Compressing Inverted Files using Modified LZW

Vasileios Iosifidis and Christos Makris
Department of Computer Engineering and Informatics, University of Patras, Rio 26500, Patra, Greece

Keywords: Inverted File, Compression, LZ78, LZW, GZIP, Binary Interpolative Encoding, Gaps, Reorder, Searching and Browsing, Metrics and Performance.

Abstract: In the paper, we present a compression algorithm that employs a modification of the well known Ziv Lempel Welch algorithm (LZW); it creates an index that treats terms as characters, and stores encoded document identifier patterns efficiently. We also equip our approach with a set of preprocessing {reassignment of document identifiers, Gaps} and post-processing methods {Gaps, IPC encoding, GZIP} in order to attain more significant space improvements. We used two different combinations of those discrete steps to see which one maximizes the performance of the modification we made on the LZW algorithm. Performed experiments in the Wikipedia dataset depict the superiority in space compaction of the proposed technique.

1 INTRODUCTION

Inverted files are considered to be the best structures for indexing in information retrieval search engines (Baeza-Yates and Ribeiro-Neto, 2011, Büttcher, Clarke and Cormack 2010). The main problem one has to tackle with them in an information retrieval system is that when the number of documents in the collection increases, the size of data indices grows significantly, hence scalability in terms of efficient compression techniques is a mandatory requirement.

At present, in the large-scale information retrieval systems, the term-oriented inverted index technology is commonly used. An inverted index consists of two parts: a search index storing the distinct terms of the documents in the collection, and for each term a list storing the documents that contain these terms. Each document appears in this list either as an identifier or it is accompanied with extra information such as the number of appearances of the term in the document. When only the document identifiers appear then the list is usually an ascending list of identifiers so that it can be easily compressed ((Baeza-Yates and Ribeiro-Neto, 2011, Büttcher, Clarke and Cormack 2010, Witten, Moffat and Bell, 1999).

In this paper we try to envisage a new compression scheme for inverted files that is based on an elegant combination of previously published solutions. In particular we try to find common appearances inside the terms, and store those common appearances as encodings that require less space. Our basic idea comes from the most widely used algorithm, the LZ78 (Ziv et al. 1978), LZW (Welch and Terry, 1984) and a method which reassigns the document identifiers of the corpus (Arroyuelo et al. 2013).

As we will describe later, the modified LZW is trying to find patterns inside the inverted file and encodes them into numbers. The reassign method (Arroyuelo et al. 2013) helps us to keep the encoded values ‘small’ and also it produces patterns for our method to find. After the reassignment the document identifier values require fewer digits for representation because the range they required previously was larger.

In section 2 we describe related work. In section 3 we present an analysis of the methods we used. In section 4 we present our ideas, the modifications we made and an example of how it works. In section 5 we present the results which came from experimentation on the Wikipedia’s dataset and also we compare this technique with the one which produces arithmetic progressions (Makris and Plegas, 2013) and is considered to achieve compression more effective than previous techniques. In section 6 and 7 we explain our experiments and conclude with future work and open problems.

Iosifidis, V. and Makris, C.
Compressing Inverted Files using Modified LZW.
In *Proceedings of the 12th International Conference on Web Information Systems and Technologies (WEBIST 2016) - Volume 1*, pages 156-163
ISBN: 978-989-758-186-1
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 RELATED WORK

Many compressing file methods have been proposed in the scientific bibliography. The majority of these compression methods use gaps, between document identifiers (DocIds), in order to represent data with fewer bits. The most well-known methods for compressing integers are the Binary code, Unary code, Elias gamma, Elias delta, Variable-byte and the Golomb code (Witten, Moffat and Bell, 1999). Over the past decade, there have been developed some methods which are considered to be the most successful. These methods are: Binary Interpolative Coding (Moffat and Stuiver, 2000) and the OptPFD method in (Yan et al., 2009) that is an optimized version of the techniques appearing in (Heman, 2005; Zukowski, Heman, Nes and Boncz, 2006) and that are known as PforDelta family (PFD).

