Optimised Phrase Querying and Browsing of Large Text Databases

Dirk Bahle dirk@mds.rmit.edu.au
Hugh E. Williams hugh@cs.rmit.edu.au
Justin Zobel jz@cs.rmit.edu.au
Department of Computer Science, RMIT University
GPO Box 2476V, Melbourne 3001, Australia

Abstract

Most search systems for querying large document collections—for example, web search engines—are based on well-understood information retrieval principles. These systems are both efficient and effective in finding answers to many user information needs, expressed through informal ranked or structured Boolean queries. Phrase querying and browsing are additional techniques that can augment or replace conventional querying tools. In this paper, we propose optimisations for phrase querying with a nextword index, an efficient structure for phrase-based searching. We show that careful consideration of which search terms are evaluated in a query plan and optimisation of the order of evaluation of the plan can reduce query evaluation costs by more than a factor of five. We conclude that, for phrase querying and browsing with nextword indexes, an ordered query plan should be used for all browsing and querying. Moreover, we show that optimised phrase querying is practical on large text collections.

1 Introduction

On the world-wide web, document databases of more than one billion documents[1] are searched more than 94 million times per day.[2] Users expect all documents to be stored online and to be readily able to locate documents in response to simple queries. Web search engines are based on well-understood information retrieval principles. They are efficient—that is, able to find documents quickly—when users pose informal ranked queries or Boolean queries to return a list of documents as answers. And they are effective, that is, able on average to find documents that satisfy users' information needs [8].

For some alternative query types, conventional information retrieval systems are less efficient. For example, conventional systems are not optimised to evaluate phrase queries, where the order and adjacency of words are important. When posed to a search engine, phrase queries are distinguised by quotation marks; the phrase query "Richmond Football Club" only returns matches that contain the exact quoted phrase. Such query types are important: web search engine databases are growing in size and new techniques to refine information needs are becoming more important.

While conventional systems are not optimised for phrase queries, there are other query types—such as phrase browsing—they do not support at all. Phrase browsing permits a user to explore a document collection by providing a phrase and exploring the words that occur in the context of that phrase. Consider a user who begins browsing with the word "Richmond". Through browsing, the user may discover that selected words such as "FC", "Football", "precinct", "station", and "Tigers" can all be added to the word "Richmond" to form a two-word phrase. A two-word phrase such as "Richmond station" might in turn be able to be extended to "Richmond station timetable" or "Richmond station platform". After refining a phrase, a user could return to conventional querying to formulate a better informal query, or the user could retrieve documents containing the browsed phrase.

We have previously proposed efficient data structures for

[1]From `http://www.google.com/`

[2]From `http://www.searchenginewatch.com/`

1530-0900/01 $10.00 © 2001 IEEE

special-purpose phrase querying and browsing [10]. We have shown that these structures can permit phrase searching that is two to four times faster than with an efficient conventional system. Phrase browsing is much slower: for short two-word phrases, browsing is between three and thirty times slower than phrase querying, while for longer five-word phrases it is between one-and-a-half and four times slower.

In this paper, we propose and test optimisations to improve the efficiency of phrase querying and browsing. We show that generating query plans and optimising the order in which words are processed can reduce phrase querying costs by more than a factor of five over a naive approach. With these optimisations, phrase querying and browsing is practical on large text collections and can be used in most situations where conventional querying techniques are applied.

2 Searching Text Databases

Information retrieval (IR) systems—web search engines are the best known—are most often used to resolve informal or *ranked* queries. Ranked queries are typically a bag of words, and the answers are the documents in a text repository that have the highest estimated statistical likelihood of being perceived as relevant to the query [8] [11]. For example, a query such as

> 1980 Richmond premiership victory Bartlett football VFL

would be statistically compared to each document in the text database, using properties including the frequency of query words in the documents and the relative rareness of the query words.

Query evaluation in an efficient IR system is supported by an inverted index consisting of a vocabulary and, for each term in the vocabulary, a list of information about where the term occurs [1] [11]. The simplest inverted index structure for evaluating ranked queries has, for each word in the vocabulary, a list of documents that contain that word and a count of occurrences of that word in each document. The inverted list for the term "Richmond" might have the structure:

1: 8; **2**: 11; **1**: 100; ...

showing that the word "Richmond" occurs once in document 8, twice in document 11, once in document 100, and so on. With the document numbers in sorted order, and the frequencies typically being small integers, this data can be compressed to less than 10% of the size of the indexed data [9] [11].

