SCANNING COMPRESSED FULL TEXT FILES

Carolyn R. Watters  
Jodrey School of Computer Science  
Acadia University  
Wolfville, Nova Scotia,  
Canada. B0P 1X0  
e-mail: cwatters@dragon.acadiau.ca

and

Matthew Young-Lai  
Department of Mathematics, Statistics, and Computing Science  
Dalhousie University  
Halifax, Nova Scotia,  
Canada. B3H 1J5

ABSTRACT

In this paper we discuss an application of compression, not with the overall goal of reducing disk space, but with the goal of extending the applicability of full text scan procedures to larger text files for use in on-line search environments.

This paper presents an alternative to inverted file generation for access to full text data files of medium size, 50-200Mbytes, for which the cost of generating a full inverted file is not warranted. Full scan techniques, which are often useful in interactive situations for small files, become unacceptably slow for interactive sessions with files above 10 Mbytes and so the use of compression to reduce the quantity of data scanned is an attractive alternative. Furthermore, an index that can be used to reduce search time further to very acceptable 1-2 second times can be generated as a byproduct of the compression.

1. This research has been supported, in part, by NSERC, the National Science and Engineering Research Council of Canada.
SCANNING COMPRESSED FULL TEXT FILES

C.R. Watters and M. Young-Lai

1. Introduction

In this paper we discuss an application of compression, not with the overall goal of reducing disk space, but with the goal of extending the applicability of full text scan procedures to larger text files for use in on-line search environments.

This paper presents an alternative to inverted file generation for access to full text data files of medium size, 50-200 Mbytes, for which the cost of generating a full inverted file is not warranted. Full scan techniques, which are often useful in interactive situations for small files, become unacceptably slow for interactive sessions with files above 10 Mbytes and so the use of compression to reduce the quantity of data scanned is an attractive alternative. Furthermore, an index that can be used to reduce search time further to very acceptable 1-2 second times can be generated as a byproduct of the compression.

Full text scan as a search technique is useful in those situations where text is available from a variety of sources that the user may want to use immediately without further processing, such as data from newsgroups, large downloaded texts, newspapers, computer programs, or disk dumps. For immediate or very occasional access the user may not want to take the time or resources to manipulate the data files into a particular format or to generate full indices for fast searching. A full scan of the text operates independently of the format the file arrives in and can be executed as soon as the file or update arrives. To get a feeling for the size of common texts, the textual component of a single issue of a newspaper, such as the Wall Street Journal [15], is a little less than one Mbyte of text. Full text scanning of text of this size does not present any difficulties for providing on-line interactive searches within a few seconds but a month's worth or a corpus of legal cases or documentation for a very large computer system can not be searched interactively in the same time frame. Nonetheless, full text scanning has several advantages that just don't go away: no additional overhead is required either for processing or space, updates can be incorporated effortlessly, and searches can be made immediately.

Any compression technique used for scanning in this situation must be a lossless operation so that the text resulting from decompression is exactly the text that was compressed. In addition, the time required for the compression must be short enough to maintain the immediacy of the full text scanning environment and certainly must be considerably less than the time required to generate an inverted file.

225
2. Background

Searches in text are typically supported by an inverted file linking each term in the file to those items (documents or records) in which that term occurs. The process of searching for a term then becomes a simple matter of looking it up in the inverted file. While this allows for very fast query processing, it requires a considerable amount of sorting and processing plus additional space to store the inverted file, typically in the order of 1 to 3 times the size of the original text [11].

Alternatives to inverted files, such as bit maps or document signatures [10][14][4] have been used to reduce the overhead in additional storage required to support text searching but at a cost of additional front-end processing of the data. This initial high cost in processing may not be warranted for infrequent searches. Improvements to hardware [6] and software scanning of text have also been explored and interested readers will find a useful overview of these algorithms elsewhere [11][7].

A considerable amount of effort has been spent examining compression techniques for text as the rate at which text is produced continues to outstrip the rate at which increases in the capacity to store or process it are achieved[1]. The properties of full text that make it an entrenched mode of communication also make it a good candidate for compression. Full text encodes the natural redundancies present in spoken language and a large portion of any text is made up of relatively few, mostly meaningless syntactical words such as 'the' or 'and' that contain little by the way of information and can be compressed greatly.

