What's Next?
Index Structures for Efficient Phrase Querying

Hugh E. Williams
hugh@cs.rmit.edu.au

Justin Zobel
jz@cs.rmit.edu.au

Phil Anderson
phil@mds.rmit.edu.au

Dept of Computer Science, RMIT
GPO Box 2476V, Melbourne 3001, Australia

Abstract. Text retrieval systems are used to fetch documents from large text collections, using queries consisting of words and word sequences. A shortcoming of current systems is that word-sequence queries, also known as phrase queries, can be expensive to evaluate, particularly if they include common words. Another limitation is that some forms of querying are not supported; an example is phrase completion, which provides an alternative way of locating information. We propose a new index structure, a nextword index, that addresses both of these problems. We show experimentally that nextword indexes can be used for rapid phrase querying, and show that they allow practical phrase completion.

1 Introduction

Digital text collections are increasingly important as information repositories and as corporate assets. With the current rapid growth in the use of internet searching and text databases, users expect all documents to be stored online and to be readily available in response to simple queries to text database systems. These systems, which are based on well-understood information retrieval principles, are efficient for simple queries—that is, able to find documents quickly—and effective—that is, able to find documents that satisfy users' information needs [Salton, 1989]. However, growth in the volume of stored information adversely affects the performance of retrieval systems. Obviously, larger indexes must be processed to resolve a query. A more insidious factor is that more information (and much more junk) is retrieved in response to each query, so that effectiveness is degraded; users respond by specifying more complex queries, further increasing costs.

These problems can be addressed in two ways. First, new, more efficient query evaluation techniques can be developed. Second, alternative forms of querying

---

Proceedings of the Tenth Australasian Database Conference, Auckland, New Zealand, January 18–21 1999. Copyright Springer-Verlag, Singapore. Permission to copy this work for personal or classroom use is granted without fee provided that: copies are not made or distributed for profit or personal advantage; and this copyright notice, the title of the publication, and its date appear. Any other use or copying of this document requires specific prior permission from Springer-Verlag.

can be used to identify relevant documents; for example, in practice users do not rely solely on traditional queries to retrieve documents, but instead combine querying with browsing and with techniques such as assisted query refinement [Agosti and Smeaton, 1996].

We propose a new structure, a *nextword* index, that provides a fast alternative approach for querying using phrases and for *phrase browsing*. Compared to a standard inverted index, our nextword index provides more efficient phrase querying. A standard inverted index for a text database contains, for each distinct word in the database, a list of the locations in the database at which the word occurs. In contrast, a nextword index contains, for each distinct word, a list of the words that immediately follow it anywhere in the database, and the locations at which they do so. Using a sequence of words, a nextword index can identify all possible successor words, or can be used to complete successor words that have been partially specified. In conjunction with a standard inverted index, a nextword index provides a rich additional mechanism for accessing documents in large collections.

We have implemented a simple nextword-based retrieval system, Bebop. Using Bebop we show experimentally that typical phrase queries can be resolved in a fraction of the time required with conventional indexes. We also show that nextword indexes can be built with only moderate resources. To our knowledge, comprehensive phrase indexes have not previously been built or tested for large text databases, and similarly phrase browsing and phrase completion have previously only been practical on small data sets.

Naively implemented, a nextword index would be unmanageably large. Using standard index compression techniques our experiments show that the size of a nextword index for a large database is 60% or less of that of the indexed data, and further size reductions are available. Our results show that nextword indexing and phrase browsing are practical on a desktop machine.

2 Text databases and query evaluation

Text database systems are used to store large volumes of text, typically as a series of discrete documents. There are several kinds of queries to such systems [Salton, 1989]. Boolean queries consist of words and Boolean operators; for example, a query such as

(seatbelt OR airbag) AND safety

would match all documents containing "safety" and either "seatbelt" or "airbag". Ranked queries are a natural language expression or a list of words such as

road safety airbag seatbelt car accident

The query is statistically compared to each stored document to identify those that are most similar to the query, using properties such as the relative rareness of each query term and the number of occurrences of each query term in each document. Both forms of query can be extended to, for example, make use of

structure such as document markup [Sacks-Davis et al., 1997]. To allow efficient and effective resolution of queries, every word occurring in every document is indexed [Moffat and Zobel, 1996] [Persin et al., 1996].