Ranked query evaluation with an inverted index proceeds as follows. First, each of the query terms is searched for in the in-memory vocabulary. Second, the inverted list of frequencies and documents in which each query term occurs is retrieved from disk. Third, a similarity between the query and each of the documents containing the query terms is calculated, with the similarities stored in accumulators. Last, the top-scoring documents—or short summaries—are retrieved from disk and presented in ranked order to the user, requiring a two-step lookup through a *mapping table* that maps document numbers to physical disk offsets.

This inverted index structure can also be used to evaluate Boolean queries. For example, the Boolean query:

> (1980 AND premiership) OR Richmond

can be evaluated through intersecting the two lists of documents for the words "1980" and "premiership", followed by taking the union of the result with the list of documents for "Richmond". To support only Boolean querying, the document frequency need not be stored.

Ranked and Boolean retrieval are sufficient to meet many information needs. However, addition of phrases to either query mode can more clearly specify an information need and has been shown to improve effectiveness [2] [12]. For example, a ranked phrase query:

> "Richmond Football Club" premiership 1980

contains three terms, one of which is a phrase. Another technique is phrase browsing, where terms in the vocabulary are explored in the context in which they occur in the database [3] [7] [10].

To support phrase querying, word positions must be stored in the index. For example, the inverted list for the term "Richmond" might be:

1: 8; 22 **2**: 11; 7, 44 **1**: 100; 12 ...

showing that the term occurs once in document 8 as the 22nd word, twice in document 11 as the 7th and 44th words, once in document 100 as the 12th word, and so on. Phrase browsing cannot be supported with this structure; we describe a structure that supports phrase browsing in the next section.

To evaluate the phrase term of our phrase query—"Richmond Football Club"—we read in the above list for the word "Richmond" and create an in-memory list for further processing. Next, we read in the inverted list for "Football" and merge this list with our in-memory list. If the "Football" list has the following structure:

1: 7; 12 **2**: 11; 8, 45 **1**: 100; 3 ...

our merging process will identify that the phrase does not occur in documents 7, 8, or 100. The partial phrase "Richmond Football" does occur in document 11 beginning at positions 7 and 44. Depending on the "club" list, we may identify the complete phrase at either of these offsets.

In the next section, we describe a structure we have previously proposed to support more efficient phrase querying. Moreover, the structure supports phrase browsing, which is not possible with the structures described in this section.

3 Phrase Querying and Browsing

We have previously described the *nextword* index structure for phrase querying and browsing [10]. A nextword index stores the words that occur in a collection and, for each such word, the words that immediately follow that word anywhere in the collection. Stored interleaved with the following words—the nextwords—are pointers to inverted lists that store the documents and offsets of the two-word phrases in the collection.

Consider an example entry in a nextword index as shown in Figure 1. The top of the figure represents the in-memory vocabulary of the collection; selected words in the range "rich" to "ride" are shown. Each word is interleaved with pointers to the disk positions of nextword lists. The nextwords for the term "Richmond"—which occurs, say, twenty times in a document collection—are shown in the middle section of the figure. There are six nextwords: "FC", "Football", "precinct", "premiership", "station", and "Tigers". Interleaved with the nextwords are pointers to inverted lists. The bottom of the figure shows that the phrase "Richmond premiership" occurs four times in three documents, 6, 12, and 47; the vector structure is as described in Section 2.

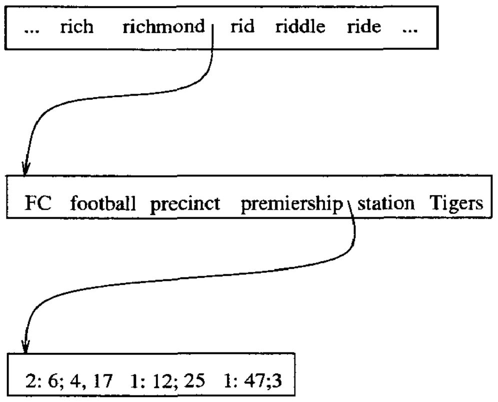
... rich richmond rid riddle ride ...
FC football precinct premiership station Tigers
2: 6; 4, 17 1: 12; 25 1: 47;3

Figure 1. *Organisation of a nextword index.*

As words and nextwords are sorted, each can be compactly stored using front-coding. With frontcoding, each word is stored as two integers and a suffix. For example, the nextword "premiership" can be stored as "3,8,miership" since it has a three-character prefix in common with "precinct" and an eight-character suffix "miership". The inverted lists for each word-nextword pair are stored compressed using Golomb and Elias variable-bit integer coding schemes [4] [5] [9] [11].