The Interpolative Coding (IPC) (Moffat and L. Stuiver, 2000) has the ability to code a sequence of integers with a few bits. Instead of creating gaps from left to right, compression and decompression are done through recursion. Consider a list of ascending integers < a_1, a_2… a_n>. IPC will split the list into two other sub lists. The middle element, $a_{n/2}$, will be binary encoded and the first and last element of the list will be used as boundary of bits. The least binary representation would require log ($a_n – a_1 + 1$) bits. The method runs recursively for the two sub lists.

Some methods for performance optimization are pruning-based (Ntoulas and Cho, 2007; Zhang et al., 2008). Some other methods try to take advantage of closely resembling that may exist between different versions of the same document in order to avoid size expansion of the inverted indexes (He, Yan and Suel, 2009; He and Suel 2011). Another method is storing the frequency of appearances from document identifiers of various terms and also the term positions (Akritidis and Bozanis, 2012; Yan et al., 2009).

Moreover, there is a variety of researches which focus on the family of LZ algorithms. The statistical Lempel-Ziv algorithm (Kwong, Sam, and Yu Fan Ho, 2001) takes into consideration the statistical properties of the source information. Also, there is LZO (Oberhumer, 1997/2005) which supports overlapping compression and in-place decompression.

Furthermore, there is one case study where the most common compressing methods were applied for evaluation of a hypothesis that the terms in a page are stochastically generated (Chierichetti, Kumar and Raghavan, 2009). In parallel, there is a recent method which converts the lists of document identifiers as a set of arithmetic progressions which consist of three numbers (Makris and Plegas, 2013). Finally, Arroyuelio et al. (2013) proposed a reassignment method that allows someone to focus on a subset of inverted lists and improve their performance on queries and compressing ratio.

In our approach we used a combination of methods which we will describe below. We compared our method with a recent method (Makris and Plegas, 2013) which has very good compressing ratio. In section 3 we present two different combinations of the methods we used so as to evaluate the behavior of the modification we made and to see which one achieves the maximum compressing ratio.

3 USED TECHNIQUES

In our scheme we employed several algorithmic tools in order to produce better compressing ratios. The tools we used for this purpose are described below.

3.1 LZ78 Analysis

LZ78 (Ziv et al. 1978) algorithms achieve compression by replacing repeated occurrences of data with references to a dictionary that is built based on the input data stream. Each dictionary entry is of the form dictionary[...] = {index, character}, where index is the index to a previous dictionary entry, and character is appended to the string represented by dictionary[index]. The algorithm initializes last matching index = 0 and next available index = 1. For each character of the input stream, the dictionary is searched for a match: {last matching index, character}. If a match is found, then the last matching index is set to the index of the matching entry, and nothing is output. If a match is not found, then a new dictionary entry is created: dictionary [next available index] = {last matching index, character}, and the algorithm outputs last matching index, followed by character, then resets last matching index = 0 and increments next available index. Once the dictionary is full, no more entries are added. When the end of the input stream is reached, the algorithm outputs last matching index.

3.2 LZW Analysis

LZW (Welch and Terry 1984) is an LZ78-based algorithm that uses a dictionary pre-initialized with all possible characters (symbols), (or emulation of a pre-initialized dictionary). The main improvement of LZW is that when a match is not found, the current input stream character is assumed to be the first character of an existing string in the dictionary (since the dictionary is initialized with all possible characters), so only the last matching index is output (which may be the pre-initialized dictionary index corresponding to the previous (or the initial) input character).

3.3 Reassignment Analysis

DocIds which are contained inside an inverted file may be large numbers using many bytes to be stored. Using the reorder method (Arroyuelo et al. 2013), all the DocIds are reassigned as different numbers in order to focus on a given subset of inverted lists to improve their performance on querying and compressing. For example consider a term < T1 >, which contains DocIds: [100, 101, 1001, 1002, 1003]. So the step that is going to be applied will re-enumerate all the DocIds inside the term and the whole corpus of the inverted file. After the reorder the result would be:
100 → 1, 101 → 2, 1001 →3, 1002 → 4, 1003 →5

So the term would be like:

<T1> = [1, 2, 3, 4, 5]

This step helps us reduce the Gaps between DocIds, and reduce some space of the Inverted File. Another use of this method is to decrease the starting encoding value of the modified LZW. So we also use the reassignment method to decrease the encoded values.

