University of Alberta

A Unified Framework for Searching Multiple Compressed Collections

by

Stanley R. M. Oliveira
Mario A. Nascimento

Technical Report TR 01-17
October 2001

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

A Unified Framework for Searching Multiple Compressed Collections

Stanley R. M. Oliveira[1,2]
oliveira@cs.ualberta.ca

Mario A. Nascimento [2]
mn@cs.ualberta.ca

[1]Embrapa Information Technology, Campinas, Sao Paulo, Brazil
[2]Department of Computing Science, University of Alberta, Canada

Abstract

In this work, we propose a new framework for searching directly through multiple compressed collections. We combine, in a single framework, four techniques for efficient, low-overhead text retrieval: inverted index compression, block addressing, sequential searching in compressed text, and a mechanism for merging and ranking the query results from different collections altogether. This single framework can reduce the storage overhead as well as search time, and can be applied to several scenarios where there is frequent access but not as many updates to textual documents, e.g., digital libraries.

1 Introduction

In this paper, we propose a new framework for searching directly in multiple compressed collections, which combines four techniques for efficient, low-overhead text retrieval in a single scheme: inverted index compression, block addressing, sequential searching in compressed text, and a mechanism for merging query results from multiple collections in order to compute a single rank. To our knowledge such an unified approach has not been considered so far in the literature.

The most important issues of our framework are that the text as well as the index can be compressed to reduce space requirements. Additionally, the occurrence list's overhead is reduced not only because our framework uses block addressing, but also because it is compressed. To search for patterns, our framework uses a sequential algorithm since the occurrence list being implemented using block addressing. The other relevant feature of our framework is the mechanism for merging multiple collections. This mechanism depends on the ranking of each collection and uses additional information (maintained within the framework) to merge the final individual ranked responses from different collections in a single and globally ranked way.

Our preliminary experience showed that our framework is able to achieve a very good compression ratio, specially when more than one collection is considered. Apart from that, the amount of time required to search through compressed files is small enough to keep the text compressed all the time. We believe that our framework can be applied to catalogs, bibliographic files and other systems that require frequent access but infrequent updates.

This paper is organized as follows. Related work is reviewed in Section 2. Section 3 discusses some text compression methods focusing on a compression technique based on Huffman coding on words. We then present in detail the framework proposed in this paper in Section 4. Finally, Section 5 closes the paper with conclusions about the presented framework.

2 Related Work

There have been many efforts aimed at searching for information through compressed text. Some of theoretical nature, e.g., [9, 2, 1, 3, 17] and others of a more practical value. We focus on the latter category.

Manber [12] proposed a new text compression scheme to speed up string-matching by searching through compressed files directly. This scheme did not require any modification of the string-matching algorithm so that any string-matching procedure could be used. For typical text files, Manber achieved about a 30% reduction of space and slightly less in search time. The drawback of this scheme is that it is convenient to use only for small-to-medium scale files. Manber and Wu [13] used this scheme in the development of Glimpse, a tool to search through entire file systems. The basic idea of Glimpse is to partition the information space (e.g., a personal file system) into blocks and build an index that is similar to an inverted index, but much smaller than most full-text indices. Glimpse also achieves a 30% reduction in space.

Another compressed text search engine, called SASE, was proposed by Varadarajan and Chiueh [20]. This tool is able to directly search through compressed text without explicit decompression. The sum of the compressed file size and the inverted indices, in SASE, is between 55-76% that of the original database, while the search performance is comparable to a fully inverted index.

Moura et al. [15] proposed a software package to search through compressed files called Cgrep. It works as an exact and approximate compressed matching tool for a large text collection, and can be used to improve indexed schemes that combine inverted file and sequential search, e.g., Glimpse. The compression scheme of Cgrep is a variant of the word-based Huffman code [7, 14]. For exact search through compressed text, Cgrep uses the Boyer-Moore family algorithm [4], which has a good practical performance. For approximate searching, Cgrep uses the automaton-based algorithm discussed in [15]. This algorithm is able to deal with complex queries which consist of complex patterns, such as ranges, complements, wild cards, and arbitrary regular expressions.