Phrase querying with a two-word query "Richmond premiership" and a nextword index proceeds as follows. First, the word "Richmond" is looked-up in the in-memory vocabulary and the disk position of its nextword list retrieved. Second, the nextword list is decoded, searching for the word "premiership". Third, when "premiership" is found, the disk position of the inverted list for the phrase "Richmond premiership" is retrieved. Fourth, the inverted list for the phrase is retrieved and decoded. Last, the three documents containing four occurrences of the phrase can be retrieved and presented to the user.

For longer phrase queries, nextword inverted lists are merged in the same way as inverted lists are merged for phrase querying with conventional structures. An optimisation with nextword indexes for longer phrase queries is to determine an evaluation order where rare pairs are processed first to create the shortest possible in-memory lists. More importantly, processing rare pairs first may permit fast early termination of querying when a phrase does not occur in the collection. We discuss this in detail in the next section.

Phrase querying with a nextword index is almost always more efficient than with a conventional inverted index, with most phrase queries being evaluated four times as fast as with a conventional inverted index [10]. The only exception is where the first word of a pair is a common word (such as "the") and the second word is a rare word (such as "aardvark"). In this case, the nextword list of "the" is long—only slightly shorter than a regular inverted list—and querying with a nextword index is just over two times faster than with a conventional structure.

Nextword indexes also support monodirectional phrase browsing. Given a word, the nextword list can be used to identify all the words that follow that word without retrieving any inverted lists. More generally, for a longer phrase, the inverted lists for the nextwords following the last word in the phrase can be retrieved. These lists then can be checked against the in-memory list, and the words that follow the phrase shown to the user. This process of phrase browsing is much slower than phrase querying and highly dependent on the length of the nextword list for the final word in the phrase. In our experiments, we found that if the final word is common, phrase browsing can be up to 20 times slower than querying, while if the word is rare then phrase browsing is less than 3 times slower.

In the next section, we present new optimisations to further improve the efficiency of nextword-based phrase querying and browsing.

4 Optimised Phrase Browsing and Querying

As we discussed in the last section, we have previously observed that careful query plan generation for longer phrase queries is crucial to efficiency. In this section, we propose four alternatives for query plan generation and evaluation.

There are two aspects to query plan generation: first, the selection of which word-nextword pairs are to be evaluated; and, second, the order of evaluation of the selected word-pairs. We begin by discussing selection of word-nextword pairs, and return to evaluation order later in this section.

Choosing word-pairs

For a two-word phrase, query resolution is simple and unambiguous. A two-word query—such as "Richmond Tigers"—is resolved by retrieving a single inverted list for the word-nextword pair and decoding this list. A three word query requires that two lists are retrieved, that is, inverted lists for both word-nextword pairs in the query must be evaluated to determine the documents that contain the query phrase. For example, the query "1980 year of" requires the retrieval and merging of the two inverted lists for the word-pairs "1980 year" and "year of".

For query phrases longer than three words, selected word-pairs need not be evaluated. A four-word query—such as "1980 year of the"—requires only the retrieval and merging of two inverted lists, in this case for the word-pairs "1980 year" and "of the". Only two word-pairs need to be evaluated since, if "of the" occurs at an offset of two after "1980 year", we know the phrase occurs in the document. We do not need to retrieve the list for "year of" since, if "of the" follows "1980 year", then "year" and "of" are adjacent. More generally, to resolve a query, each word in phrase need only be evaluated as a member of one word-pair, that is, as either a word or a nextword. The exception, as we have seen for a three-word query, is when the query is of an odd-length there is one overlapping word-pair.

For a five-word or longer odd-length query, we have choice as to which nextword pairs are evaluated. For example, for the five-word query "1980 year of the Tiger" there are two possible query plans. We could choose to evaluate:

"1980 year", "of the", and "the Tiger"

or we could choose to evaluate:

"1980 year", "year of", and "the Tiger"

Both plans are complete since they both evaluate all words as either the first or second word in a word-pair. The difference in the plans is which word-pair is chosen as the overlapping word pair for the odd-length query.