We also used a modified reassignment method (Arroyuelo et al. 2013) where we reordered the pages based on the term intersections. For example consider a term < T1 >, < T2 >, which contain DocIds: [100, 105, 110, 120] and [29, 100, 105, 106, 107, 110, 120, 400] respectively. Now if we use the first reorder method the result would be:
100 → 1, 105 → 2, 110 →3, 120 → 4, 29 → 5, 106 → 6, 107 → 7, 400 → 8
So the terms < T1 > and< T2 > would be like:

< T1 > = [1, 2, 3, 4]
< T2 > = [5, 1, 2, 6, 7, 3, 4, 8]

Now if we apply the second reorder method the encoding would be the same in this case but the output of the terms would be like this:

< T1 > = [1, 2, 3, 4]
< T2 > = [1, 2, 3, 4, 5, 6, 7, 8]

We use this method (Arroyuelo et al. 2013) in order to create more repeated patterns for the modified LZW. The above example shows that if we used first method < T1 > and < T2 > would not have the same nth elements in common. The modified LZW has a good compressing ratio when it's locating common sequences. So the second method produces sequences which are common to the previous lists. In this example, if the first method is applied it will produce more encoding values than the second method (assumed we have already encoded the unique pages inside the index). After we applied this technique, we noticed a slight improvement between the compressed files, of the modified LZW, for the two reordering methods.

3.4 GZIP Analysis

GZIP (Witten, Moffat and Bell, 1999) is a method of higher-performance compression based on LZ77. GZIP is using hash tables to locate previous occurrences of strings. GZIP is using Deflate algorithm (Deutsch, L. Peter, 1996) which is a mix of Huffman (Huffman, David A. et al. 1952) and LZ77 (Ziv et al. 1977). GZIP is "greedy"; it codes the upcoming characters as a pointer if at all possible. The Huffman codes for GZIP are generated semi-statically. Because of the fast searching algorithm and compact output representation based upon Huffman codes, GZIP outperforms most other Ziv-Lempel methods in terms of both speed and compression effectiveness.

4 OUR CONTRIBUTION

We present two different schemes with a combination of the described methods. The main intuition behind these methods is based on "greedily" compressing by repetitively applying algorithmic compression schemes. We used these methods because they are considered of being state of the art methods in compression (gap, binary interpolative code, GZIP), so they have been used to achieve the maximum compressing ratio.

One of the pre-processing methods is the reorder of the corpus where we change all the DocIds starting for value 1 and we go on, reassigning all the DocIds of the inverted file. The re-enumeration is done based on one or more lists (Arroyuelo et al. 2013). When we used the re-enumeration based on more than one list, the modified LZW was slightly better than the re-enumeration based on one list. Another step is, after reordering the inverted file, to sort the reenumerated DocIds and store the intervals of them inside the inverted file. This step has a good compression ratio and for it we coin the term *gap method***.** We also use binary interpolative encoding (Moffat and Stuiver, 2000) and GZIP (http://en.wikipedia.org/wiki/GZIP, Witten, Moffat and Bell, 1999) compression.

In the first scheme we have four different methods that we apply. First is reorder method (Arroyuelo et al. 2013), second we apply our modified LZW, third we use binary interpolative coding (Moffat and L. Stuiver, 2000) and last we use GZIP (Witten, Moffat and Bell, 1999). In the second scheme we again use four different methods but this time they are a bit different. Again we apply reorder method as a first step but as a second step we employ the gap method. As a third step we use modified LZW and for last step we use GZIP.

We propose this combination of techniques because they are state of the art. We experimented with other compressing methods such as gamma, delta, Golomb encodings but their results were not as good as interpolation's encoding. Also gap method was easy to implement and it achieved great compressing ratio. Reorder method was primarily used to enhance our modification on LZW. GZIP on the other hand was used to minimize the output so we could achieve the maximum compressing ratio.