Other related effort, discussed in [16], focuses on a single scheme which includes text compression, block addressing and index compression. In this approach, the authors introduced a new framework in which the index is built by using an in memory compression scheme that improves the performance by reducing the I/O cost during index construction. The authors devoted special attention to sequential search on compressed text by comparing different algorithms for simple queries, and even for complex ones. They also used some case studies to evaluate compression ratio versus space overhead, search time, and text updating. Using this scheme, the overall space used by the compressed text as well as by the compressed index take nearly one third of the original uncompressed files without index.

Our work is an extension of the work presented in [16] in sense that we also use a scheme which includes text compression, index compression, and block addressing. However, our work differs from the related work as follows. First, our framework is able to search for information through multiple compressed collections and this has not been addressed so far. Second, our approach takes into account a single dictionary composed of all distinct terms from multiple collections in order to further reduce space requirements. The inverted file of each collection shares the same dictionary. Another difference of our framework from the related work is that we consider the vector model [4] which allows the search results match the query exactly or partially using a ranking scheme. In addition, our framework contains a mechanism for merging the results obtained from querying multiple collections. This mechanism considers the ranking for each collection and uses additional information (which is maintained in the framework) from the collections in order to merge the query results, and output a single rank.

3 Text Compression Methods

There are many compression methods in common use today [19, 6]. Ziv-Lempel [24], an important family compression method, replaces a sequence of terms with a pointer to a previous occurrence of that sequence. Because the pointer requires less space than that phrases, the compressed file is much smaller than the original one. LZW [17] is a variant of Ziv-Lempel taking advantage of the repetition of strings in the data.

Arithmetic coding [23] is another well-known, but CPU-intensive, compression method. Arithmetic coding completely bypasses the idea of replacing an input symbol with a specific code. Instead, it takes a stream of input symbols and replaces it with a single floating point output number. The longer (and more complex) the message, the more bits are needed in the output number.

As discussed in [25], the Ziv-Lempel, LZW and Arithmetic methods present two disadvantages in an information retrieval environment. First, they require that decoding starts at the beginning of a compressed file, making the access to terms expensive. Second, it is very complex to search efficiently for information in compressed files which use one of these methods. The reason is that the compressed files are difficult to search without decompressing the files. On the other hand, these compression methods do not need to store a table of symbols as some other compression methods do (e.g., Huffman's), but this plays a little importance in information retrieval applications because the vocabulary of the full text is required for indexing and querying purposes.

Another well-known coding method is Huffman coding. The strategy is to compress the text by assigning shorter codes to symbols with higher frequencies. Huffman coding is more attractive when compared to Ziv-Lempel, LZW and Arithmetic code because of its simplicity. As discussed in [25, 15, 16], word-based Huffman coding is the most effective compression technique used in information retrieval systems. Words are the atoms or single terms on which most information retrieval systems are built.

In word-based Huffman coding, compression occurs in two passes over the text. In the first pass, the encoder obtains the frequency of the terms, while in the second, the encoder performs the actual compression. The terms with highest frequencies are located in the root of the Huffman tree, and the ones with low frequencies are located in the next nodes. The leaves contain the terms and separators of a text.

To decompress a text, the algorithm traverses the stream of bits in the compressed file starting at in the root. When the algorithm reaches a leaf node, it prints the corresponding term and starts the tree traversal again.

The compressed file generated is a sequence of bits in which each term is a bitstream starting at in the root of the Huffman tree. Figure 1 illustrates compression using Huffman Coding. In this case, the set of symbols is {"a", "text", "file", "is"} whose frequencies are 2, 1, 2, 1 respectively.