A *naive query plan* generator would choose the first alternative, that is, it would choose non-overlapping pairs from left-to-right, and finish by choosing the overlapped pair "of the" and "the Tiger"; we report experiments with naive query plans in the next section. We might speculate that the second query plan would be more efficient, since it involves evaluating the pair "year of"—which is likely to have a short inverted list—and avoids the pair "of the"—which is likely to require retrieval and merging of a much longer inverted list.

A simple metric for estimating the cost of a query plan is to sum the nextwords for each first-word in a pair; there are other alternatives, but nextword-count is efficient since it is held in memory with the vocabulary. In our example, the first plan may have a cost of 20 nextwords for "1980", plus a cost of 10,000 nextwords for "of", and 40,000 nextwords for "the", giving a total of 50,020. The second plan may have a cost of 20 nextwords for "1980", plus a cost of 70 nextwords of "year", and 40,000 nextwords for "the", giving a total of 40,090. With this minimum query plan metric, the second query plan is cheaper.

More generally, for an odd-length query of length n, there are $\lfloor n/2 \rfloor$ possible *minimum query plans*, that is, query plans with the same minimum count of inverted lists that must be retrieved. For each such minimum query plan, $\lfloor n/2 \rfloor + 1$ word-pairs must be evaluated. For an even-length query, there is exactly one minimum query plan that is identical to the naive plan, since an overlapping word-pair need not be evaluated; an even-length query plan requires that $n/2$ word-pairs are evaluated. We report experiments in the next section with a minimum query plan generator that chooses a query plan with the smallest sum of nextwords.

Interestingly, evaluating more than the minimum set of word-pairs in both odd and even queries can offer faster query evaluation. Consider an eight-word query "Richmond FC then won often in about 1995". We may speculate that the best query plan for this query would avoid the long lists for "then" and "in" by avoiding the "then won" and "in about" word pairs. To do this in our example, we need to evaluate two overlapping pairs: first, "Richmond FC" and "FC then"; and, second, "won often" and "often in". To complete the query, we also need to evaluate "about 1995" giving a total of five pairs in the query plan while the minimum is four.

We call this scheme of avoiding word-pairs with high nextword frequencies an *ordered query plan*. In this scheme, word-pairs are ordered from least-nextwords to most-nextwords and pairs added to the query plan until all words are members of at least one word-pair. In the worst case, for a query of length n, a total of $n - 1$ word-pairs may be evaluated. We report experiments with the ordered query plan generator in the next section.

Evaluation order of query plans

Evaluation order of query plans is also important. If a phrase does not occur in the collection, and we can identify this without processing the complete query plan, then early query termination is possible.

For all query plan generators, we first check if the words in the query occur in the collection. If any word does not occur, we can report that the phrase does not occur. If all terms do occur in the collection, we can then sort the word-pairs selected as members of the query plan from least-nextwords to most-nextwords and evaluate the pairs in that order. This permits two efficiencies: first, the in-memory list will be as short as possible—since it is created from the shortest nextword list in the phrase—and, second, it permits fast detection of a phrase that does not occur through comparison of the offsets of rare pairs that are less likely to occur in proximity by chance.

As an example we return to our phrase query "Richmond FC then won often in about 1995" and assume that each of the eight words occur in our collection. For this even-length query—assuming we use a naive or minimum query plan—we might elect to process the pairs:

> "Richmond FC", "then won", "often in", and "about 1995"

We would then sort the query plan from lowest first-word nextword count to highest:

> "Richmond FC", "often in", "about 1995", "then won"

and begin by evaluating the rarest pair, "Richmond FC". We retrieve the inverted list—which is the shortest of all word-pairs in the query—and create an in-memory list. To this list, we merge the list of "often in". If "often in" does not occur immediately after "Richmond FC" in any document, we can report to the user the phrase does not occur.

For phrase browsing, query plan generation is similar. The significant difference is that we must retrieve the nextwords of the final word in the phrase, meaning that from a plan generation perspective an even-length phrase becomes an odd-length phrase, and an odd-length phrase an even one.

5 Experiments

In our experiments, we used 981 megabytes of words[3] extracted from the TREC Very Large Collection web data (WEB) and 508 megabytes of the Wall Street Journal (WSJ) from TREC disks 1 and 2 [6]. TREC is an ongoing international collaborative information retrieval experiment sponsored by the NIST and ARPA. The nextword index for WSJ requires 278 MB of disk space or 56% of the collection size, while the WEB index requires 696 MB or 71% of the collection size; these indexes are large compared to conventional indexes, while still being practical. Details of the index size, construction, and compression can be found elsewhere [10].