The most efficient data structure for evaluating ranked or Boolean queries on large collections is an inverted index [Zobel et al.]. An inverted index consists of a vocabulary, containing each distinct word w that occurs in the database, and a set of inverted lists, one for each w. At a minimum, each list contains the identifier of each document d containing w; to allow effective ranking, the list must contain each frequency of occurrence of w in d; to allow phrase querying, it must also contain the sequence of ordinal positions at which w occurs in d. Thus the inverted list for the term "safety" might be

16, 1: 11; **51**, 3: 11, 81, 90; **52**, 1: 136; ...

representing the fact that it occurs once in document 16, at position 11; three times in document 51, at positions 11, 81, and 90; and so on. Using simple coding techniques this information can be compressed to around 25% of the size of the indexed data (perhaps one-sixth of the size of an uncompressed representation) [Witten et al., 1994].

These coding techniques are based on the principle that, with variable-bit representations for the integers such as Elias or Golomb codes [Elias, 1975] [Golomb, 1966], small numbers can be efficiently represented in just a few bits. Variable-bit codes can be directly applied to counts such as the number of occurrences of a word in a document; for values such as document identifiers, taking the difference between adjacent entries in an inverted list yields small positive integers suitable for variable-bit coding. Use of such compression not only reduces space requirements but also reduces typical query response times, as the cost of decoding is offset by the reduction in disk transfer costs.

Using an inverted file, a query is evaluated by fetching the inverted list of each query term. For Boolean queries, the lists are logically combined to give a set of answer documents; for ranked queries, the lists are used to update values in a structure of accumulators (each corresponding to a document), and the documents with the largest accumulator values are fetched as answers.

Both ranked and Boolean queries can involve phrases, that is, sequences of words that must be contiguous in matching documents. A ranked query involving phrases might be

"road safety" airbag seatbelt "car accident"

which has four query terms, two of which are phrases. Use of phrases in queries can significantly increase effectiveness [Callan et al., 1995] [Xu and Croft, 1996]. To evaluate such a query it is necessary to construct a temporary inverted list for each of the phrases, by fetching the inverted list of each of the individual terms and combining them. If the inverted list for "safety" is as above and the inverted list for "road" is

16, 1: 16; **51**, 2: 12, 30; **68**, 2: 9, 53; ...

then both words occur in document 16, but not as a pair; one or the other but not both occur in documents 52 and 68; but the phrase “road safety” does appear once in document 51, at position 11. The list for “road safety” is thus

51, 1: 11; . . .

Phrase queries can involve more than two words, as for example when searching for documents containing a particular business name.

Ranked or boolean querying is not the only way of searching a text database. An alternative is to use browsing, typically via manually-added or automatically-generated hypertext links; in practice most users find documents using a mix of querying and browsing. Another searching mechanism is to browse the vocabulary of the database to refine the query. A powerful mechanism of this kind is to view terms, and more generally phrases, in the contexts in which they occur in the database [Dennis et al., 1998] [Nevill-Manning and Witten, 1997] [Nevill-Manning et al., 1998].

Improving the efficiency of phrase queries and phrase browsing is the subject of this paper. In the next section we describe our proposed index structure for supporting this kind of query, then in Section 4 give our experimental results.

3 Nextword indexes

We propose that phrase querying and browsing be supported by a *nextword* index. A nextword index consists of a vocabulary of distinct words and, for each word w, a nextword list and a position list. The nextword list consists of each word s that succeeds w anywhere in the database, interleaved with pointers into the position list. For each pair ws there is a sequence of locations (document identifier and position within document) at which the pair occurs; these sequences are concatenated to give the position list.

The structure of a nextword index is illustrated in Figure 1. In this structure, the vocabulary is held in a structure such as a B-tree. The nextwords are sorted and stored contiguously. This allows the use of front-coding; only the characters of each word that differ from those of the previous word need be stored, giving substantial space savings. For example, the sequence of words

4 acne 7 acolyte 5 acorn 6 acorns 8 acoustic,

in which the numbers indicate the length of each string, can be replaced by

4,0 acne 5,2 olyte 3,3 rn 1,5 s 5,3 ustic,

in which the first number is the length of the stored string and the second is the number of characters shared with the previous string. These two numbers are typically small and in a practical application can be stored in codes of 4 bits each; one code must be reserved as an escape for the rare occurrences of larger values. Note that front-coding (the implementation used for the experiments in