Pseudo code describes the steps below. Figure 1 is the first scheme and Figure 2 is the second scheme.

```
//Reorder - Inverted File
reordered_file=Reorder inverted file;
//Modified LZW on reordered_file
mlzw_file=LZW(reordered_file);
//Binary Interpolative Encoding on the //mlzw_file
BIE_file=BIE(mlzw_file);
//GZIP on the BIE_file
final_compressed_file=GZIP(BIE_file);
```

Figure 1: 1st scheme.

```
//Reorder- Inverted File
reordered_file=Reorderinverted file;
//Intervalmethod on the reordered_file
gapped_file=GAP(reordered_file);
//ModifiedLZW on gapped_file
mlzw_file=MLZW(gapped_file);
//GZIP on the mlzw_file
final_compressed_file=GZIP(mlzw_file);
```

Figure 2: 2nd scheme.

On the first scheme, we reorder the inverted file in order to reduce the range of the numbers and create the patterns based on the second approach we described in section 3.3. Then we use the modified LZW and after that we proceed with binary interpolation coding and GZIP in order to reduce the inverted file even more.

On the second scheme we again reorder the inverted file for the same reason as previously but now we use the gap technique. Gap method combined with the reorder method, has great compressing ratio but it makes modified LZW inefficient as we will explain below.

4.1 Modification of LZW

In our algorithm we are using a modification of the LZW. Instead of characters the modified LZW reads DocIds as characters and tries to find patterns inside terms. Furthermore, the LZW has an index which contains letters and numbers and their encoded values which goes till 255, so it starts encoding after 255. We build an index, which in the start is completely empty, that consists of patterns that are found and their encoded number. As we know, the DocIds are webpages which are enumerated. In order to avoid collisions on decompression we must locate the largest number inside the inverted file and take the max (DocIds) + 1, as the starting encoding number of the modified LZW (this is done by the previous step, the re-enumerate step, so we will not have to scan the whole file from the start).

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4.2 Compression with Modified LZW

After we find the maximum document identifier, we are ready to begin building the index. For each term we build a list which contains the DocIds of each term. The algorithm goes: for each DocId in the list check if it exists inside the index. There are two cases:

Case 1: The DocId does not exist inside the index. In this case the DocId is inserted into the index and it is encoded. The index contains pairs of key-values (keys are the DocIds and values are the encoded values of the DocIds). After the insertion to the index the compressor will output the DocId, not the encoded value, to the compressed inverted file, so when the decompressor starts decoding it will build the index the exact way the compressor did.

Case 2: The DocId of the list exists inside the index. In this case we have two sub cases:

- **Sub Case 1:** The current DocId and the next DocId of the list are being united and checked if their union exists inside the index. If their union does not exist then their union is encoded and inserted into the index. The compressor outputs the encoded value of the DocId which exists inside the index and also the next DocId is checked if it is inside the index. If it does not exist then it is inserted into the index. If it exists then we proceed with the next element in the list.

- **Sub Case 2:** The current DocId in union with the next DocId, inside the list, is already stored inside the index. In this sub case the algorithm checks iteratively if the union of DocIds takes the union of the previous step in union with the next DocId inside the list, exists inside the index. It will go on and on till the list finishes or when the union is not stored inside the index. In the first case, when we reach the end of the list, compressor just outputs the encoded value of the union which is already stored inside the index. If the union does not exist then execute **Sub Case 1**.

So for each term we build a list which contains the document identifiers and we check if their unions exist inside the index.

4.3 Decompression with Modified LZW

Decompression works the same way as the compression, by building the index. The encoded values begin from the maximum value of the re-enumerate method. So modified LZW decompressor is creating a list for every term, storing the DocIds or the encoded values of patterns. For each element inside the list it checks if the element is inside the index. Again there are two cases:

Case 1: The element does not exist inside the index and its value is smaller than the bound which separates DocIds and encoded values. So the decompressor will process the element as a DocId. It will encode the element and store it to the index. After the insertion decompressor will output the current list element and continue with the next element inside the list.

Case 2: The element exists inside the index and its value is bigger than the bound's value. In this case the decompressor will know that the element is the encoded value of a DocId or a union of DocIds. Decompressor will get the DocId or the union of DocIds from the index and output it to the file. But the algorithm does not stop here. Decompressor knows that the compressor outputted the encoded value because the union with the next element of the list did not exist into the index. So the outputted value is united with the next element inside the list and the union is encoded and stored into the index. After that, decompressor continues with the next element inside the list.