Queries for our query plan generation experiments were drawn from both WSJ and WEB. For both collections, we carried out the same three-step process:

1. We extracted 100 random ten-word queries beginning with a common word; a common word was one of the 100 words with the highest nextword count.
2. We then extracted 100 random ten-word queries beginning with a medium-frequency word; a medium-frequency word was one of the 100 words around the median nextword frequency in the collection.
3. Last, we extracted 100 random ten-word queries beginning with a rare word; a rare word is a word with only one nextword.

After completing this process, we had 600 queries in six classes, where half were ten-word queries from WSJ and half were ten-word queries from WEB. We call these six classes COMMON-WEB, MEDIUM-WEB, RARE-WEB, COMMON-WSJ, MEDIUM-WSJ, and RARE-WSJ.

We then took each of the 600 ten-word queries and produced 600 nine-word queries by removing the last word from each query. We then produced 600 eight-word queries by removing the last word from the nine-word queries, and so on, until we had two-word queries. The final result was 9 sets of 100 queries in each of the six classes described above, that is, 100 queries of each length between 2 and 10 in each class. In total we had 5,400 queries.

Before running each set of 100 queries, we flushed all system caches for a "cold start". That is, we ensured that for the first query in each set all index data is fetched from disk. After that, index entries may be cached for the 99 remaining queries in the set. All experiments are carried out on an Intel Pentium III-based machine with 256 Mb of RAM under light-load; all reported measurements are of elapsed time for index processing, without fetching of answer documents.

Figure 2 shows the average time taken to evaluate all COMMON-WEB queries of lengths two to ten with the naive unsorted, naive sorted, ordered (by definition sorted), and minimum sorted query plan generators on the WEB collection. The results are not surprising: each of the queries occurs in the collection, so early termination of querying is not possible and the schemes perform similarly. Overall, the ordered query plan generation is more efficient for longer queries, since in some cases processing additional overlapped lists permits avoidance of word-pairs with long inverted lists. As expected, the minimum scheme chooses better overlaps than the naive scheme for odd-length queries, but is identical for those of even-length. Sorting the naive plan makes little difference, with the only significant benefit that a sorted query plan creates a smaller initial in-memory inverted list. We observed similar results with the MEDIUM-WEB and RARE-WEB queries on WEB, as again with the COMMON-WSJ, MEDIUM-WSJ, and RARE-WSJ queries on WSJ; we do not report these results in detail here.

Figure 3 shows the average time taken to evaluate all COMMON-WSJ queries of lengths two to ten with the naive unsorted, naive sorted, ordered, and minimum query plan generators on the WEB collection. Many of the COMMON-WSJ queries do not occur in the WEB collection and, therefore, early query termination is possible in many cases, producing contrasting results with those of Figure 2. For four-word queries, the ordered scheme is around 15% faster than either naive approach, while for longer queries ordered is more than five times faster than the naive unsorted approach and twice as fast as the naive sorted scheme.

Figure 3 shows the naive unsorted evaluation times are roughly constant for queries of length three or more, suggesting most of the cost is in evaluating the commonly-

[3]We removed non-words, including punctuation, special characters, SGML and HTML markup, sequences of ASCII characters containing more than two integers or beginning with an integer, and sequences of ASCII characters containing special characters. As an example, the word "don't" is represented as "dont", while non-words such as "1996", "08-362-3106", and "27th" are removed.