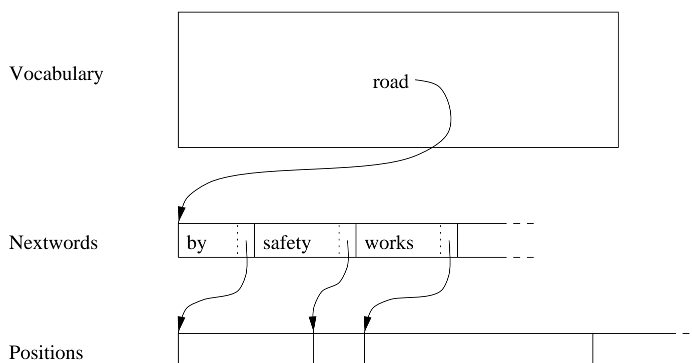

Fig. 1. *Organisation of a nextword index.*

this paper) means that nextword lists must be processed sequentially even if only one nextword is required.

For each word-nextword pair the locations are sorted, then compressed with the techniques used for standard inverted lists. The set of location lists are then concatenated and stored contiguously, so that overall only two disk accesses are required to fetch all the information concerning each index term. As we show below, a nextword index represented in this way is around 60% of the size of the indexed data, an overhead that is significant but not prohibitive.

Contiguous storage of variable-length structures (with, in practical implementations, only the longest lists divided into separate blocks for ease of maintenance) does add somewhat to update costs, but, compared to alternatives such as dividing lists into separately-stored blocks, has the benefit of greatly reducing the cost of query evaluation. This is because disk accesses, particularly to randomly-placed linked structures, are a significant bottleneck in current architectures: in some cases in excess of a million instructions can be executed during the time required to fetch a single block from disk.

Nextword indexes can be used to support querying as follows.

- A phrase query $w_1 w_2$ of two words is evaluated by fetching the nextword list for w_1, decoding to find w_2, then fetching the location list for $w_1 w_2$. (The other positions for w_1 are not required and do not need to be fetched.) Compared to an inverted index—where the complete inverted lists for both w_1 and w_2 must be fetched—the cost saving should be particularly high if the second word is common, that is, has a long inverted list.
  For longer phrases greater cost savings can be achieved, as the locations of the rarest pairs within the phrase can be processed first, typically narrowing down the number of locations to either zero or one after processing only one or two location lists. In the former case processing can stop immediately; in the latter the document can be fetched.
  The only instance in which there is potential for a nextword index to be

slower than an inverted index is when the first word is common and the second is rare, since the nextword list is typically only a little shorter than a conventional inverted list. Even in this case, however, the location list for the pair is by construction shorter than the inverted list of the second word, and if the pair does not occur a disk access is saved.

- Given a word w, the nextword list can be used to identify all following words. This functionality cannot be provided with a conventional inverted index, other than by the costly mechanism of accessing all documents in which the word occurs.

  More generally, a nextword list can be used for phrase browsing. Given a phrase of several words, the nextword index can be used to identify all nextwords for the phrase, as follows. The phrase is processed to find all locations, then the nextword list and position list of the last word in the phrase is filtered to give only the nextwords that follow the phrase. This is more costly than determining nextwords for a single word (in which case the positions do not need to be accessed) but, without a nextword index, this information can only be computed by accessing the documents.
- A nextword index can be used for phrase completion. Suppose that, via the nextword index or an inverted index, it is known that two given words occur in the same document k words apart. The nextword index can be used to fill in the gap.
- A nextword index can also be used for word completion in the context of a phrase. That is, given a phrase such as "road safety l", the nextword index can suggest completions such as "aw" or "egislation".

For example, a user might be interested in finding information about the liabilities of the Central Bank. All of these query terms are common in a hypothetical database of finance articles, so the query "liability Central Bank" might find no relevant documents, while the phrase "Central Bank" has no matches. However, exploration of the word "Central" with a nextword index reveals that "Central Bank" does not exist, but that the phrase "Central Commercial Bank" is common. The user could then give a new ranked query or alternatively directly use the phrase to retrieve documents.

When constructing a nextword index, some decision must be taken about the treatment of punctuation. Some punctuation, such as stops and arguably commas and colons, terminates phrases; other punctuation, such as quotes, can be ignored. When a word occurs at the end of a phrase, it has no nextword. In this case it is helpful to explicitly store a null nextword, to allow accurate processing of phrase completion.