Although the original method proposed by Huffman [11] is mostly used as a binary code, a good alternative is to associate a sequence of whole bytes with each term in the text. This technique is very convenient for achieving a very good compression ratio by using only simple operations with integers. One such technique is called Tagged Huffman Code. In this case, one term can be encoded in a sequence of at most 3 bytes instead of a sequence of bits. This strategy uses only the 7 lower-order bits of each byte so that the Huffman tree has degree 128. The most significant bit in the last byte of each encoded term is used to indicate the ending of the codeword, i. e., we add 128 to the last byte of each term (this bit is the "tag"). For instance, let us suppose that the term *text* is compressed into 2 bytes (8 112). The encoder sets the most significant bit of the last byte to 1, i.e., the last byte has the value $240 = 112 + 128$. Note that (8 112) is the logical representation of the term *text*, while (8 240) is its physical representation.

As discussed in [15], there is no degradation of the compression ratio when using bytes instead of bits. However, decompression of byte Huffman code is faster than decompression of binary Huffman

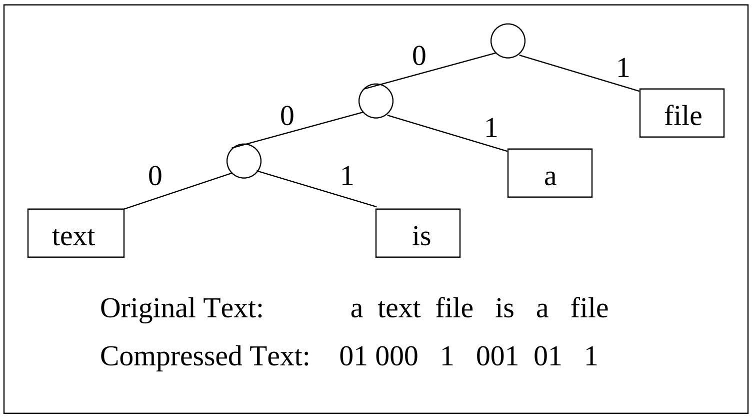

Figure 1: Compression using Huffman Coding

code. The reason is that processing of bytes is much faster than processing of bits because bit shifts and masking operations are not necessary during decoding time or search time.

Another advantage for using byte Huffman code is that this method can be integrated with inverted files. By translating terms as symbols, the table of symbols in the compression coder becomes the text vocabulary, thus allowing a natural integration of an inverted file and a word-oriented Huffman method, as illustrated in Figure 2. This figure shows the idea used in our framework for the patterns *text*, a term coded in two bytes and *file*, a term coded in tree bytes. Each time the algorithm finds a sequence of bytes like that in the compressed text, it reaches the corresponding leaf in the Huffman tree and reports the occurrence.

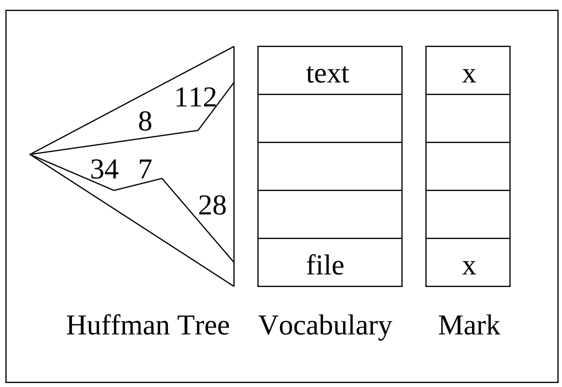

Figure 2: Integration of Huffman Tree and Inverted File

4 The Framework for Searching Multiple Compressed Collections

Figure 3 shows the structure of our framework. As aforementioned, this single scheme encompasses four techniques for efficient, low-overhead text retrieval: inverted index compression, block addressing, sequential searching in compressed text, and a mechanism for merging and ranking the query results from different collections.

In this framework, each collection has its corresponding compressed file using the byte-oriented Huffman coding compression method presented in Section 3. In addition, each collection has a corresponding Huffman tree whose leaves are uncompressed terms located in the single dictionary. In the next subsections, we will discuss the four techniques mentioned above, and how every Huffman tree can profit of the single dictionary in a simple and effective way.