Text compression techniques and models fall largely into two classes: probabilistic models and dictionary models. Hybrid techniques are often used in practice to provide the best results.

Probabilistic models include Huffman and arithmetic encoding [1][2]. The basic concept of the probabilistic models is to replace those characters or characters strings most likely to occur (i.e., occur most frequently) within a text with the shortest unique code available. Space is saved by the difference in the space needed for the original text and for the replacement code. The compression performance of these methods depends on how successfully they are able to model the individual probabilities of occurrence of the characters and/or strings in the text. Many different approaches to probabilistic modelling exist ranging greatly in terms of effectiveness and complexity [12][3] but some form of probabilistic modeling is evident in almost all compression techniques. Probabilistic methods come closest to providing optimal coding schemes at the expense of effort to produce.
The second basic class of compression models for text include the dictionary-based models. Dictionary-based methods work by constructing a dictionary of substrings contained in the text, and substituting a number representing the dictionary entry whenever one of these substrings is encountered in the text. Again compression results as long as the dictionary entry number, or pointer, is smaller than the original substring and the amount of compression depends on how much repetition occurs in the text. The dictionary may be explicitly stored or the algorithm may use simple heuristics which allow the decompressor to reconstruct the dictionary from the compressed stream in the same way that the compressor does, from the uncompressed stream, thus circumventing the requirement that the dictionary be explicitly stored (e.g. LZ77). A variation is to let the "dictionary" float through the text and be relevant to a current "window" of text (e.g. LZ78). Dictionary-based compression schemes are typically much faster than probabilistic compression methods for both compression and decompression and some even produce comparable compression results. Dictionary-based schemes may be implemented in one pass, in which the dictionary is built at the same time as the data is compressed.

The goals of most efforts at text compression have been to reduce storage space, better utilize CD-Rom space [3], reduce disk accesses [8], and/or to reduce network or telecommunications charges. Typically actual searches are done on the decompressed text or using non-compressed inverted files to access the compressed text files. Manber[9] reports using compressed files to speed up sequential search. Some compression schemes, such as n-grams [12] and LZ schemes may be effective for compression and decompression but not good candidates for searching in the compressed form [1].

3. Compression Application

DalText [16] is an experimental text system that provides interactive access to textual data in its native format (i.e., without requiring preprocessing of the data) and that provides the user with a variety of models of retrieval, such as browsing, instantiation of tables, and Boolean term querying. One of the primitive operations of this system is string searching by scanning the data stream. Since the system is meant to be highly interactive search times of more than a few seconds are not acceptable nor are they conducive to real-time investigations of the data by the user. For data files in the 1-5 Mbyte size (one day's worth of the Wall Street Journal is about 1 Mbyte) scan searching gives adequate response times. As the data file size increases, however, response time degrades until by 25 Mbytes the user may wait 5 minutes for a lengthy or complex query to be processed. Within the concept of a real time data interaction environment, we felt that files of at least 50 Mbytes (two months of a paper, good size ftp source file, set of full text journal articles, patent
cases, or medical histories) should be able to be handled with ease without incurring the cost of generating and storing an inverted file. Access rates to data of this nature may be infrequent making it hard to justify the time and space requirements for more sophisticated inverted file based processing. In addition, updates to the text files should be handled continuously so that the user has access to an evolving text base.

3.1 Compression Method

Compression can be used to increase access speed for the full text scan operations by reducing the amount of text actually scanned. In addition the dictionary of terms generated during the compression can be used as an index of terms in the text. Speed-up of the scan is then achieved in two ways: fewer disk accesses and fewer bytes scanned.

The model of compression used is a variation of the word-based dictionary approach. Substrings or "words" are chosen as dictionary entries as the input text is parsed according to a few simple rules, described below. The first time a word is encountered it is entered into the dictionary. Each occurrence of that word in the text is then replaced by the entry number (i.e., pointer) in the dictionary.

Using the basic method, a search can be done on the compressed text by first looking up the search word in the dictionary (either with a simple scan or by hashing), and then reading in the compressed text file (i.e., now a file of pointers) and scanning for matches. This will result in searches somewhere between 2 and 4 times as fast as those on the original file (i.e. exactly the ratio of the uncompressed size to the compressed size). As a side effect, this also allows for very quick elimination of queries for words that do not match any words in the dictionary, i.e., do not occur in the text file.