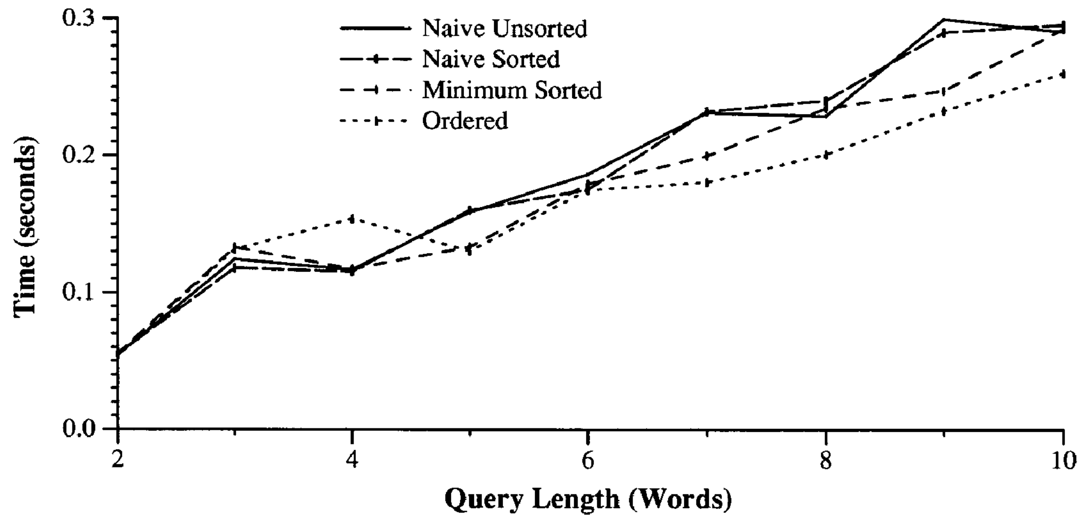

Figure 2. *Average time taken to evaluate the* COMMON-WEB *queries of lengths two to ten (3,600 in total) with the Naive unsorted, Naive sorted, Ordered (sorted), and Minimum sorted query plan generators on the* WEB *Collection.*

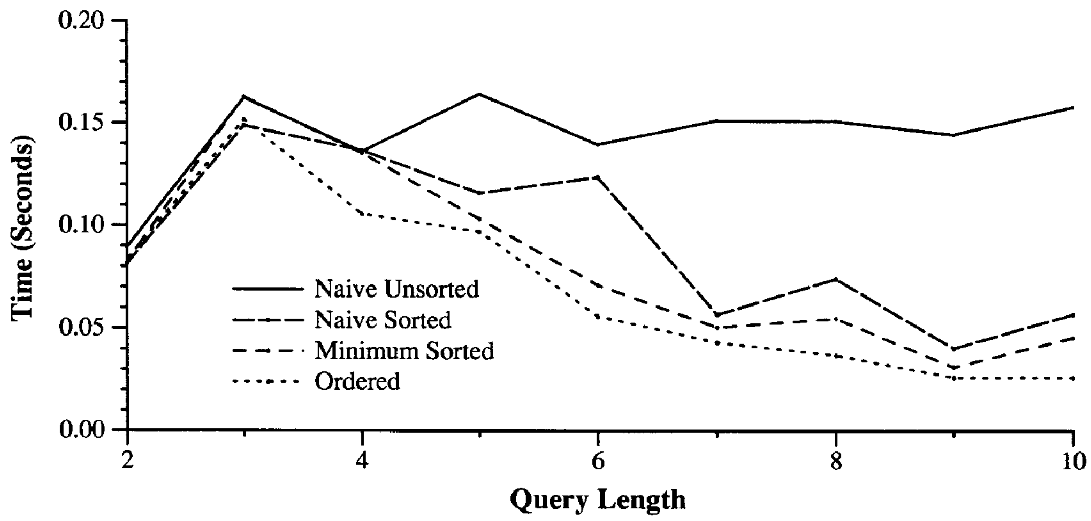

Figure 3. *Average time taken to evaluate the* COMMON-WSJ *queries of lengths two to ten (3,600 in total) with the Naive unsorted, Naive sorted, Ordered, and Minimum sorted query plan generators on the* WEB *collection.*