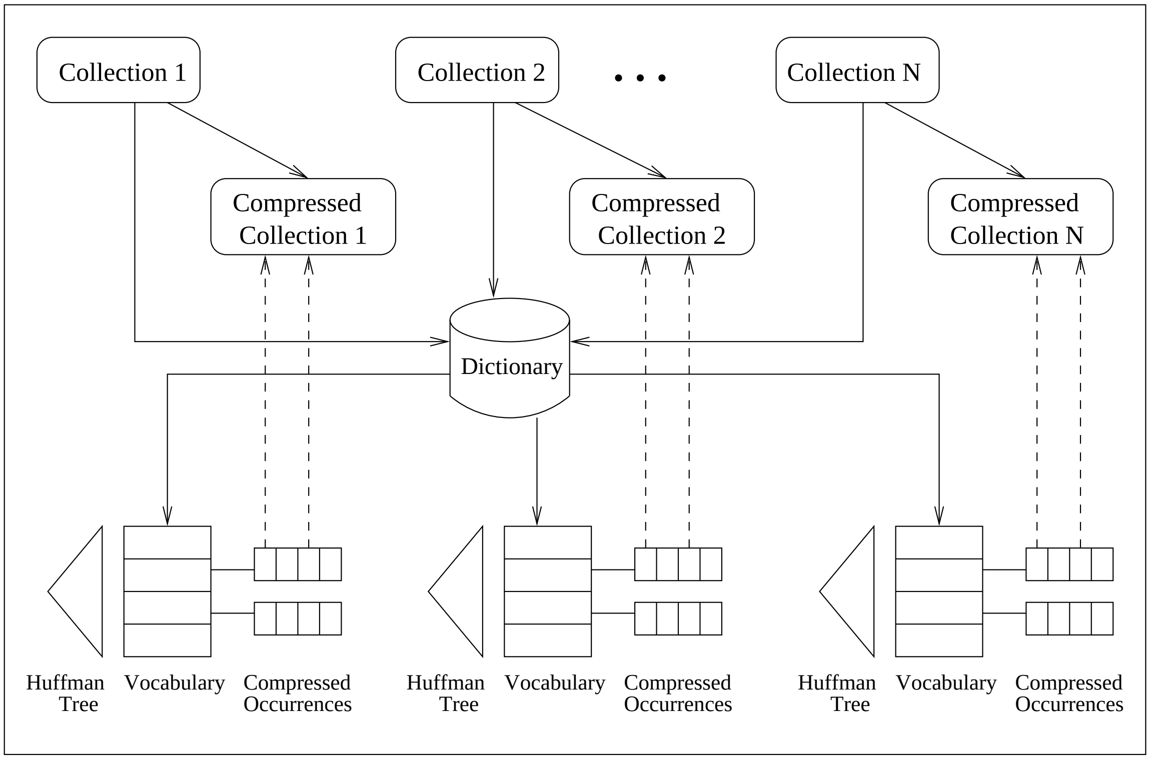

Figure 3: The Framework Structure

4.1 Single Dictionary Updating

As can be seen in Figure 3, our framework contains a dictionary which has all terms from different collections. The dictionary can be stored in memory by using a hash table and it can be updated dynamically as follows: the dictionary receives all the distinct terms from Collection 1, after which it receives all terms from Collection 2 that do not already exist in the dictionary. This process continues until collection N is processed. After updated, a single dictionary is available combining terms from all collections.

4.2 The Inverted Index Compression

Until now, we have discussed how to compress a collection in an efficient way by using byte-oriented Huffman compression. However, indices can be compressed as well. In [4], various kinds of indices are discussed in details. Some indices can be identified according to their granularity, e.g.: *Full inverted index* (this scheme stores the exact positions of each term in the text); *Inverted file index* (this type of index stores the documents in which each term appear); and *Block addressing index* (it divides the text into blocks of fixed size which can contain many documents, part of documents, or even overlap with document boundaries). More detailed analysis about these types of indices can be found elsewhere [16, 22]. Nevertheless, we can combine any of these three kinds of inverted index with index compression.