4.4. Index Creation

As we described in the section 4.1 the pattern matching method we applied is based on building an index. We scan the list of document identifiers of each term and for each element we check if it exists inside the index and then we encode it or search for DocId unions that are not encoded.

In the below example we will show exactly how the compression and decompression algorithms work. Let's assume we have 5 terms T1, T2, T3, T4, and T5 which consist of the below DocIds:

T1: < 1, 2, 3, 4, 5, 9, 10 >
T2: < 1, 2, 3, 4, 5, 9, 10, 14, 17 >
T3: < 1, 2, 3, 4, 5, 9, 10, 17 >
T4: < 1, 2, 3, 4, 5, 6, 7, 8, 21, 23 >
T5: < 1, 2, 3, 4, 5, 6, 7, 8, 21, 23, 29 >

The bound is 29, so the encoding numbers will begin on 30. We run the Modified LZW and we get:

T1: < 1, 2, 3, 4, 5, 9, 10 >
T2: < 30, 31, 32, 33, 34, 35, 36, 14, 17 >
T3: < 37, 32, 33, 34, 35, 36, 42 >
T4: < 43, 33, 34, 6, 7, 8, 21, 23 >
T5: < 46, 34, 48, 49, 50, 51, 52, 29 >

The encoded values of DocIds and unions:

First list

'1': 30, '2': 31, '3': 32, '4': 33, '5': 34, '9': 35, '10': 36

Second list

'1 2': 37, '3 4': 38, '5 9': 39, '10 14': 40, '14': 41, '17': 42

Third list

'1 2 3': 43, '4 5': 44, '9 10': 45

Fourth list

'1 2 3 4': 46, '5 6': 47, '6': 48, '7': 49, '8': 50, '21': 51, '23': 52

Fifth list

'1 2 3 4 5': 53, '6 7': 54, '8 21': 55, '23 29': 56, '29': 57

In this case the data do not seem very compressed because this is a small input, but if the input was gigabytes of DocIds then we could see a difference.

Decompression takes as an input the compressed inverted file and with the same logic (reading the DocIds and building the index) it restores the original inverted file.

5 RESULTS

We ran both the schemes to see which one was better. The machine we used has these specs: AMD PHENOM II X6 1100T 3.3 GHz, 16 GB ram, 1 TB hdd, Linux 12.04 64bit.

We used as Inverted File: Wikipedia 21 GB text file (Callan, 2009). As we noticed on the data set, Wikipedia has almost 6.5 million pages, but the numbers of those pages are not enumerated sequentially. Some pages had numbers bigger than 20 million. So if we used the Wikipedia's page labels the modified LZW would start the compression from the biggest number which would require more digits to be stored inside the compressed file. In order to start from the smallest possible number we used the reorder method.

Table 1: First scheme.

Steps	**Ratio of compression**
Reorder	22%
Modified-LZW (+ above steps)	38%
IPC (+ above steps)	65%
GZIP (+ above steps)	82%

Table 2: Second scheme.

Steps	**Ratio of compression**
Reorder	22%
Gaps (+ above steps)	72%
Modified-LZW(+ above steps)	73%
GZIP (+ above steps)	90%

Ratio is based on the original inverted file (Wikipedia 21 GB) for both Table1 and Table 2.

Modified LZW on the first scheme in table has a 16% ratio which could be improved if we had a machine with more ram, because in our case we had to split the reordered file and run modified LZW for each sub file separately. In total the first scheme had output a compressed file which is 82% smaller than the original Inverted File.

On the second scheme we see that the modified LZW has 1% compressing ratio. Also in this case (second scheme) we had to slice the file to sub files to run Modified LZW faster. So it may have different results if we could build the index for the whole inverted file and not on separately sub files.

A main drawback is the fact that we cannot decompress a specific term. We have to go all the way back decompressing each file, for each step so we can obtain the initial (after reorder method) inverted file. Another drawback of our technique is that the decompression is extremely slow. In compression we use dictionaries where we hash the keys so we can search in constant time for the patterns. In decompression we are using hashes to values too so we can retrieve the keys which are the original values. So in our machine, which lacks of memory for this purpose, ram is used for hashing keys and values. In order to avoid memory overflow, we use external memory, hard disks, to store the key, value pairs. After we reach 90% of ram space we start appending key-value pairs to disk. For every encoded value we have to search inside ram and if it is not there we also have to search inside disk which is very time consuming.