In tests using this method on text files from the Wall Street Journal, we found that this method improved the access times for data files up to 10 Mbytes at the same rate as the compression ratio, about 3:1.

We then looked for variations on this method that would provide acceptable performance for larger data files, in the 50 Mbyte range. One obvious choice would be to aim for better compression, possibly using a more sophisticated algorithm. This approach alone, however, is clearly an exercise in diminishing returns as search time would remain directly proportional to compression ratio and any number of enhancements on the basic method would never result in compression ratios that were orders of magnitude better.

An alternative, that we chose to pursue, was to split the dictionary into subdictionaries each associated with compressed data and to build a small merge index along with the dictionaries to use for reconstructing the original text from the data in the compressed data files. This would mean that we would trade off a small increase in
storage for a considerable reduction in the amount of compressed data actually scanned. Such an index would be produced during the compression, not contribute to processing time, and not significantly affect the compression ratio. The original text is not just compressed by the replacement of terms by smaller codes but the terms from the text are put into a set of compressed files associated with the subdictionaries. It is these compressed files that are actually scanned.

We chose to split up the single master dictionary into multiple sub-dictionaries, called split dictionaries or splits, using some property of the word in the text to determine which split dictionary it should be placed in (similar to hashing). Each split dictionary has its own, separate compressed file associated with it. This compressed file contains the code of each word in that dictionary as it is found in the text. Each word in the text is encoded by placing a pointer (i.e., code into the appropriate dictionary) in the compressed file associated with that dictionary and a pointer to the split dictionary is placed in the merge file. This is conceptually the same as taking a full pointer into the original single dictionary, removing several bits to be placed in a merge file, so that uniqueness is maintained by the combination of merge file bits and sub-dictionary bits.

To illustrate the compression algorithm, consider the text "Tomorrow and tomorrow and tomorrow creeps in this petty pace" [13] and a dictionary subdivided by first letter. Initially the split dictionaries are empty. The first word encountered is "Tomorrow" which becomes the first entry in the first split. A pointer to that split is added to the merge file and a pointer or entry number replaces the term "Tomorrow" in the compressed file associated with that split. The next word, "and" becomes the first entry in the second split and is replaced with a pointer into that split in the corresponding compressed file, and so on. The result would be

<table>
<thead>
<tr>
<th>split 1:</th>
<th>split 2:</th>
<th>split3:</th>
<th>split4:</th>
<th>split 5:</th>
<th>split 6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomorrow</td>
<td>and</td>
<td>tomorrow</td>
<td>creeps</td>
<td>in</td>
<td>petty</td>
</tr>
<tr>
<td>this</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pace</td>
</tr>
</tbody>
</table>

with associated split compressed files (CF) containing:

<table>
<thead>
<tr>
<th>CF1:</th>
<th>CF2:</th>
<th>CF3:</th>
<th>CF4:</th>
<th>CF5:</th>
<th>CF6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

and the merge file would be 123234566
where each term is replace by two bytes, one to the split stored in the merge file and one to the position within dictionary for that split.

Reconstruction of a document or section of text requires both the merge information and the split information. The merge file determines which compressed file the next term comes from and the split compressed file determines what that term is. Searching for a term requires only the information in a split. The word is looked up in the appropriate split dictionary and the corresponding compressed file is scanned, a procedure that requires as many times less secondary storage access as there are splits of the dictionary.

As matches are made within the split compressed files the offsets of these matches (within the split compressed file) need to be related to documents (or items) of the data file so that the user can be presented with a list of items or hits containing that term. The most direct way to handle this is to build a document index of offsets with respect to each of the split compressed files during the compression process. Using this document-offset index, a single offset within any split file can be used to obtain a document number. The processing required to build this index is trivial and the extra storage is a few bytes per item in the file.

The nature of natural language allowed us to modify our method slightly with concepts from the probabilistic model to improve the performance of the compression. Recognizing that much of text [11] is made up of repetitons of a few common words and that none of these common words are likely to be used in a query (the results would not be meaningful) meant that these common words could be treated differently than other words found in the text. Firstly, the common words can be targeted for shorter codes which take up less space. Secondly, as they are not useful search terms, the codes for these words need not be kept in the compressed split dictionary files that are actually searched but can be kept in an separate split. This extra split will not generally be searched or used except for decompression. Furthermore, within a given language and type of text (newspaper or scientific writing, for example), the composition of the set of very common words has been shown to vary little from data file to data file [5][1].