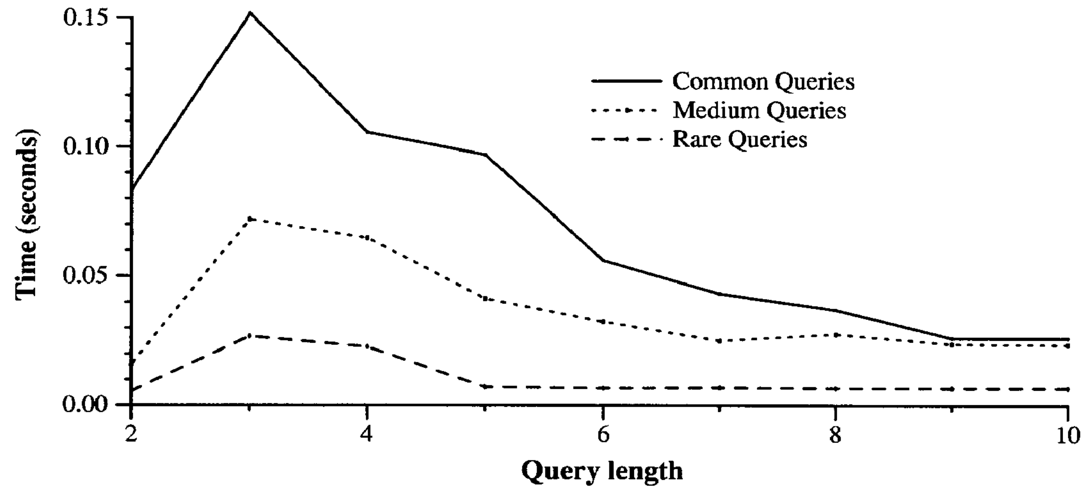

Figure 4. *Average time taken to evaluate the* COMMON-WSJ, MEDIUM-WSJ, *and* RARE-WSJ *queries with the ordered query plan generator on the* WEB *Collection.*

occuring first word and that termination usually occurs after a few words. For the ordered, minimum sorted, and naive sorted query plans, average query times fall with increasing query length, as a longer query increases the likelihood of finding a term that does not occur in the collection, careful selection of word-pairs becomes possible, and optimisation of processing order permits early termination. Overall, sorting a query plan has the most significant impact on query cost, while choosing an ordered query plan offers some additional improvement.

The ordered approach results are slightly better than the minimum results, suggesting that selecting more overlapping rare pairs permits earlier termination and that number of queries is less important than list length. The query costs for the sorted schemes are slightly less than the naive scheme for queries of length three, since a shorter initial in-memory list is created.

A comparison of the average time for evaluating the COMMON-WSJ, MEDIUM-WSJ, and RARE-WSJ query types on the WEB collection with the ordered scheme is shown in Figure 4. The most striking difference in the curves shown is for shorter queries: this is not surprising since a short COMMON-WSJ query includes a word such as "the" that requires retrieval of a long nextword list, while a RARE-WSJ query requires only a short list. As query length grows, the ordered plan permits fast identification of rare pairs and termination, regardless of the frequency of the first word.

6 Conclusions

The special-purpose nextword index structure supports fast phrase querying and practical phrase browsing on large text collections, allowing users to find documents that would be difficult to locate with other mechanisms.

We have shown that optimised query planning and query evaluation for querying with a nextword index, can reduce query times by a factor of five over the original, naive query evaluation scheme. With our ordered query plan, the cost of evaluating two to ten word phrase queries on a collection of almost one gigabyte is between 0.1 seconds and 0.3 seconds. Moreover, optimised phrase querying reduces the costs of nextword-based phrase browsing. Optimised phrase querying with nextword indexes is as practical as ranked or Boolean querying on large text collections.

We are currently investigating techniques to better optimise common-word queries. As part of this work, we are developing new methods of compressing lists.

Acknowledgements

This work was carried out in the Multimedia Database Systems group at RMIT University.

References

[1] E. Bertino, B. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems.* Kluwer Academic Publishers, 1997.

[2] J. Callan, W. Croft, and J. Broglio. TREC and TIPSTER experiments with INQUERY. *Information Processing and Management*, 31(3):327–343, 1995.

[3] S. Dennis, R. McArthur, and P. Bruza. Searching the world wide web made easy? the cognitive load imposed by query refinement mechanisms. In J. Kay and M. Milosavljevic, editors, *Proc. Australian Document Computing Conference*, pages 65–71, Sydney, Australia, 1998. University of Sydney.

[4] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, Mar. 1975.

[5] S. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT–12(3):399–401, July 1966.

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

[7] C. Nevill-Manning and I. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 40(2/3):103–116, 1997.

[8] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, Massachusetts, 1989.

[9] H. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.

[10] H. Williams, J. Zobel, and P. Anderson. What's next? Index structures for efficient phrase querying. In J. Roddick, editor, *Proc. Australasian Database Conference*, pages 141–152, Auckland, New Zealand, Jan. 1999.

[11] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[12] J. Xu and W. Croft. Query expansion using local and global document analysis. In H.-P. Frei, D. Harman, P. Schäuble, and R. Wilkinson, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 4–11, Zurich, Switzerland, Aug. 1996.