Now we are going to compare these results with a new technique (Makris, and Plegas 2013). The specific construction achieves good compressing ratio and has been used for the same dataset (Wikipedia). This new technique initially converts the lists of DocIds to a set of arithmetic progressions; in order to represent each arithmetic progression they use three numbers. In order to do that, they provide different identifiers to the same document in order to fill the gaps between the original identifiers that remain in the initial representation. They use a secondary index in order to handle the overhead which is produced because of the multiple identifiers that have to be assigned to the documents. They also use an additional compression step (PForDelta or Interpolative Coding) to represent it. The tables 3 and 4 show the experiments which have been done to the Wikipedia's dataset.

Table 3: The compression ratio achieved by the (Makris and Plegas, 2013) algorithms, with the secondary index uncompressed.

	Base	Multiple Sequences	IPC	PFD
Wikipedia	78%	70%	44%	42%

Table 4: The compression ratio achieved by the (Makris and Plegas, 2013) proposed algorithms, when compressing the secondary index.

	Base + IPC	MS + IPC	Base + PFD	MS + PFD
Wikipedia	40%	38%	39%	38%

Table 3 shows the compression ratio which was achieved in relation to the original size for the proposed techniques (when the secondary index is uncompressed) and the existing techniques. Table 4 depicts the compression ratio which was achieved by the compression techniques in relation to the original size when combining the proposed methods with the existing techniques for compressing the secondary index.

As we can see, the algorithm in this paper has a better compression ratio than the algorithm from the recent technique (Makris, and Plegas, 2013). The main difference in these two papers is that in (Makris and Plegas, 2013), they try to find numerical sequences and they are compressing them with the use of PForDelta or Binary Interpolative Encoding. In this paper we are trying to find patterns and then compress them with Binary Interpolative Encoding and GZIP. The reason this paper is better than the previous is because in this paper the algorithm is “greedy” and we are using all the state of the art compression techniques and also we are using GZIP which is not used in the other method (Makris and Plegas 2013). Without GZIP we get 65% and 73% compressing ratios on Wikipedia’s dataset. We also tested the dataset using GZIP as the only compressing method, in order to see if it has greater compressing ratio than our 2 schemes. GZIP compressed the dataset by 72%. It is clear that in both schemes we achieved greater compressing ratio than GZIP alone.

6 CONCLUSIONS

In our schemes we employed a set of pre-processing and compression steps in order to achieve more compression gains than previous algorithms. Let’s explain why we used this order. The reorder step minimizes the range of DocIds so the new inverted file has smaller DocIds inside. Furthermore this step is also helping the modified LZW. If we used the modified LZW on the original inverted file, without the reorder step, then the initial value of the codes would be a bigger number than the reordered inverted file. After, we use the modified LZW to look for ‘word’ patterns inside the inverted file, which has better results on our first scheme rather than the second scheme. More analytically modified LZW ran better in the first scheme because the Gap method changed the structure of the whole inverted file, gaps between the DocIds are not constant, plus codes that modified LZW produces are longer than the actual pattern gaps which are compressed. So in many cases the lists that the Gap method produced had numbers with fewer digits than the encoded values of the modified LZW.

The last two steps, Binary Interpolative Encoding and GZIP are used for a greedy approach. Binary Interpolative Encoding is a very good integer compression method and GZIP is the best compression technique, using Deflate algorithm, so we used them in order to find how much smaller inverted file we can get.

Furthermore, a general disadvantage of the modified LZW is that it demands machines with large amount of main memory. In our experiments we had to slice the re-enumerated inverted file into smaller sub-files because we had memory overflow problems if we used the whole re-enumerated inverted file.

7 FUTURE WORK AND OPEN PROBLEMS

We presented a set of steps that achieve a good compression when handling inverted files. In a further analysis we would like to test the schematics to a larger set of data using stronger machines because we believe that the modified LZW will have better compressing ratio if we could store more patterns inside the index. Furthermore we would like to modify the algorithm so we can compress and decompress separately terms and not the whole set of data. We also want to implement the PForDelta method as a third step instead of Binary Interpolative Encoding; this seems worthwhile since it is expected to improve the performance of our techniques.