A further improvement on search time was made by separating out "non-words" as well. "Non-words" include those strings found in texts that are not likely to be found in a regular dictionary: such as strange punctuation combinations, long strings of repeated white space or separating characters, and possibly numbers. For the most part these strings are not considered to be searchable terms and so were put in their own split dictionary as found. This split dictionary is not scanned during searches. Unfortunately, the growth of the "non-words" does vary widely amongst different types of text files. For example, newspaper reports include a fair amount of numeric data, most of which is nonrepeating.
The "non-word" split dictionary then is the most likely split to overflow during additions to the file for some text types.

The following table presents the basic characteristics of 6 data files used with the compression algorithm.

Parameters for test files

<table>
<thead>
<tr>
<th>file</th>
<th>size(MB)</th>
<th>%c</th>
<th>%n</th>
<th>doc</th>
<th>gr-non</th>
<th>gr-dict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall Street Journal</td>
<td>100</td>
<td>70</td>
<td>2.6</td>
<td>612.4</td>
<td>43 N^5</td>
<td>.07 N^8</td>
</tr>
<tr>
<td>San Jose Mercury News</td>
<td>100</td>
<td>72</td>
<td>2.5</td>
<td>637.8</td>
<td>47 N^5</td>
<td>.05 N^8</td>
</tr>
<tr>
<td>alt.folklore.urban</td>
<td>5</td>
<td>77</td>
<td>3.9</td>
<td>303.9</td>
<td>14 N^5</td>
<td>.76 N^7</td>
</tr>
<tr>
<td>comp.lang.c++</td>
<td>5</td>
<td>78</td>
<td>4.1</td>
<td>386.6</td>
<td>21 N^5</td>
<td>.80 N^7</td>
</tr>
<tr>
<td>comp.os.linux</td>
<td>5</td>
<td>74</td>
<td>6.0</td>
<td>359.4</td>
<td>10 N^6</td>
<td>1.5 N^7</td>
</tr>
<tr>
<td>rec.arts.startrek.current</td>
<td>5</td>
<td>76</td>
<td>3.6</td>
<td>430.6</td>
<td>16 N^5</td>
<td>.38 N^7</td>
</tr>
</tbody>
</table>

where
%c is percentage of words covered by the common dictionary
%n is percentage of words covered by the non-word dictionary
doc is average document length in words
gr-non is the growth rate found for the non-words
gr-dict is the growth rate found for the dictionary words.

3.2 Search Time

The average processing required to search for all occurrences of a word will be largely determined by the size of a single split dictionary and the average size of a single split compressed file. The amount of processing needed to search for occurrences of any term is inversely proportional to the number of splits while the compression ratio is directly related to the number of splits.

Choosing the optimal number of splits to balance search time and compression performance is the most obvious target for inquiry in light of the current results. Ideally, it would be best if searches could be made instant for perceptual purposes without requiring the sacrifice of a large amount of compression performance.

The rate of increase of the number of different words in the dictionary is a factor in determining the effectiveness of this type of dictionary-based compression as larger files are used. If the number of different words increases linearly with the length of the input file, i.e., little repetition is used, then the dictionary will become too big too fast. If the number of new words increases slowly as the file increases in size then search times will grow much more slowly than the file size.

Vocabulary growth has been shown, at least empirically, and in our test files, to be a function of the number of words in the input file [5][11] and is approximated by
\[ n = a N^b \]

where \( N \) = total number of words in input, \( n \) = vocabulary size (number of different words), and typically, \( 10 \leq a < 50 \), with \( 0.5 \leq b \leq 0.6 \). With this expected growth rate, we can see that the number of new words encountered as the text size increases drops off rapidly.

Once the parameters of the equation have been determined for a given type of data, it is possible to make a reasonable prediction of dictionary sizes and compression performance.