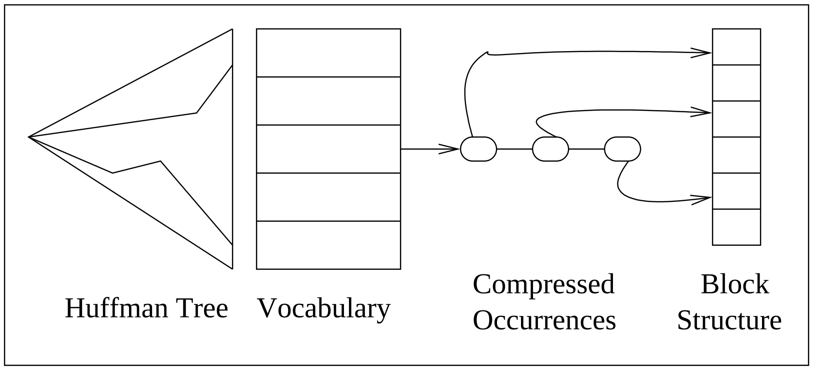

Figure 4: Index Structure for our Framework

The structure of our index is illustrated in Figure 4. Our index is an extension of the idea presented in [16], since we use the same indexing scheme proposed in that paper, i.e., text compression, block addressing, and index compression. The main difference is that our framework is able to search for information through multiple collections, besides merging and ranking the query results for multiple collections.

The physical structure of our index in memory (integrating Huffman tree, vocabulary and occurrences using block addressing) is presented in Figure 5, considering the terms *text* and *file*. Again, the leaves of the Huffman tree is the vocabulary of one collection. We maintain each term of the vocabulary compressed and a pointer to the dictionary which has the terms uncompressed. In addition, we also have the occurrence list for each term.

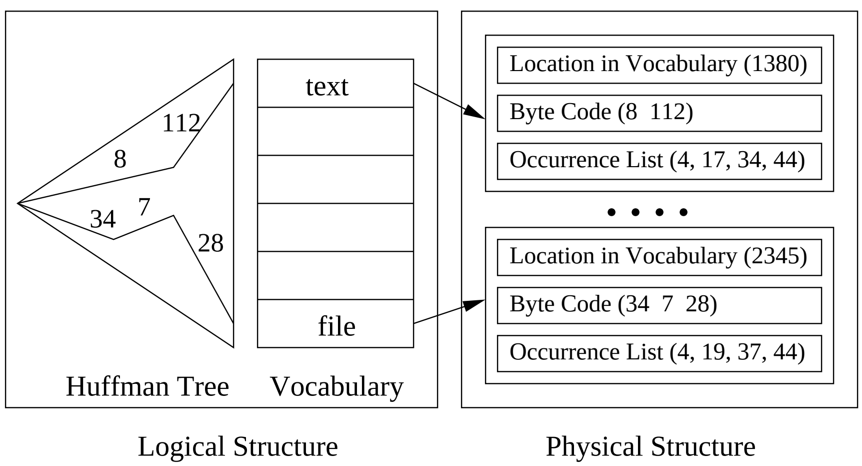

Figure 5: Logical and Physical Structure of our Index

We implemented this index structure (in memory) as well as three compressed files (in disk) which are created by our framework. This strategy may increase slightly the space overhead but the search time is reduced dramatically. Let us consider an example for the Time Collection[1]. The file `time.bhc` (*bhc* stands for byte Huffman code) contains the compressed text assigned to the Time collection. The occurrence list of the compressed inverted index is the file `time.inv` and the file `time.idx` contains pairs (position_term$_i$, size_term$_i$) where position_term$_i$ represents the position of the term$_i$ in the Huffman

[1] A collection of articles from the magazine Time. It can be found at `http://www.dcs.gla.ac.uk/idom/ir\_resources/test\_collections`

tree and size_term_i is the size of the occurrence list. In other words, it is the number of blocks in the occurrence list assigned to the term_i.

