Based Intelligent Systems: Current Research and Practice in Information Extraction and Retrieval*, pages 215–225. Lawrence Erlbaum Associates, 1992.

[30] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[31] E. Voorhees and D. Harman. Overview of the Fifth Text REtrieval Conference (TREC-5). In D.K. Harman, editor, *Proceedings of the Fifth Text REtrieval Conference (TREC-5)*, pages 1–28, 1996.

[32] S. Walker, S.E. Robertson, M. Boughanem, G.J.F. Jones, and K. Sparck Jones. Okapi at TREC-6 Automatic ad hoc, VLC, routing, filtering and QSDR. In E.M. Voorhees and D.K. Harman, editors, *Proceedings of the Sixth Text REtrieval Conference (TREC-6)*, pages 125–136, November 1997.

[33] R. Wilkinson. Effective retrieval of structured documents. In W.B. Croft and C.J. van Rijsbergen, editors, *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 311–317, July 1994.

[34] H. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*. To appear.

[35] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and indexing documents and images*. Van Nostrand Reinhold, 1994.

[36] J. Xu and B. Croft. Query expansion using local and global document analysis. In H.-P. Frei, D. Harman, P. Schäuble, and R. Wilkinson, editors, *Proceedings of the 19th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 4–11, August 1996.

[37] J. Zobel and A. Moffat. Exploring the similarity space. *SIGIR Forum*, 32(1):18–34, Spring 1998.

[38] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 1998. To appear.

[39] J. Zobel, A. Moffat, R. Wilkinson, and R. Sacks-Davis. Efficient retrieval of partial documents. *Information Processing and Management*, 31(3):361–377, 1995.