3.3 Implementation Considerations

The criterion for deciding which split a given word will be located in is an aspect of the method that is open to variation. As we decided to allow partial prefix word matches, such as \texttt{COMP*}, in addition to exact matches in the queries, words with similar beginnings should be located in the same split dictionary. This was accomplished, albeit roughly, during compression by using a preset table to determine which split to place a word in based on the first characters. If the exact prefix frequency for a given file were available during compression then this would also allow for the division of the dictionary into equal sized splits. The number of characters used as prefix characters depends on the number of splits desired, e.g. using only the first character of the word would provide roughly 64 splits (for a character set of upper and lower characters plus special characters).

Notice that balanced dictionary splits will not generally result in balanced splits of compressed text as the proportion of different words in the dictionary starting with a given prefix does not reflect the proportion of actual words in the text that start with that prefix. Simple empirical testing done with the sample data files does reveal, however, that the relative proportions are about the same, at least for those texts.

Using a specific prefix table does, however, imply that the data has been processed initially in order to generate this prefix information. Consequently, two passes of the input text are needed: one to build the initial dictionary and one to compress the text after the initial dictionary has been split into its final divisions. An alternative and preferable method requires only one pass over the input text and each new word is assigned to a split when it is first encountered. This allows the dictionary to be built and the text to be compressed simultaneously in one pass of the input text but does not guarantee optimal balance of words within the splits of the dictionary.

The words chosen for the common split must be determined prior to building specific dictionaries for input files. Fortunately, the most common words in natural language tend to be used heavily enough over a range of data file types that they can be
determined beforehand and used over and over. In addition, the use of the common split with respect to given files or groups of files can be monitored during the compression phase to determine how good the fit is and hence decide when modification might be beneficial.

Using a two-pass algorithm on a 40MHz Sparc, a 52 Mbyte file of Wall Street Journal data takes 472 (UNIX user) seconds for compression compared to 332 secs for gzip on the same file, and 900 to over 2000 seconds, depending on the sophistication of indexing, to generate an inverted file. The one pass algorithm takes only slightly more than one half the time of the two pass algorithm. As our initial purpose was to test the feasibility of our scheme we did little to optimize the compression time.

4. Updating

One of the advantages to full text scanning is the ease at which updates can be incorporated into the text. Any changes or additions to a text are simply made and subsequent searches access the new version immediately. No auxiliary changes are needed. The compressed search scheme described in this paper also provides for immediate update, both for changes and for additions to the text.

Generally, incremental additions of text to a compressed file require only adding new entries as needed in the dictionary and replacing the new text with compressed codes at the end of the old ones. Complications arise when one or more of the dictionary splits becomes full as adjustments to the already existing dictionaries is not possible and the original criterion for splitting them must be maintained. How much the distribution of words in the new text differs from the original distribution depends on how representative the original text was of the entire text. As a general rule, the larger the size of the original text file, the more likely is the initial split to endure through a number of additions.

Incremental updates were executed against the sample text files starting at about 15Mbytes and ending at roughly 50Mbytes.

Updates to the text file, including changes and deletions, may result in entries in the dictionary that are not in fact currently used in the text. Changes and deletions require only decompression and recompression of the affected text. Subsequent searches may find "matches" in the dictionaries but scans of the compressed text will correct this oversight during actual searches.

5. Searching

We considered the basic search operation to return a list of documents in the text in which a given word occurs. This is accomplished, in the general case, by searching a split
of the dictionary for that word to obtain the compression code and then scanning the compressed file for that split for occurrences of that code. The document offset index is used to determine the relative document number or identifier for the resultant list.

The dictionary method of compression provides a fast way of determining when search words do not occur in the text file at all and hence may provide fast responses to AND queries in which one of the terms does not occur. Searches for the subsequent words in an AND query may be limited to the document matches for the first word using the document index.

For OR queries it is always necessary to do a separate search for each word and append the lists. In summary then, a query involving the OR of \( n \) words may take \( n \) times as long as a single word whereas the AND of \( n \) words may be much faster.

6. Conclusions

One interesting side effect of applying this type of compression is that the resultant files are themselves quite amenable to further compression by other methods. In particular, using the UNIX utility gzip we were able to further compress the dictionary files by a ratio of about 2:1. This additional compression must be undone before searching can be done but does increase savings for disk storage and transfer rates.

The method described in this paper provides a relatively quick way of searching larger text files in their native format. This method can be extended easily to much larger files with a linear growth in set-up time.

REFERENCES