Perhaps the most significant drawback of nextword indexes is that they are monodirectional—they cannot be used to identify preceding words in a phrase. For this task an independent index of prior words is needed. Note that stopping cannot readily be applied to reduce the size of a nextword index, nor indeed is it clear that it is desirable, since phrases often involve common words.

Nextword indexes have several advantages compared to the alternative structures that can be used for the same tasks. One such alternative is PAT trees

[Gonnet and Baeza-Yates, 1991], in which each node is the root of a tree, each pointer is from a predecessor to a successor, and each leaf has a pointer to a list of document occurrences of the sequence of words. A PAT tree of words could be used for phrase browsing, but is impractical for large collections: the whole structure must be kept in memory, and is likely to be considerably larger than the data being indexed.

Another structure that can be used for phrase browsing is the hierarchical non-recursive grammar generated by the Sequitur general-purpose compression algorithm [Nevill-Manning and Witten, 1997] [Nevill-Manning et al., 1998]. In this grammar each rule has a nonterminal on the left and a sequence of symbols on the right, each of which is either a nonterminal or a word. This grammar allows browsing of any sequence of words in the collection and is designed to help the user to grasp the general content of a corpus. This approach does not readily yield the information required for ranking, but note that ranking is not an application for which it was designed. However, the Sequitur approach has several other disadvantages: in particular, to generate the grammar the corpus must be held in memory, and the size of the grammar, which must be held in memory during phrase browsing, is prohibitive.

Perhaps the most obvious alternative structure is an inverted index of word pairs, a structure that, assuming that the vocabulary is lexicographically sorted, is highly similar to a nextword index. The major difference is that the inverted lists of such an index—containing the same information as a location list in a nextword index—would not be stored contiguously, significantly increasing the cost of phrase completion. In other respects there is little to choose between the structures, but this difference is crucial to efficiency.

Note that an index of nextword lists (without positions) together with a separate index of word positions does not provide the same functionality as a nextword index. An index of nextword lists cannot be used for phrase completion or to verify the existence of phrases of more than two words, and only yields a cost saving when querying for phrases that do not exist. The design of nextword indexes means that documents do not have to be accessed until query evaluation is complete; that is, there is no need for costly processes such as false match elimination.

4 Experiments

To test the efficiency of nextword indexes we implemented a nextword indexing system, Bebop. This system is currently a prototype, but in speed of phrase querying is likely to be little different to a production implementation. Each query to Bebop is a phrase of at least one word; output is either the number of matching instances (documents and occurrences within documents); the list of nextwords for the phrase; or the documents with the matching instances.

As test data we used the 508 megabytes of the Wall Street Journal (WSJ) from TREC disks 1 and 2 [Harman, 1995]. (TREC is an ongoing international collaborative experiment in information retrieval sponsored by NIST and ARPA.)

As test queries we generated lists of phrases as follows. From WSJ we randomly extracted 100 two- and five-word phrases (200 phrases in total), giving query sets W2 and W5; in these phrases we imposed the restriction that the first word have at least five letters, to avoid selecting large numbers of phrases beginning with common words, as we judged these to be less interesting queries. We also extracted the commonest phrase for the commonest word (which was "the company"), the 100 rarest two-word phrases beginning with the commonest word, and 100 rare two-word phrases ending with the commonest word, giving query sets Wcc, Wcr, and Wrc. Each of these "W" phrases has, by construction, a least one match in WSJ. We would expect Wcr to show the worst relative performance for Bebop, because for both kinds of index costs are dominated by the need to decode a long list. Wrc should show the best relative performance for Bebop because decoding of a long list is entirely avoided.

We repeated the process of random extraction for the 500 megabytes of Associated Press (AP) data from TREC, giving query sets A2 and A5. For many of the "A" phrases there were no matches in WSJ, allowing early termination when the query evaluation mechanism detected that the phrase did not occur. We also extracted 174 two-word phrases from the TREC queries, giving query set Q2.

To give a baseline for Bebop performance we measured MG [Bell et al., 1995] [Witten et al., 1994] on the same data. The prototype text retrieval engine MG, which is based on compressed inverted files, has recently been extended to evaluate phrase queries. All experiments were run on a Sun SPARC E2000.

We first used MG and Bebop to build indexes of WSJ. (Bebop index construction is a fairly simplistic implementation that could be improved using the pre-planned layout method of MG [Moffat, 1992].) Index construction time for MG was approximately 40 minutes and index size was 153 megabytes, or 30.1% of the indexed data.[1] Index construction time for Bebop was approximately 83 minutes and index size was 57% of the indexed data, consisting of 18% for vocabulary and nextwords and 39% for positions. Only 70 megabytes of memory was used during index construction.

We then used MG and Bebop to evaluate our 775 test queries and timed their performance. Before each run with each query set all system caches were flushed to give a "cold start", ensuring that index information was indeed fetched from disk. Results are shown in Table 1.

Bebop was four to five times faster than MG in finding the matching documents and positions of each query in W2 and A2 and in the Q2 set of two-word phrases from real queries. For each query, Bebop only decodes a single list for the matching phrase, while MG retrieves two lists and subsequently processes both lists. This effect is equally marked in searching with "the company" (Wcc), where MG retrieves and processes two long lists, while Bebop processes the much

[1] Typical index sizes reported elsewhere for a word-position index are 22%–25%. The higher figure here is because in these experiments index terms were neither stemmed nor casefolded, increasing the number of inverted lists and reducing their compressibility.

Table 1. *Average perquery evaluation time for MG and Bebop on WSJ (milliseconds). The second and third columns are time to find the locations of all matching phrases. The fourth column is the time under Bebop to find all nextwords for each phrase. (This functionality is not supported by MG.)*

Query set	MG	Bebop	Nextwords
W2	304	80	2213
W5	383	190	864
Wcc	2093	464	21929
Wcr	121	48	138
Wrc	713	443	9263
A2	254	62	1813
A5	263	80	119
Q2	131	24	164

shorter "the company" list and the list of nextwords for "the"; Bebop is over four times faster than MG for this query.

Bebop is also 1.6 times faster than MG in searching with the Wrc query set. The Wrc query set is ideal for Bebop, since the single list required is that of a rare word, while the two lists required for MG are that of the rare word and the common following word ("the"). However, as the same following word was used in each case MG was able to buffer its inverted list, giving a considerable saving; in retrospect this query set may not have been well chosen. The savings in searching with Wcr are also good—Bebop is 2.5 times faster than MG—which is surprising, as we had constructed Wcr as a worst case for Bebop. As both systems retrieve long common word ("the") lists, Bebop processes many nextwords in locating the rare term, while the second list retrieved by MG is very short.

For longer queries, Bebop is twice as fast as MG to process the W5 queries that are known to occur in the collection, while Bebop is three times faster than MG in processing the A5 queries that sometimes do not occur. To process five-word phrases, both systems retrieve and process several lists. In the case of Bebop, three lists are required to locate a phrase: the nextword lists for the first and fourth words, as well as the shortest of the second and third word lists; these are processed in order from rarest (shortest) to most common (longest). In MG, up to five lists are retrieved, again beginning with shortest list for the rarest term.

For A5 queries, early termination in Bebop is common during processing of the first nextword list for the rarest query term, resulting in a saving in query evaluation time. For MG on the same queries, two lists are required to get early termination. For W5 queries, no early termination is possible and the processing costs of both systems are similar: MG retrieves and processes five lists (where the most common term has a long list of occurrences), while Bebop retrieves

and processes three lists (where the most common term often has a shorter list of occurrences than MG, but requires processing of large numbers of nextwords to locate the list).

Bebop, in contrast to MG and other systems based on inverted files, can be used to find the nextwords occurring after a phrase. For the five-word queries, the costs in finding nextwords are much less than finding nextwords for the two-word queries. For A5 queries, many of the phrases do not occur, allowing early termination, while the number of matching occurrences in W5 is typically small. For Wrc, the cost of finding nextwords is high since all nextwords and related lists for "the" must be processed to resolve the query. Wcr has much lower costs, since the nextword lists and occurrences are much shorter. A2, W2, and Q2 have similar costs in finding nextwords relative to the costs of finding occurrences of the phrase; in some cases, Q2 and A2 phrases do not occur, allowing early termination, while for most of the queries further processing is required to find all possible nextwords. Wcc is the worst-case for Bebop and requires the complete processing of all lists associated with the 19,000 nextwords of "the company", which have over 77,000 separate occurrences.

5 Variations on the theme

Our experiments show that nextword indexing is practical and efficient. However, further efficiency gains are available. For example, the cost of processing each nextword list can be significantly reduced by partitioning the list into groups of words, and then front-coding within groups only, yielding fixed-length chunks [Moffat et al., 1997]. With this strategy the first word of each chunk can be used to determine whether the chunk should be decoded, and binary search can be used to search amongst chunks. Other strategies could be used to further reduce index size. For example, after front-coding the characters in the nextword lists can be compressed with a scheme such as Huffman coding. The nextwords could also be back-coded with respect to the collection vocabulary; for example, if "acolyte", "acorn", and "acoustic" are adjacent, "acorn" can be unambiguously represented by the string "acor", which is sufficient to distinguish it from its neighbours. We estimate that the total size of nextword structures can be reduced to around 13%.

Significant reductions in the size of the position lists can be achieved by observing that, in a practical system, a nextword index will not be used in isolation; there will also be an inverted index for use in conventional querying. The function of the nextword index can in this context be regarded as a mechanism for identifying whether a phrase is present and what the nextwords are. If the inverted index can then be used to efficiently identify the locations of the phrase, it is possible to reduce the size of location information held in the nextword index.

The stored locations can be approximate rather than exact and therefore represented in fewer bits; so long as the information allows correct matching between position lists it does not matter what location values are actually stored. We have developed a model of such a scheme and estimate that the total size of

position lists can be reduced to approximately 34%. The greater compactness of such alternatives may well make them attractive in practice, particular if phrase-browsing is used to construct queries rather than for direct access to documents. We plan to explore these options in further work.

6 Conclusions

Phrases provide a powerful alternative form of accessing text databases, via use of phrases in conventional queries and via new querying modes such as phrase browsing. In this paper we have shown that use of phrases is practical and efficient.

We have proposed a new index structure, a nextword index, for supporting queries on phrases. This structure allows rapid evaluation of phrase queries and provides phrase-browsing functionality that is not available for large databases with any existing structure. Our experiments with Bebop, a prototype retrieval system based on nextwords, show that phrase completion is practical and that phrase queries can be evaluated four times as fast as with an inverted index. The relative improvement in speed should improve with increasing database size. In the initial Bebop implementation index size was approximately 60% of that of the indexed data. Such an index size is not impractical, and, we have argued, can be further reduced through application of additional compression techniques.

We believe that the useability of text databases is greatly enhanced by provision of a rich set of alternative methods of expressing information needs. Practical phrase browsing and phrase completion are important additions to these alternatives.

Acknowledgements

We are grateful to Neil Sharman and William Webber for their assistance with the experiments. This work was supported by the Australian Research Council.

References

[Agosti and Smeaton, 1996] Agosti, Maristella and Smeaton, Alan, editors (1996). *Information Retrieval and Hypertext.* Kluwer Academic Publishers, Dordrecht, Netherlands.

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

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

[Dennis et al., 1998] Dennis, S., McArthur, R., and Bruza, P. (1998). Searching the world wide web made easy? the cognitive load imposed by query refinement mechanisms. In Kay, J. and Milosavljevic, M., editors, *Proc. Australian Document Computing Conference*, Sydney, Australia. University of Sydney. To appear.

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

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

[Gonnet and Baeza-Yates, 1991] Gonnet, G. and Baeza-Yates, R. (1991). *Handbook of data structures and algorithms*. Addison-Wesley, Reading, Massachusetts, second edition.

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

[Moffat, 1992] Moffat, A. (1992). Economical inversion of large text files. *Computing Systems*, 5(2):125–139.

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

[Moffat et al., 1997] Moffat, A., Zobel, J., and Sharman, N. (1997). Text compression for dynamic document databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):302–313.

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

[Nevill-Manning et al., 1998] Nevill-Manning, C.G., Witten, I.H., and Paynter, G.W. (1998). Browsing in digital libraries: a phrase-based approach. In Allen, R.B. and Rasmussen, E., editors, *Proc. ACM Digital Libraries*, pages 230–236, Philadephia, Pennsylvania.

[Persin et al., 1996] Persin, M., Zobel, J., and Sacks-Davis, R. (1996). Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764.

[Sacks-Davis et al., 1997] Sacks-Davis, R., Dao, T., Thom, J.A., and Zobel, J. (1997). Indexing documents for queries on structure, content, and attributes. In Yoshikawa, M. and Uemura, S., editors, *Proc. Int. Symp. on Digital Media Information Base*, pages 236–245, Nara, Japan.

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

[Witten et al., 1994] Witten, I.H., Moffat, A., and Bell, T.C. (1994). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York.

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

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