Below we sketch the sequence of steps, repeated for each collection, that we need to build the compressed inverted index proposed in our framework.

1. Computing global information (this step builds the vocabulary of one collection and computes the frequency of all of its terms).

2. Building occurrence lists and compressing the Collection.

3. Merging (and bookkeeping) occurrence lists.

We should stress that by using this compression technique, we face one problem when one collection is updated: the frequency of the words change, and consequently, the structure of the Huffman tree must be changed. Therefore, all collections must be recompressed. However, we assume this is to happen seldomly, since the most frequent activity performed is the searching through the compressed collection.

4.3 Searching Through Multiple Compressed Collections

In this section, we discuss how to search for information through compressed collections. Compression using Huffman method yield an interesting property, it allows we search through compressed text in the similar manner as we would search uncompressed text.

There are some efforts related to searching through compressed collections using Plain Huffman code and Tagged Huffman code [15, 25]. When a user submits a query, the query pattern is in compressed form but the collection is in uncompressed form. A phrase pattern is a sequence of elements, where each element is either a simple term or a sequence of terms.

Searching for phrase patterns is more complicated. In this case, we can use two strategies. The first one is to concatenate the codewords of all terms of the phrase and search the result as a single pattern. Other alternative is to search for each single term in the query and then merge the query results. The latter requires a larger computational cost than the former. The process for searching for information using Tagged Huffman code, can be summarized as follows:

1. Preprocessing, which includes obtaining the query pattern compressed using the same compression technique for compressing a collection.

2. Searching: Our framework can use any known algorithm to process the search, e.g., In [15], Moura et al. used the Boyer-Moore algorithm [4], which has a good practical performance. However, as the occurrence list in our framework uses block addressing other algorithms for sequential search can be also used.

3. Ranking: The query results are ranked and stored in memory to support the merging and single ranking of all collections.

Although we have used Tagged Huffman code to implement our framework, there are other alternatives to search through compressed text. For instance, Plain Huffman code, which uses 8 bits of each byte, achieves a better compression ratio than Tagged Huffman code. This method can handle complex queries (e.g. sequences of patterns, each of which can be derived from a simple term or complex regular expressions, allowing errors [15]. Another approach combines both Tagged Huffman and Plain Huffman code. The basic idea is that this automaton-based algorithm can be used for enhanced searching, allowing insertions, deletions, or replacement of single terms. For instance, the pattern "*searching multiple* *compressed collections*" could find the occurrence of "*searching a number of compressed collections*" in the text with one replacement error, assuming that the stopwords "*a*" and "*of*" are disregarded. Considering that this automaton searching is always slower than direct searching, one should use it only for complex queries. This strategy is fully covered in [15, 25].

We chose the Tagged Huffman code method to search through compressed text because of the following reasons: (1) the simplicity of this method (it deals with only integer calculations), (2) This compression scheme achieves a very good compression ratio, and (3) this approach fits well with any sequential algorithm.

Finally, we should point out that we do not need to decompress a whole compressed collection before displaying the results of a query. Instead, it decompresses only parts of the compressed text where the occurrence of matches exists.

4.4 The Mechanism for Merging and Ranking Multiple Collections

Ranking multiple collections is still a problem which deserves special attention in information retrieval systems area. The main issues with this problem, also known as *collection fusion* are the follows [18, 10] : (a) the sources use different ranking algorithms; (b) the ranking algorithm used by the sources are unknown; and (c) the parameters used with these algorithms (e.g., inverse document frequencies or term frequencies cannot be obtained from the sources.

Another relevant problem related to collection fusion is that different collections are likely to have different semantics (and consequently vocabulary). For instance, one collection may contain medical information, another has bibliographical references, other may contain the Bible (King's James version), and so on.

There have been some efforts towards solving these problems [10, 8, 21], especially when different sources as well as different information retrieval models are considered. For instance, some collections still use boolean model whereas others use vector model. In addition, some of these sources may have a ranking strategy whereas others may not. In our approach, this problem is minimized since our framework considers the same ranking strategy for each collection individually. We use the TF-IDF model [4] in which the query results of each collection is ranked according to their degree of similarity.

To compute the ranking of one collection, we first need to specify how index term weights are obtained. The basic idea is that similarity can be quantified by measuring the raw frequency of a term k_i inside a document d_j. Such term frequency refers to the tf factor and provides one measure of how well that term describes the pattern content. On the other hand, the dissimilarity is quantified by measuring the inverse of the frequency of a term k_i among the documents in the collection. This factor refers to the inverse document frequency or the *idf* factor. The reason for usage of an *idf* factor is that a term which appears in many documents are not very useful for distinguishing a relevant document from a non-relevant one. Details about computing the weights for each term within the documents, as well as how to measure similarity between query and documents can be found in [4].

There are various models proposed for the collection fusion problem in the literature [18, 5, 8]. The only prerequisite for using any one of these models is that each collection should provide some parameters, e.g.:

- The ranked result list. In our case, we use the TF-IDF scheme.
- Length of the result list.
- Document frequencies (DF).
- Similarity scores for every delivered document in the result list.

- Collection frequencies (CF), i.e., the number of non-empty result list.

The above parameters are basics for merging query results into a single rank. The common prerequisite is that the partial result lists are ordered by relevance rank. In [18], Steindinger experimented with five different models for the collection fusion problem and found the following ones to be more effective in terms of Precision and Recall, despite their apparent simplicity:

Round Robin Random (RRR) – This model has one prerequisite, the length of the result lists should be known. One must build one array of length $\sum_{k=1}^{|C|} l_k$ where the sum goes over all the lengths l_k of the result lists. Then one toss a dice with $\sum_{k=1}^{|C|} l_k$ faces which will yield get an index to collection j with probability $P_j = \frac{l_k}{\sum_{k=1}^{|C|} l_k}$ where |C| is the number of searched collections. Thus, the first document of the list hit is removed and the procedure starts again with the current result lists.

Round Robin Block (RRB) – This model divides the length of all result lists by the length of the shortest non-empty list, denoted by L, that is, the length of the result lists is also one prerequisite for this model. One should remove from each result list L documents in a round robin fashion and put these documents into the global result list. The next round starts with the new shorter result lists, but with the same L value. Should L be bigger than the length of an actual list at the end, the whole list is taken into the global result list.

It is important to note that our framework provides all the parameters for the above models, as well as for others not reviewed here but presented in [18]. However, we acknowledge that our framework may yield different results from those obtained by Steindinger. The reason is simple, unlike in his work, we use the same ranking algorithm for all collections. How much this can influence the behavior of the final rank is unknown at this point. Nevertheless we believe the results from [18] can serve as an starting point for our single ranking approach.

5 Conclusion

In this paper, we proposed a new framework for searching directly in multiple compressed collections, which combines four techniques for efficient, low-overhead text retrieval in a single scheme: inverted index compression, block addressing, sequential searching in compressed text, and a mechanism for merging query results from multiple collections in order to compute a single and meaningful rank. Our approach also takes into account a single dictionary composed of all distinct terms from multiple collections in order to reduce space requirements. The inverted file of each collections shares the same dictionary.

Our experience with compression techniques has showed that our framework is able to achieve a very good compression ratio. The compressed text for one collection, plus a compressed inverted index built on it, takes no more than 50 percent of the original collection size without any index.

Another issue is that index construction, including text compression, is faster than the indexing of the uncompressed text and reduces a great deal of space overhead for various text collections.

Our sequential search scheme performs faster than if is the search system were based on uncompressed text. We have observed that the amount of time required to search through compressed collections is satisfactory to keep the collections compressed all the time.