REFERENCES

Akritidis, L., Bozanis, P., 2012, Positional data organization and compression in web inverted indexes, *DEXA 2012*, pp. 422-429.

Anh, Vo Ngoc, and Alistair Moffat. "Inverted index compression using word-aligned binary codes." *Information Retrieval 8.1 (2005)*: 151-166.

Arroyuelo D., S. González, M. Oyarzún, V. Sepulveda, Document Identifier Reassignment and Run-Length-Compressed Inverted Indexes for Improved Search Performance, *ACM SIGIR 2013*.

Baeza-Yates, R., Ribeiro-Neto, B. 2011, Modern Information Retrieval: the concepts and technology behind search, second edition, Essex: Addison Wesley.

Büttcher, S. Clarke, C. L. A., Cormack, G. V., 2010, Information retrieval: implementing and evaluating search engines , *MIT Press, Cambridge, Mass*.

Callan, J. 2009, The ClueWeb09 Dataset. available at http://boston.lti.cs.cmu.edu/clueweb09 (accessed 1st August 2012).

Chierichetti, F., Kumar, R., Raghavan, P., 2009. Compressed web indexes. In: *18th Int. World Wide Web Conference*, pp. 451–460.

Deutsch, L. Peter. "DEFLATE compressed data format specification version 1.3." (1996).

He, J., Suel, T., 2011. Faster temporal range queries over versioned text, In the *34th Annual ACM SIGIR Conference*, China, pp. 565-574.

He, J., Yan, H., Suel, T., 2009. Compact full-text indexing of versioned document collections, Proceedings of the *18th ACM Conference on Information and knowledge management*, November 02-06, Hong Kong, China.

Heman, S. 2005. Super-scalar database compression between RAM and CPU-cache. MS Thesis, Centrum voor Wiskunde en Informatica, Amsterdam.

Huffman, David A., et al. A method for the construction of minimum redundancy codes. proc. *IRE, 1952*, 40.9: 1098-1101.

Jean-Loup Gailly and Mark Adler, GZIP Wikipedia [http://en.wikipedia.org/wiki/GZIP]

Kwong, Sam, and Yu Fan Ho. "A statistical Lempel-Ziv compression algorithm for personal digital assistant (PDA)." *Consumer Electronics, IEEE Transactions on 47.1 (2001)*: 154-162.

Makris, Christos, and Yannis Plegas. "Exploiting Progressions for Improving Inverted Index Compression." *WEBIST. 2013.*

Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.

Ntoulas A., Cho J., 2007. Pruning policies for two-tiered inverted index with correctness guarantee, Proceedings of the 30th Annual International *ACM SIGIR conference on Research and development in Information Retrieval*, July 23-27, Amsterdam, The Netherlands.

Oberhumer, M. F. X. J. "LZO real-time data compression library." User manual for LZO version 0.28, URL: http://www. infosys. tuwien. ac. at/Staff/lux/marco/lzo. html (February 1997) (2005).

Welch, Terry (1984). "A Technique for High-Performance Data Compression". *Computer 17* (6): 8–19. doi:10.1109/MC.1984.1659158.

Witten, Ian H., Alistair Moffat, and Timothy C. Bell. Managing gigabytes: compressing and indexing documents and images. *Morgan Kaufmann, 1999*.

Yan, H., Ding, S., Suel, T., 2009, Compressing term positions in Web indexes, pp. 147-154, Proceedings of the *32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Zhang, J., Long, X., and Suel, T. 2008. Performance of compressed inverted list caching in search engines. In the *17th International World Wide Web Conference WWW*.

Ziv, Jacob, and Abraham Lempel. "A universal algorithm for sequential data compression." *IEEE Transactions on information theory 23.3 (1977)*: 337-343.

Ziv, Jacob; Lempel, Abraham (September 1978). "Compression of Individual Sequences via Variable-Rate Coding". *IEEE Transactions on Information Theory 24* (5): 530–536. doi:10.1109/TIT.1978.1055934.

Zukowski, M., Heman, S., Nes, N., and Boncz, P. 2006. Super-scalar RAM-CPU cache compression. In the *22nd International Conference on Data Engineering (ICDE) 2006.*

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS