Web presentation of very large text and speech corpora

Zhibiao Wu,Mark Liberman
Linguistic Data Consortium
University of Pennsylvania
3615 Market Street, Suite 200
Philadelphia PA 19104
wzb@ldc.upenn.edu

Abstract

This paper describes new algorithms used in the LDC Online implementation. LDC Online provides regular expression text pattern retrieval and speech segment retrieval for very large text and speech corpora. All text is tagged with a part of speech tagger. The search pattern as well as the retrieval result can be any one of five different types. For speech corpora, it provides speaker dynamic categorization and speech segment retrieval with regular expression search on transcripts. In contrast with today's conventional Web search engines which use an inverted index for the full-text data and only provide word, or word phrase search, we use a two level indexing scheme to handle these five different types so that the regular expression search algorithm can be very fast. We will introduce an algorithm used in speaker categorization for speech segment grouping. With a special treatment of the speaker categorization and transcription file indexing, the algorithm achieves a performance in the order of $O(N)$ rather than $O(N^2)$.

Introduction

With rapidly developing Web technology, presenting very large corpora on the Web is not only possible but also necessary. At LDC, there are hundreds of gigabytes of data available, and the amount is roughly doubling every year. Very few research sites have adequate available disk space to store this data with necessary indices for searching. In addition, convenient and rapid searching requires a significant investment in program development and index creation, and so few sites have adequate indexing, search and retrieval facilities for all the LDC data, or even for very large pieces of it. Therefore, we have implemented the LDC Online service (http://www.ldc.upenn.edu) which attempt to provide rapid, flexible and convenient search and retrieval of all LDC data for linguistic research and development.

This paper describes new algorithms used in the LDC Online implementation. LDC Online provides regular expression text pattern retrieval and speech segment retrieval for very large text and speech corpora. All text is tagged with a part of speech tagger. The search pattern as well as the retrieval result can be any one of five different types. For speech corpora, it provides speaker dynamic categorization and speech segment retrieval with regular expression search on transcripts. In contrast with today's conventional Web search engines which use an inverted index for the full-text data and only provide word, or word phrase search, we use a two level indexing scheme to handle these five different types so that the regular expression search algorithm can be very fast. We will introduce an algorithm used in speaker categorization for speech segment grouping. With a special treatment of the speaker categorization and transcription file indexing, the algorithm achieves a performance in the order of $O(N)$ rather than $O(N^2)$.

When a text database is large, but centralized, special indexing mechanisms can be employed to speed up the search. For relatively static documents, a popular indexing mechanism is the use of inverted files (Harman *et al.* 1992),(MacFarlane, Robertson, & McCann 1995),(Moffat & Zobel 1996). Depending on the interest in the words with reference to the positions, the line numbers or the document numbers with which the words appear, a text file can be considered as an array of words, lines or documents of text. A text file is thus a kind of index, giving for each position, line number, or document number the words that appear. In order to find occurrences of words in a file, a different index is built, which is the inverse of the file itself: for each word, a list of word positions, line numbers or document numbers is stored. The inverted file scheme turn out to be the fastest way to index words, since its cost is only $O(1)$ time. However, it normally uses a lot of disk space and memory.

On the other hand, various algorithms have been presented for sequentially searching strings or multi-string patterns without inverted indexing. Aho and Corasick (Aho & Corasick 1975) presented a linear-time algorithm based on an automata approach. This algorithm serves as the basis for the UNIX tool *fgrep*. Boyer and Moore (Boyer & Moore 1977)

presented a regular string-searching algorithm which demonstrated that it is possible to actually skip a large portion of the text while searching, leading to faster than linear algorithms in the average case. Commentz-Walter (Commentz-Walter 1979) presented an algorithm which combines Boyer-Moore and Aho-Corasick's algorithms and is used by GNU fgrep version 2.0. Although these algorithms are designed for fast search, due to its sequential nature, the performance is linear on the file size.

Recently a new indexing mechanism, called Glimpse (Manber & Wu 1994) is becoming popular, because it requires much less space than inverted files and other indexing techniques. The idea is to first compress the text by substituting each of the common pairs of text with a special byte allocated for it and then do the search directly on the compressed text rather than on the text itself, therefore, not only does the system take less space but also the search speed is faster than search on the full text. This is an efficient scheme for sequential search, and serves as a compromise between inverted indexing and sequential search in terms of space and speed.

LDC Online requires search on very large text with very fast response to the Web user's query. Speed is the most important criteria which indicates an inverted indexing scheme. However, since the text is part of speech tagged, there are five different forms for a text corpus. For a speech corpus, beside the transcripts of the speech, the starting time and the ending time as well as the channel are also important properties of an utterance. They have to be retrieved so that a speech segment can be transmitted to the Web user. Applying inverted indexing for each of the different forms turns out to be not only a waste of memory but also inefficient. We use a two level indexing scheme to reduce the memory requirement and increase the retrieval speed. For speech retrieval, a new algorithm is introduced which provides a linear grouping on speech segments based on speaker information. The core data structure used in the two level indexing scheme has its roots in Ken Church's unpublished work (Church 1991) at AT&T Bell laboratories and was used by Dave Yarowsky in his concordance program "conc" (private communication). Our work differs from theirs is that we use a different scheme for regular expression search in addition to new algorithms for speech indexing. We present this approach as a potential standard for internet presentation of very large multi-media data. In the following sections, we start by describing in more detail about the problem and the two level indexing. After discussing the regular expression search algorithm, we will focus on speech indexing and discuss the algorithm for speaker categorization and speech segment grouping. We conclude with a discussion of possible applications and the future of LDC Online.

Making the text database indexing

For purposes of LDC-Online searching, texts are divided into lexical tokens or words. Part-of-speech tags are assigned to these words automatically by Eric Brill's tagger (Brill 1992). In searching, words may be referenced or displayed in five different ways. Search patterns are made up of words in these five forms, and the result of the search can be any of the following forms.

- The 'raw' form (e.g. flies) is a word simply as a sequence of letters.
- The 'parts' form (e.g. flies/NNS) is the word with a part-of-speech tag. This will match only if the part-of-speech tagger assigns the specified tag.
- The 'pos-only' form is a part-of-speech tag without any specified word (e.g. NNS), or a part-of-speech macro (e.g. N). This will match any word assigned the specified part(s) of speech.
- The 'lemma' form (e.g. fly/N) is the word's lemma and a corresponding part-of-speech macro.
- The 'casefree' form (e.g. @fly) is the word with all characters mapped into lower case. The '@' sign is used to denote that the subsequent word is a casefree form.

Of these five forms, the 'parts' form can be viewed as the basic form since all the other forms can be uniquely derived from the 'parts' form in one way or the other, but not vice versa. Therefore, we chose the 'parts' form to represent the corpus and all other forms are indexed based on it. They are called secondary forms.

It is wise to compress the corpus to save storage. For the BROWN corpus, the average length of the part of speech tagged words is 7.14. That means on average each part of speech tagged word takes 8 bytes. If we use an integer to represent it, it will only use 4 bytes, half the space.

In the following, we will first introduce the compression method, then we will explain the basic indexing scheme and finally discuss the secondary word form index.

Compressing the corpus

A corpus of part of speech tagged text can be viewed as an array of string tokens. Let

$$C = w_0, w_1, ..., w_{N-1} \quad (1)$$

be the corpus, here N is the corpus length, w_i is any part of speech tagged word at position i. Let

$$T = W_0, W_1, ..., W_{M-1} \quad (2)$$

be the unique words (types) sorted in the part of speech tagged text corpus, M is the number of types. Since all the words in T are unique, we can use their subscripts to represent them, therefore, we have numbers from 0 to $M-1$ to represent all the types in the corpus. These

numbers are called type numbers. The encoded corpus $EN(C)$ then is

$$EN(C) = n_0, n_1, ..., n_{N-1}, \quad (3)$$

$$where\ n_i \in \{0, 1, ..., M-1\} \quad (4)$$

$$w_i = W_{n_i} for\ i = 1, 2, ..., N-1 \quad (5)$$

The decoding of a type number is quite easy. When the words in T are sorted and stored in a character array separated by NULL characters, they can be indexed by:

$$I = s_0, s_1, ..., s_{M-1} \quad (6)$$

in which s_i is the offset of word W_i in the character array. Therefore, once we have a type number, we can easily get the word from the array at position s_i. The cost of this indexing is $O(1)$.

The encoding can be done in a constant time too. If you search through the sorted array S from the beginning to the end to find out the type number for a particular word, the performance is $O(n)$. However, we can make use of the binary search scheme. For a range $[p, q]$ in (0, M-1), we first compare the word with $W_{(p+q)/2}$, if the word is greater than $W_{(p+q)/2}$, we then search for the word in the range of $[p + q/2, q]$, otherwise we search the word in $[p, p + q/2]$. then by divide and conquer, until a W_j word is found, then j is the type number. In this way, the cost of the encoding is $O(\ln M)$.

Basic indexing

The inverted index is pre-processed in the following way: For each W_i in T, we can find all positions where the word W_i occurs in the corpus. The list is defined below:

$$P_i = p_{i,0}, ..., p_{i,f_i}, when\ w_{p_{i,j}} = W_i, j \in \{0, 1, ..., N-1\} \quad (7)$$

Here, f_i is the frequency of the word W_i. By composing all the lists into one integer array, we have

$$P = p_{0,0}, ..., p_{0,f_0}, ..., p_{M-1,0}, ..., p_{M-1,f_{M-1}} \quad (8)$$

$$= p_0, ..., p_{f_0}, p_{f_0+1}, ..., p_{N-1} \quad (9)$$

The length of I is N which is exactly the same as corpus length N. In order to get the index for a type number n_i, we need another index which contains the offset of the first indexing of word n_i in P. That is:

$$PI = c_0, c_1, ..., C_{M-1}, N \quad (10)$$

Here, c_i points to the location of $p_{i,0}$ in the array P. The last element in the PI array is N. It is used as the boundary of the ending position of the last type.

With this data structure, finding out all the positions where a word w_i appears in the corpus involves the following steps:

1. Find out its type number x for w_i.
2. The value of c_x is the starting position in array P. $c_{x+1} - 1$ is the ending position of the index. It is easy to see that $c_{x+1} - c_x$ is the frequency of word w_i.
3. Those positions in the corpus are $p_{c_x}, p_{c_x+1}, ..., p_{c_{x+1}-1}$.

That is the key point of the inverted index which achieve a O(1) time rather than O(n) time in the search for a word in the corpus.

Secondary indexing

Since the secondary type can be uniquely derived from the basic type, one way to index the secondary type is to keep a mapping between the basic form and the secondary form rather than doing a full inverted indexing on the corpus in its secondary form.

Let

$$V = B_0, B_1, ..., B_{L-1} \quad (11)$$

be the secondary types in a frequency decreasing order. L is the number of the secondary words. Here $L < M$. We can use numbers from 0 to $L - 1$ to represent all the secondary words. The encoding and decoding of the secondary types are the same as the encoding and decoding of the basic type. First, we sort the secondary types as

$$SS = S_0, S_1, ..., S_{L-1} \quad (12)$$

We then put the types into an character array and separate them with a NULL character according to the sorting order. We call this array the 'SSORT' array. then we can make the following indices:

- Decoding index: Suppose B_i occurs at the position p_i in SSORT array, the decoding index is:

  $$DE(V) = p_0, p_1, ..., p_{L-1} \quad (13)$$

  With this index, given any internal number i, the secondary type can be easily gotten by looking at the position p_i at SSORT array.

- Encoding index: We make an index from the SS list to the V list.

  $$EN(SS) = e_0, e_1, ..., e_{L-1} \quad (14)$$

  $$where\ B_{e_i} = S_i \quad (15)$$

  Therefore, given any secondary type S, with binary search on SS, the subscript of S can be quickly found within $O(\ln L)$ time. The internal encoding is e_i.

- Mapping from the basic type to the secondary type: Suppose a basic type W_i is mapped to a secondary type B_j, and q_i is the position where B_j occurs in the SSORT array, we have an index from the basic type to the SSORT array:

  $$BM = q_0, q_1, ..., q_{M-1} \quad (16)$$

  Given any basic type number i, its secondary type form can be easily gotten at the position q_i in SSORT array.

- Mapping from secondary type to basic type: Since this mapping is a one to many mapping, two indexing lists are involved. For any B_i, it might be mapped to $W_{r_{i,0}}, W_{r_{i,1}}, ..., W_{r_{i,f_i-1}}$, here f_i is the

number of the different basic types the B_i mapped to. We then list all its subscripts here, we have:

$$R = r_{0,0}, ..., r_{0,f_0-1}, ..., r_{L-1,0}, ..., r_{L-1,f_{L-1}-1} \quad (17)$$

Another index is built upon R based on the position where $r_{i,0}$ occurs in list R. We have:

$$SM = h_0, h_1, ..., h_{L-1}, L \quad (18)$$

$$hereR[h_i] = r_{i,0} \quad (19)$$

With these indices, when given a secondary type number i, elements in the list R from $h_i$to $h_{i+1} - 1$ are the basic types which this i is mapped from.

Regular expression match

LDC Online offers a special regular expression search on the text corpus. Some restrictions are applied to the regular expression so that the search won't take too much time. In this section, we will first explain the format of the regular expression, then discuss our special pattern matching algorithm.

Regular expression format

A search pattern is made up a sequence of search pattern elements. There is one important overall restriction on search patterns: the pattern as a whole must reference at least one specific word.

There are three types of search pattern elements, listed below:

1. Single word elements: A search pattern element can be a word in one of its five forms, or the period '.' that matches any word. The following are examples of such single-word elements.

   ```
   bank, Bank, BaNK, bank/NNP, bank/N,
   @bank, NNS, N, .
   ```

   However, the last three examples NNS, N, . are not by themselves legitimate search patterns, because they do not contain any specific word reference.

2. Word alternative elements: A search pattern element can also be several words separated with the '|' vertical bar (with no spaces around the |). In this case, POS, or POS macros are not allowed to mix with other word forms. The search result is the union of the individual words. For example, the following are legal elements of this type:

   ```
   bank|school, bank/NNP|school/N,
   bank|@school, N|V
   ```

3. Repetition elements POS and POS macros can be modified to indicate various types of repetition. Currently, the system supports the following modifications:

   ```
   ?  = 0 to 1 occurrences.
   +  = 1 to 6 occurrences.
   *  = 0 to 6 occurrences.
   *m,n  = m to n occurrences, m,n <= 6
   ```

   Thus the following are legal repetition patterns:

   ```
   V+  = 1 to 6 verbs
   N*2,3  = 2 to 3 nouns
   J?  = an optional adjective
   ```

   These modifiers can also be used alone, in which case they match the appropriate number of occurrences of any lexical token. You can think of these as being equivalent to a period '.' followed by the repeating modifier. However, the forms such as '.*' do not currently work.

4. Word sequences: A search pattern can be a sequence of the three types of search pattern elements defined above. The search result will be all the occurrences of sequences of words that match the specified sequence of pattern elements. For example, the following are legal patterns:

   ```
   "the @bank", "senior high school"
   "bank V", "the NNP bank/NNP|school",
   "the . bank", "the . . bank",  "based N+ on"
   "school V+", "the *2,3 bank",
   ```

   However, if the overall pattern does not contain any specific words, the program will refuse to search for it. Thus the following are illegal patterns:

   ```
   "bank|NNS", "V NNP", "VB NN", "N+ V V"
   ```

The regular expression match algorithm

As we have defined above, the following restrictions make our search simple and fast.

- With a limited range of repetition (in our case, the range is 6), the search is restricted locally. Patterns such as 'take into + account' will only match the '+' for 1 to 6 occurrences rather than the whole corpus.
- With the limitation of at least one word that has to be referenced in the pattern, the search can find at least a set of positions to start the match. It avoids the search throughout the corpus for patterns like 'V', 'N V' etc.
- With the restriction that POS, or POS macros cannot appear in the alternative elements, such that 'BANK|NNS' is illegal, we avoid doing a union operation on the list of 'BANK' and the list of 'NNS' where the list of 'NNS' is very large.

Based on these restrictions, we can separate the elements in a pattern into two categories. One is called *anchor element* which match at least one word. For example, 'bank', 'bank/NNS', '@take', etc. are all anchor elements. The other is called *conditional element* which belongs to POS, or POS macros with repetition modifiers. Generally, conditional elements match many more words than anchor elements. The basic idea of our algorithm is to first find out the unique set of the anchors, and then check to see if the conditional element is matched or not. The steps are shown below:

1. Parse the pattern, only accept legal patterns.
2. Identify the anchor position and the conditional position.
3. Find out the frequency of each anchor position. For alternative element, the frequencies is the sum of the individual word frequencies separated by |. For example: a pattern 'take N into' may have the following frequencies:

```
pattern    take  A  N*  into
anchor     yes   no no  yes
frequency  548           1682
position  (-2,-8)       0
```

4. Find out the corpus position list for the element which has the lowest frequency.
5. Keep doing the following procedure until there are no anchor elements to the left:
   (a) Find out the anchor element to the left.
   (b) Calculate the nearest and farthest allowed positions according to the range of the conditional elements in the middle. In the above example, 'A' takes one position, N* takes 0 to 6 positions, so the nearest position of 'take|get' is 2 position away from 'into', the farthest position is 8 away from 'into'.
   (c) With the current list, generate a new list which at the positions from nearest to the farthest (here it is from -2 to -8), the left element occurs and record the left positions of the match.
6. Then do a similar operation for right anchor elements. After the operation, we have a list match all for the anchor elements with each matching position recorded.
7. Finally, we check each position in the list to see whether the conditional elements are matched or not.

Speech time indexing and speaker categorization

Besides providing text retrieval for the speech transcripts, speech retrieval has to take speech segments into consideration. When a word or a regular expression is retrieved, users have to be offered the choice of retrieving the speech segment for that word or regular expression. Actually, in speech retrieval, speech segment retrieval is much more important than transcript retrieval.

Indexing scheme for speech time alignment

For research purposes, normally a speech transcript not only has the text, but also has the time alignment (beginning time and ending time in the speech file), and the channel information (Conversations on the telephone have both local channels and remote channels) attached to each word. Let us take a piece of transcription in the SWITCHBOARD corpus as an example:

```
A.1    1.06    1.68    Okay.
A.1    1.76    2.34    So,
A.1    2.36    2.60    uh,
A.1    3.06    3.38    then,
A.1    3.54    3.70    do
A.1    3.72    3.96    you
A.1    4.00    4.58    keep
A.1    4.58    4.80    kids?
```

The first column represents channel A which is the local channel, (B is the remote channel), The number '1' here represents the first turn in the conversation. The second column represents the starting time of the speech for that word in the speech file, the third column is the ending time of that speech segment, the fourth column is the transcribed word. We build the index by first treating the speech transcripts as one large text corpus and building the necessary text index. Then the time alignment index can be built.

1. Time alignment table: We use 1 to represent channel A (local), and 2 to represent channel B (remote), 0 represents both channels, the time alignment information for a word w_i in the transcript corpus can be written as (fc_i, fs_i, fe_i). In a transcript, not all words has the time alignment information. Some transcripts only have time alignment information at the level of conversational turns. Therefore, in order to save space, we can have a list for the time alignment information, another list as an index on it. They are defined below: Suppose in the corpus $C = w_0, w_1, ..., w_{N-1}$, only words $D = w_{i_0}, w_{i_1}, ..., w_{i_{H-1}}$ have time alignment, we can have the table:

$$fc_{i_0}, fs_{i_0}, fe_{i_0}, \quad (20)$$
$$fc_{i_1}, fs_{i_1}, fe_{i_1}, \quad (21)$$
$$..., \quad (22)$$
$$fc_{i_{H-1}}, fs_{i_{H-1}}, fe_{i_{H-1}} \quad (23)$$

2. Time alignment index: We then compose an index for each w_i based on the above list.

$$TI = t_0, t_1, ..., t_{N-1} \quad (24)$$

   Here t_i is -1 if w_i has no time alignment, and t_i is j if w_i has time alignment fc_j, fs_j, fe_j.

Therefore, for any word w_i in the transcripts, we can find its time alignment by looking at the position t_i in the time alignment table.

Speaker information and dynamic categorization

An important property of a speech corpus is the speaker information attached to a transcript file. It is important to know the age, gender, education level, dialect region of the speakers for training speech recognition systems. The following is an example from the SWITCHBOARD corpus.

```
File Channel id   Sex Age Edu region
```

```
2295 A      1000 F   50  1   SOUTH_MIDLAND
2678 B      1034 M   60  3   NORTHERN
2778 A      1110 M   60  3   WESTERN
......
```

The number of choices for gender, age, education level and dialect region are different. Gender has two choices. Ages are divided into five levels (i.e., 20-29, 30-39, 40-49,50-59, and 60-). Education level is (1,2,3,4,5). 10 different dialect regions are listed. It is good to let the user choose any of the properties here and get the speech segment of a particular regular expression, for example, to get the speech segments of the word 'hello' in the SWITCHBOARD corpus whose speakers are female and from dialect region NORTHERN and WESTERN, or to get all the speech segments of the word 'take into' in the corpus whose speakers are male and with education level 3 and up, etc. With all the different choices here, the possible number of different speaker categories is: $500 = 2 \times 5 \times 5 \times 10$. If other properties of the speech file is taken into consideration, such as the microphone quality, the training and testing set etc, the number of categories will be even larger. It is not feasible to generate all the possible regions before the retrieval. Our implementation lists all the possible values for each property and lets the user make the selection so the program can generate the region dynamically at run time.

Since a transcript file naturally defines a region in the speech corpus, an index file is generated to index the beginning position and the ending position of the transcript file in the corpus. Normally, a speech file is recorded in one session with the same speakers and the same recording conditions. The transcript file index also represents the smallest region for a speaker category. The user defined speaker category can be generated by combining the transcript regions which meet the user's property selection. This is actually done with UNIX 'egrep' program. First a file is generated by listing transcript region together with the properties. An example of transcript indexing is shown below:

```
Start End   File Cha id   Sex Age Edu region
1233  2323  2678 B   1034 M   60  3   NORTHERN
4546  5678  2778 A   1110 M   60  3   WESTERN
......
```

Then, based on user's selection, a regular expression is composed and used by 'egrep' to search for the speech file index position. With the above example, suppose user choose NORTHERN or WESTERN, a regular expression 'NORTHERN|WESTERN' is composed, the result of 'egrep' get the region (1233,2328), and (4546,5678) and all other regions which matches the regular expression.

The speech segment grouping algorithm

The final step is to output only those matched regular expressions which fall into the selected regions. A natural way to do it is to get the postion of each matched regular expression in the corpus and check to see if the position falls into the selected regions. In this way, if a number of N regions are selected, and a number of M regular expressions are found, the performance is $O(N \times M)$. Since the output of the matched regular expression is always in a position of increasing order. If the transcript regions are sorted according to start positions, we can reduce the cost from $O(N \times M)$ to a $O(N + M)$. Suppose P_i, i is from 0 to $M - 1$ is the position of i-th regular expression found in the corpus, and S_j, j is from 0 to $N - 1$, is the j-th selected region, the algorithm output all the positions which are in the selected regions.

```
1  i <- 0, j <- 0;
2  while i != N and j != M
3    if P[i] < S[j] then
4      i <- i + 1;
5    else if P[i] <= E[j] then
6      output P[i]
7      i <- i + 1;
8    else j <- j + 1;
```

Discussion and future Work

We have presented data structures and algorithms implemented in our Online service. These algorithms have made the retrieval quick enough to meet online user's needs. In the future, we plan to include video documents online for retrieval. The main goal of LDC Online is to build a centralized linguistic data and computing service which will be used for research and education purposes. We have built an experimental linguistic data computing server which will generate and merge histograms for different corpora as well as the concordance output. N-gram data used in speech language modeling will also be provided in the near term. in addition, we will explore the possibility of making use of our Online material in the area of second language teaching.

Acknowledgments

This work is partially supported by National Science Foundation under grant contract No. IRI-9528587. The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

References

Aho, A. V., and Corasick, M. J. 1975. Efficient string matching: An aid to bibliographics search. *Communications of the ACM* 18:333–340.

Boyer, R. S., and Moore, J. S. 1977. A fast string searching algorithm. *Communications of the ACM* 20:762–772.

Brill, E. 1992. A simple rule-based part of speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing.*

Church, K. 1991. A set of unix tools for processing large text corpus. In *Unpublished manuscript.*

Commentz-Walter, B. 1979. A string matching algorithm fast on the average. In *Proceedings of 6th International Colloquium on Automata, Languages, and Programming*, 118–132.

Graham, I. S. 1995. *HTML Sourcebook: A Complete Guide to HTML.* New York: John Wiley & Sons, Inc.

Harman, D.; Fox, E.; Eaeza-Yates, R.; and Lee, W. 1992. Inverted files. In Frakes, W. B., and Baeza-Yates, R., eds., *Information Retireval Data Structure & Algorithms.* Prentice Hall.

MacFarlane, A.; Robertson, S. E.; and McCann, J. A. 1995. On concurrency control for inverted files. In *Proceedings of 18th Annual BCS Colloquium on Information Retrieval.*

Manber, U., and Wu, S. 1994. Glimpse: A tool to search through entrire file systems. In *Proceedings of Usenix Winter 1994 Technical Conference*, 23–32.

Moffat, A., and Zobel, J. 1996. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems* 14.

Naughton, P. 1996. *The Java Handbook.* New York: Osborne McGraw-Hill.