Finally, our mechanism for merging query results into a single rank can be accomplished by using any one of the five models presented in [18].

The compressed method as well as the mechanism for merging query results allow an optimal integration of massive data into text databases, so that this framework can be applied to several scenarios where there is frequent access, but not as many updates, to textual documents, e.g., digital libraries.

6 Acknowledgments

Stanley R. de M. Oliveira was partially supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) of Ministry for Science and Technology of Brazil, under Grant No. 200077/00-7. Mario A. Nascimento was partially supported by a Research Grant from NSERC, Canada. We acknowledge Long Li's collaboration in early stages of this work.

References

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proceedings of the Data Compression Conference*, pages 279–288, IEEE, New York, USA, 1992.

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. In *Proceedings of the 3rd Symposium on Discrete Algorithms*, pages 440–452, SIAM, Philadelphia, PA, USA, 1992.

[3] A. Amir, G. Benson, and M. Farach. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. In *Proceedings of the 5th Symposium on Discrete Algorithms*, pages 705–714, SIAM, Philadelphia, PA, USA, 1994.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley Longman, 1999.

[5] Christoph Baumgarten. A Probabilistic Solution to the Selection and Fusion Problem in Distributed Information Retrieval. In *ACM SIGIR Conference*, pages 246–253, 1999.

[6] T. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, Englewood, NJ, USA, 1990.

[7] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[8] J. P. Callan, Z. Lu, and W. B. Croft. Searching Distributed Collections with Inference Networks. In *Proceedings of the 18th Annual SIGIR Conference*, pages 21–28, Seattle, WA, USA, 1995.

[9] T. Eilam-Tsoreff and U. Vishkin. Matching Patterns in Strings Subject to Multilinear Trasformations. *Theoretical Computer Science*, 60(3):231–254, December 1988.

[10] L. Gravano, C. K. Chang, H. Garcia-Molina, and A. Paepcke. STARTS: Stanford Proposal for Internet Meta-Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 207–218, Tucson, AZ, USA, May 1997.

[11] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, 1952.

[12] U. Manber. A Text Compression Scheme that Allows Fast Searching Directly in the Compressed File. *ACM Transactions on Information Systems*, 15(2):124–136, 1997.

[13] U. Manber and S. Wu. GLIMPSE: A Tool to Search Through Entire File Systems. In *Proceedings of the USENIX, Winter 1994. Technical Conference*, pages 23–32, San Francisco, CA, USA, 1994.

[14] A. Moffat. Word-Based Text Compression. *Software - Practice Experience*, 19(2):185–198, February 1989.

[15] E. S. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and Flexible Word Searching on Compressed Text. *ACM Transactions on Information Systems*, 18(2):113–139, April 2000.

[16] G. Navarro, E. S. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding Compression to Block Addressing Inverted Files. *Information Retrieval*, 3(1):49–77, 2000.

[17] M. Nelson. LZW Data Compression. *Dr. Dobb's Journal, (October)*, pages 29–37, 1989.

[18] A. Steidinger. Comparison of different Collection Fusion Models in Distributed Information Retrieval. In *Proceedings of the First DELOS Workshop on Information Seeking, Searching, and Querying in Digital Libraries*, Zurich, Switzerland, December 2000.

[19] J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, MD, USA, 1988.

[20] S. Varadarajan and T. Chiueh. SASE: Implementation of a Compressed Text Search Engine. In *Proceedings of the First USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, December 1997.

[21] E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird. The Collection Fusion Problem. In *Proceedings of the 3rd Text Retrieval Conference (TREC-3)*, 1995.

[22] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* New York: Morgan Kaufmann Publishers, second edition, 1999.

[23] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[24] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[25] N. Ziviani, E. S. Moura, G. Navarro, and R. Baeza-Yates. Compression: A Key for Next-Generation Text Retrieval Systems. *IEEE Computer Society*, 33(11):37–44, 2000.