Journal of Information Processing Vol.23 No.2 185–191 (Mar. 2015)

[DOI: 10.2197/ipsjjip.23.185]

Regular Paper

Compressing Inverted Index Using Optimal FastPFOR

Veluchamy Glory[1,a)] Sandanam Domnic[1,b)]

Received: June 20, 2014, Accepted: November 10, 2014

Abstract: Indexing plays an important role for storing and retrieving the data in Information Retrieval System (IRS). Inverted Index is the most frequently used indexing structure in IRS. In order to reduce the size of the index and retrieve the data efficiently, compression schemes are used, because the retrieval of compressed data is faster than uncompressed data. High speed compression schemes can improve the performance of IRS. In this paper, we have studied and analyzed various compression techniques for 32-bit integer sequences. The previously proposed compression schemes achieved either better compression rates or fast decoding, hence their decompression speed (disk access + decoding) might not be better. In this paper, we propose a new compression technique, called Optimal FastPFOR, based on FastPFOR. The proposed method uses better integer representation and storage structure for compressing inverted index to improve the decompression performance. We have used TREC data collection in our experiments and the results show that the proposed code could achieve better compression and decompression compared to FastPFOR and other existing related compression techniques.

Keywords: Index Compression, Information Retrieval, Inverted File, FastPFOR

1. Introduction

Information Retrieval Systems play a very important role in information processing because the source of information is explosive in recent years. Development of an effective and efficient method of providing the information is essential in information processing. Effective processing refers to the identification of the relevant information (quality) whereas efficient processing means minimizing the amount of space and time required to process the data. Hence, the main objective of IRS is to provide maximum efficiency and effectiveness with the proper balance between them. Simultaneous achievement of highly efficient and effective information processing is quite a challenging task. Increase in the efficiency (retrieval speed) would affect the effectiveness (quality) and vice-versa.

IRS is widely used in many applications such as digital libraries, search engines, e-commerce, electronic news, genomic sequence analysis etc. [1], [2]. One of the efficient techniques used to locate the data for fast retrieval in IRS is indexing. Inverted index [3] is the most commonly used indexing structure because query evaluations are done at faster rate when compared to signature files [4], PAT tress [5] and Bitmaps [6].

An inverted index contains two main parts: a *lexicon file* (*dictionary*) and *inverted list* (*posting list*). Lexicon files list all the *distinct terms* (that appear in the document collection) and *document frequency* which means the total number of documents in which the term appears. Each inverted list contains a sequence of *document identifiers* (*id*), *term frequency* (*tf*) and *positions*. Document identifier (*id*) and term frequency (*tf*) indicate where the term occurs and how many times the particular term appeared in the document. The document identifiers are stored by increasing order in each inverted list and each document identifier is replaced by D-Gap (difference between the document identifier except the first one). Since document identifier is distinct, the d-gaps show some probability distribution. Depending on the d-gaps probability distribution, many coding methods have been proposed for compressing inverted list.

Index compression [7], [8], [9], [10], [11] can improve the performance of IRS through transferring and keeping more data in storage. In order to reduce the time-consuming transfers between Hard disk and RAM, and increase the caching capacities of the IR system, high speed compression strategies are applied on inverted index. Compression techniques are classified into two categories such as integer compression techniques and integer list compression techniques. Integer compression processes each integer individually whereas integer list compression compresses a group of integers. Some of the earliest integer compression techniques are Golomb Code (GC) [12], Rice Code (RC) [13], Elias Gamma Code (EC) [14], Elias Delta Code (EDC) [14], Variable Byte Code (VB) [15], Fast Extended Golomb code (FEGC) [16] and Re-ordered Fast Extended Golomb code (RFEGC) [17]. Golomb and Rice coding are much slower than Variable Byte code. Also, Variable Byte code is twice as fast as Elias Gamma and Elias Delta Code. Interpolative Code [18], Simple Family (Simple-9 and Simple-16) [8], Frame-Of-Reference (FOR) [19], [20], Patched coding methods (PFORDelta, NewPFD, OptPFD and FastPFOR) [21], [22], [23] are integer list compression techniques. Simple family, FOR and patched coding methods have been used for faster decoding performance. Interpolative coding is slower than Golomb coding [8], [22]. Simple family techniques compress slightly better, but are generally slower. FOR can have a competitive compression ratio, but sometimes compress poorly.

[1] Department of Computer Applications, National Institute of Technology, Tamilnadu, India
[a)] glory.nitt@gmail.com
[b)] domnic@nitt.edu

© 2015 Information Processing Society of Japan

FastPFOR (FastPFD) [23] gives competitive compression rates and fast decoding, but its decompression (both disk access + decoding speed) might not be better.

In order to achieve better performance in compression as well as in decompression, in this paper, we propose a new patched coding technique called Optimal FastPFOR which is based on FastPFOR. In Optimal FastPFOR, we have used new optimal cost formula to compute an optimal (b) for compressing the block of integers. We have also used fixed number of 128 bits for every block to store the positions of the exceptions that occur in that block to reduce the storage cost of the exceptions. We have done experiments on the three standard datasets, namely Gov2, Clueweb 09 and Yandex to evaluate the performance of the proposed method and the results are compared with the existing techniques.

The rest of the paper is organized as follows: Section 2 gives the review of some of the compression schemes. The proposed method is discussed in Section 3. In Section 4, experimental results are discussed. Finally, conclusions are derived in Section 5.

2. Compression Techniques

In this section, we summarize some of the integer compression techniques which have been used for inverted list compression.

2.1 Integer Encoders

2.1.1 Unary Code

Unary code [15] represents an integer n as $n-1$ one bits followed by single zero bit or $n-1$ zero bits followed by single one bit. Unary code is advantageous when the small range of integers are occurring more than the large range of integers.

2.1.2 Golomb and Rice Code

Golomb/Rice code is a parameterized code; it encodes an integer into two parts. The integer n is divided by the divisor d. Then, the quotient q is coded by unary code and the remainder r is coded by binary code in $\log_2 d$ bits. In Golomb code [12], the parameter d is chosen depending on the distribution of the integers and normally the parameter value is $0.69{*}avg$, where the avg is the average value of the numbers being compressed. In Rice code [13], the parameter should be the power of two for making more efficient to implementation with use of bitwise operators.

2.1.3 Elias Gamma and Elias Delta Code

Elias Gamma code [14] represents an integer n as two parts. The first part is unary representation for the length of the binary representation of the integer n i.e., unary ($|B(n)|$) and the second part is the binary representation of the integer n without its most significant bit i.e., $\sim B(n)$.

Gamma code becomes inefficient when the integers are large. In order to overcome that Elias delta code [14] was proposed which is encoding the length of the binary representation of the integer ($|B(n)|$) using the Elias gamma code. The second part is the same $\sim B(n)$. Elias Gamma and Elias Delta Codes are slower than variable byte code [23]. Hence, these methods have not been considered in the experiment.

2.1.4 Fast Extended Golomb Code and Re-ordered Fast Extended Golomb Code (RFEGC)

Fast Extended Golomb Code [16] encodes an integer n based on Extended Golomb Code (EGC) [24]. Here, the integer n is divided repeatedly M times by the divisor d until the quotient (q) becomes zero and the divisor is confined to powers of two. The remainders r_i ($i = 1$ to M) are saved in each division. M is coded by unary code and remainders are coded by binary code.

RFEGC [17] is based on the ideas used in RC and FEGC to represent the given non-negative integer n. In RFEGC, the given integer n is divided by a divisor d (2^k), a power of 2, recursively L times until the condition either $q_1 + 1 \leq (L-1)k + L$ or $q_L = 0$ is satisfied, where $q_1 = n/2^k$. Each remainder after the division is encoded using two components: flag bit and data bits. The flag bit is used to indicate whether the next remainder is still in the part of current integer or not. The data bits are the binary representation for that remainder.

2.1.5 Variable Byte Code

Variable Byte code [15] encodes the integer n into sequence of bytes. The lower-order seven bits of each byte is used to store the data part and the higher-order bit is used to check whether the next byte is part of the current integer or not. Compared to bitwise technique like Rice code, Variable Byte code requires a single branching condition for each byte which is more cost-effective in terms of CPU cycles.

2.2 Integer List Encoders

2.2.1 Simple Family

Simple-9 and Simple-16 encode the groups of integers within a single 32-bit word. Basically, in Simple-9, there are nine possible ways of encoding a list of positive integers but it wastes the bits when encoding some combination of integers. In order to overcome this issue, Simple-16 technique was proposed by Anh and Moffat [8]. Simple-16 uses the sixteen possible ways to encode the list. In Simple 9 & 16, each 32 bits is partitioned into 28-bits used for *data bits* and 4-bits used for describing the organization of the data bits. While using these schemes, they may sometimes compress slightly better, but generally it is slow [23].

2.2.2 FOR

FOR compresses the fixed size block of integers. It finds the maximum M and the minimum m value in the block and then all the values in that block are coded by using b bits each. The bit width (b) is calculated using the formula $\lceil \log_2(M+1-m) \rceil$.

2.2.3 PFOR

PFOR (or PForDelta when used in conjunction with delta coding) [21] approach is a Patched Frame-Of-Reference and it uses a reasonably small bit width (b) to represent most of the values in a block and the values in a range larger than 2^b are treated as exceptions, which are stored in a separate location. PFOR overcomes the drawback of the FOR approach where the presence of single large value increases the bit width (b) which leads to a poor compression performance.

In PFOR approach, the integers to be coded are divided into pages of fixed length 2^{16} integers. For each page, the small bit width b is calculated, but the data is encoded in blocks of 128 integers with a separate array for storing the exceptions (c). Two storage arrays (normal data & exception data sections) are used for coding the integers. One is for storing the exceptions (greater than 2^b) and another for storing the normal values

© 2015 Information Processing Society of Japan

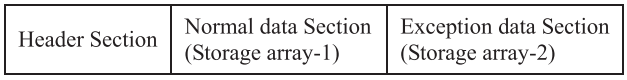

Fig. 1 PFOR storage structure.

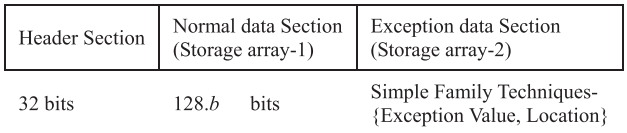

Fig. 2 New PFD/Opt PFD storage structure.

(smaller than 2^b) and the integer offsets pointing to the next exception in the block of 128 integers. These two storage arrays are preceded by a 32-bit word containing two pointers. The first pointer is pointing to the location of the first exception in the block of 128 integers and the second pointer points to the location of the first exception value in the storage array of the exceptions. Each exception value and the offset value is stored in 32 bits and b bits respectively. If the bit width b is too small to represent an offset value in the normal data section, a compulsive exception is created. The storage structure used in PFD is given in **Fig. 1**.

2.2.4 NewPFD and OptPFD

To get better compression than PFOR, Yang et al. [22] proposed two new schemes called New PFD and OptPFD. These schemes divide the data into blocks of fixed length 128 integers, and the small bit width b is calculated for each block. The main difference between NewPFD and PFOR is the way in which the exceptions are treated. NewPFD compresses the exceptions where as PFOR does not compress the exceptions. Two storage arrays are used in NewPFD for each block, one is for storing the normal values ($< 2^b$) and lower b bits of the exceptions ($\geq 2^b$); another is for storing the locations of the exceptions and the $(32-b)$ higher bits of the exceptions, which are compressed by simple family technique. These two arrays are preceded by a 32 bit word used to store the b, the number of exceptions, and the number of 32 bit words used to store the compressed exception values. The bit width (b) is selected for each block and it does not allow more than 10% of integers as exceptions. OptPFD works similarly, but it uses optimal cost formula to optimize the compression ratio and the decompression speed. The storage structure used by OptPFD and NewPFD for implementation is given in **Fig. 2**.

Illustration:

Here, the NewPFD is taken for illustration and we find the minimal bit width b illustrated with an example.

Let a set of document identifiers be:

<2, 3, 5, 43, 45, 47, 48, 49, 52, 54, 56, 88, 91, 94, 146, 148>

D-Gap values for the given set of document identifiers are:

<2, 1, 2, 38, 2, 2, 1, 1, 3, 2, 2, 32, 3, 3, 52, 2>

NewPFD determines a value b such that most of the values (90%) to be encoded are less than 2^b and thus fit into a fixed bit field of b bits for each block in the normal data section and it does not allow more than 10% of integers as exceptions. According to above condition, the bit width b = 6 is selected for the above example (D-Gaps) and hence, the number of exceptions is zero.

As per the NewPFD storage structure, we need 96 (6∗16) bits for the storage in normal data section, 32 bits in header section. So NewPFD totally uses 128 bits to compress the data.

2.2.5 FastPFOR

In an attempt to offer better decoding speed and compression rate, Lemire et al. [23] proposed FastPFOR technique which is similar in design to OptPFD and NewPFD. In FastPFOR, the small bit width b is calculated for every block of 128 integers and the $(\max b - b)$ higher bits of the exceptions are stored for every page of 2^{16} integers in one of the 32 arrays. The difference between the maximal bit width (max b) and b is used to locate the storage array (1 to 32) of the exceptions. The decoding speed is achieved in FastPFOR when encoding and decoding of the exceptions are done in bulk. The performance of the FastPFOR depends on the way in which the exceptions are determined, compressed and stored. More or less 10% of integers treated as exceptions in a block may lead to a poor compression performance. To optimize the number of exceptions (C), it uses the cost optimization formula: $8+(b\times 128)+C\times(8+\max b-b)$ to determine b for every block of 128 integers by varying the values of b and C, So that it can achieve better search performance (disk access + decoding) than NewPFD and OptPFD.

For each block of a page, FastPFOR compression technique has the header section and it contains the sequence of bytes to store the following information

- The bit width (b) – One byte
- Maximal bit width (max b) – One byte
- The Number of Exceptions (C) – One byte
- Location of the exceptions – C bytes.

The storage structure used by FastPFOR is given in **Fig. 3**.

Illustration:

The calculation of minimal bit width b is best illustrated with an example.

D-Gap values: <2, 1, 2, 38, 2, 2, 1, 1, 3, 2, 2, 32, 3, 3, 52, 2>

According to the cost optimization formula: $8+(b\times 128)+C\times(8+\max b-b)$, the optimum bit length is chosen by the following method.

In the cost formula, the value 128 indicates the number of integers in a block, since the example taken consists of 16 integers, instead of 128, the value 16 is used for calculation. In the initial iteration, b is assigned to max b value, hence C becomes zero, the formula is reduced to 128 (16) × b. According to this formula, cost value is calculated first.

For the given example (D-Gaps), Max b = 6 and b = 6.

Initially the best_cost = 16 × b = 16 × 6 = 96; then b is decremented by one in each iteration until b = 0.

(i) b is decremented by one, now b = 5, the number of exceptions is 3;
this_cost = 8 + (16 ∗ 5) + (3 ∗ (8 + 6 − 5)) = 8 + 80 + 27 = 115;
Check If (this_cost < best_cost) => (115 < 96) => false

(ii) Now b = 4, number of exceptions = 3;
this_cost = 8 + (16 ∗ 4) + (3 ∗ (8 + 6 − 4)) = 8 + 64 + 30 = 102;
So, 102 < 96 is false;

(iii) Now b = 3, number of exceptions = 3;
this_cost = 8 + (16 ∗ 3) + (3 ∗ (8 + 6 − 3)) = 8 + 48 + 33 = 89;
So, 89 < 96 is true then best_cost is 89 and assign b = 3 & No. of exceptions = 3;

© 2015 Information Processing Society of Japan

<table>
<tr><td colspan="9">Header Section</td><td colspan="3">Normal data Section (Storage array)</td><td colspan="4" rowspan="2">Exception data Section (32 storage arrays)</td></tr>
<tr><td colspan="4">Block 1</td><td>….</td><td colspan="4">Block n</td><td>Block1</td><td>…</td><td>Block n</td></tr>
<tr><td>b
8 bits</td><td>Max b
8 bits</td><td>C
8 bits</td><td>(C×8) bits</td><td></td><td>b
8 bits</td><td>Max b
8 bits</td><td>C
8 bits</td><td>(C×8) bits</td><td>Block Size * b bits</td><td></td><td>Block Size * b bits</td><td>…</td><td>(Max b – b) bits</td><td>(Max b – b) bits</td><td>..</td></tr>
</table>

Fig. 3 Fast PFOR storage structure (used per page).

(iv) Now $b = 2$, the number of exceptions = 3;
this_cost = $8 + (16 * 2) + (3 * (8 + 6 - 2)) = 8 + 32 + 36 = 76$;
So, $76 < 89$ is true then the best_cost is 76 and $b = 2$ & No. of exceptions = 3;

(v) Now $b = 1$, the number of exceptions = 13;
this_cost = $8+(16*1)+(13*(8+6-1)) = 8+16+169 = 193$;
So, $193 < 76$ is false, b is decremented by one, so $b = 0$ then stop the iterations.

Finally $b = 2$ and the number of exceptions: 3 are determined according to the cost formula.

The number of exceptions and the location of each exception are determined after selection of (b). According to FastPFOR storage structure, we need 32 (2∗16) bits for the storage in normal data section, 48 (8 + 8 + 8 + (3 ∗ 8)) bits in header section and 12 bits for exceptions, so FastPFOR totally uses 92 bits to compress the data.

3. Proposed Work: Optimal FastPFOR

In the patched schemes (PFOR, NewPFD, OptPFD and FastPFD) the integers are broken down into small blocks which are then compressed. In the original patched coding scheme (PFOR), the exceptions are uncompressed and stored as 32 bits each. On the other hand, newer alternatives, like NewPFD and OptPFD store exceptions using simple family compression techniques. Though OptPFD compresses better than PFOR, it is slower than PFOR. Hence, the preferable scheme would be the one which compresses like NewPFD with the PFOR's speed. To achieve this, FastPFOR was proposed by Lemire et al. [23]. But, when it is compared to OptPFD, compression performance is inferior because the number of exceptions is larger. If the number of exceptions is more, then the header size of the FastPFD is increased and it affects the decompression speed (disk access time + decoding time) of FastPFOR. If the number of exceptions is increased, the storage requirements for the locations of the exceptions will also be increased, which in turn will reduce the compression performance of FastPFOR. It is concluded that the number of exceptions and the representations of the locations of the exceptions is vital in enhancing the compression performance of FastPFOR. After considering this, we have proposed the new technique called Optimal FastPFOR in which new optimal cost formula is used to optimize the number of exceptions to be determined. We also use binary pattern to denote the locations of the exceptions. So that the proposed method will always require 128 bits as the storage requirements for the locations of the exceptions where as FastPFOR requires variable number of bits for storing the locations. The steps used for encoding and decoding in the proposed method are given below.

Algorithm for Encoding:

The following steps are used to encode the set of integers.

1. The given set of integers is divided into blocks of 128 integers.
2. For each block, the max b is calculated to represent the highest integer value in the block.
3. Then, recursively compute the optimal b by changing the value of b, which in turn will vary the C value for each block using the optimal cost formula given in Eq. (1) until get the minimum compression size for the block.

$$\text{Optimal cost} = 128 + (C \times (\max b - b)) + b \times 128 \quad (1)$$

where C is the number of exceptions to be determined based on 2^b.

4. Using the optimal b, the number of exceptions is predicted.
5. The normal values, which are smaller than 2^b, are represented using b bits each and lower b bits of exception values are stored in a separate storage array (Normal data section).
6. The max b and b values computed for each block are used to represent higher order bits of each exception value of the block in (max $b - b$) bits. The represented value is stored in one of 32 storage arrays (exception data section). The difference between max b and b value is used to locate the represented value in one of these arrays. These 32 arrays are maintained for every page of 2^{16} integers.
7. For each block of 128 integers, the following information is maintained as header information in the header section.
   - The optimal bit width (b) – One byte.
   - Maximal bit width (max b) – One byte.
   - Exception locations – 128 bits (binary pattern).

In the binary pattern, the position of '1' bit indicates the presence of exception in the respective positions of the block. In addition to the 128 bits binary pattern, 32 bits binary pattern is used to know whether all 32 arrays are used for storing exceptions like FastPFOR. Each bit of the 32 bits binary pattern corresponds to one array. If an array is not empty, the corresponding bit is set to 1, otherwise set to 0. The storage structure used in Optimal FastPFOR is given in **Fig. 4**.

Algorithm for Decoding:

The following steps are used to decode the compressed data.

1. For each page, all the exception arrays (1 to 32) are read from the exception data section.
2. Then, for each block in a page, read max b and b from header section and find their difference. If the difference is not zero, then there are exceptions, so decode the 128 bits binary pattern of the block to know the positions of the exceptions in the block. Otherwise, there is no exception in that block.

© 2015 Information Processing Society of Japan

3. Decode each block of 128 integers according to the corresponding bit width b of that block. The exceptions of the block are decoded after identifying their locations in the block through checking the 128 bits binary pattern of the block and merging b bits read from normal data section and (max $b-b$) bits read from exception data section of the block.

Repeat 1 to 3 until all the pages are decoded.

Illustration:

The calculation of minimal bit width b is best illustrated with an example.

D-Gap values:

<2, 1, 2, 38, 2, 2, 1, 1, 3, 2, 2, 32, 3, 3, 52, 2>

The selection of bit width b will be done by applying 3rd&4th steps of the encoding algorithm as follows:

According to the optimal cost formula:

$128 \times (1 + b) + (C \times (\max b - b))$

The optimum bit length is chosen by the following method.

In the optimal cost formula, the value 128 indicates the number of integers in a block, since the example taken consists of 16 integers, the value 128 is replaced with 16 for calculation. In the initial iteration, b is assigned to max b value, hence C becomes zero, binary pattern (128 (16) bits) does not exist and the formula is reduced to 128 (16) $\times b$. According to this formula, cost value is calculated first.

For the given example (D-Gaps), Max $b = 6$ and $b = 6$. Initially the best_cost = $16 \times b = 16 \times 6 = 96$; then b is decremented by one in each iteration until $b = 0$.

(i) b is decremented by one, now $b = 5$, the number of exceptions is 3;
this_cost = $16 * (1 + 5) + (3 * (6 - 5)) = 96 + 3 = 99$;
Check If (this_cost < best_cost) => (99 < 96) => false

(ii) Now $b = 4$, number of exceptions = 3;
this_cost = $16 * (1 + 4) + (3 * (6 - 4)) = 80 + 6 = 86$;
So, 86 < 96 is true then best_cost is 86 and assign $b = 4$ & No. of exceptions = 3;

(iii) Now $b = 3$, number of exceptions = 3;
this_cost = $16 * (1 + 3) + (3 * (6 - 3)) = 64 + 9 = 73$;
So, 73 < 86 is true then best_cost is 73 and assign $b = 3$ & No. of exceptions = 3;

(iv) Now $b = 2$, the number of exceptions = 3;
this_cost = $16 * (1 + 2) + (3 * (6 - 2)) = 48 + 12 = 60$;
So, 60 < 73 is true then the best_cost is 60 and $b = 2$ & No. of exceptions = 3;

(v) Now $b = 1$, the number of exceptions = 13;
this_cost = $16 * (1 + 1) + (13 * (6 - 1)) = 32 + 65 = 97$;
So, 97 < 60 is false, b is decremented by one. As $b = 0$ stop the iteration.

Finally $b = 2$ and the number of exceptions: 3 are determined according to the cost formula.

The number of exceptions and location of each exception are found after selection of (b). We use 32 ($2*16$) bits for the storage in normal data section, 32 (8 + 8 + 16) bits in header section and 12 bits for exceptions, So Optimal FastPFOR totally uses **76** bits to compress the data. But FastPFOR requires 92 bits including 48 ($8 + 8 + 8 + 3 * 8$) bits for header section, 32 bits for normal data section and 12 bits for exceptions, to compress the same data for the b determined according to the cost formula given in Section 2.2.5. Compared to FastPFOR, Optimal FastPFOR needs less number of bits.

4. Experimental Results

The main purpose of our experiment is to evaluate the performance of compression techniques. We have implemented the proposed code and tested on TREC document collections such as Clueweb 09, Gov2 and Yandex. The storage requirements and query processing time for each integer encoding method are measured using compression rate and search time for all the document collections. We have used Intel Xeon processor machine equipped with 16 GB of RAM and the 64-bit version of the windows 7 Operating System in our experiments. We have implemented our code in Java and used some of the source codes, which are available as open source in Ref. [25].

In our experiments, we have applied the integer encoding methods to compress the document identifiers of the inverted lists, which are constructed [26], [27] from TREC Web collections (Clueweb 09, Gov2, Yandex). The GOV2 is a crawl of the .gov sites, which contains 25 million HTML, text, and PDF documents. The Clueweb09 collection is a more realistic HTML collection of about 50 million crawled HTML documents, mostly in English. The formula used to calculate compression rate (bits per integer) in our experiment is:

$$\text{Compression (or Bit) Rate} = \frac{\text{Compressed Size of Inverted List (docids)}}{\text{Total number of Document Identifiers in the List}} \quad (2)$$

Table 1 Bit rate (bits per docid) for TREC data collections.

Compression Techniques	Gov2	Clueweb	Yandex
Optimal FastPFOR	**4.436**	**6.285**	**5.259**
FastPFOR	4.661	6.871	5.460
New PFD	4.682	7.055	5.860
Opt PFD	4.509	6.708	5.384
RFEGC	4.957	6.444	5.799
FEGC	4.957	6.444	5.799
Rice	14.07	13.66	13.748
VBC	8.631	9.23	9.23

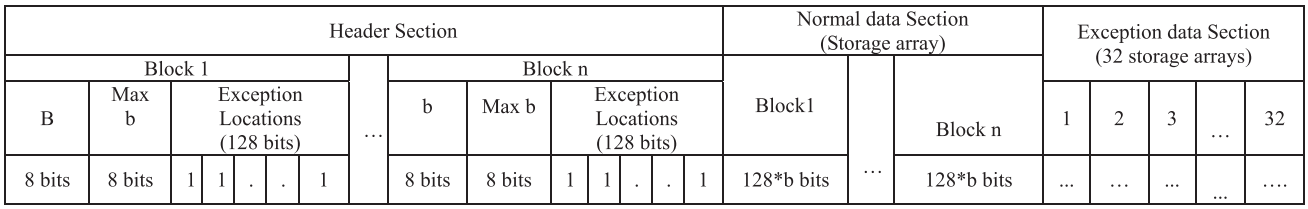

Fig. 4 Optimal FastPFOR storage structure.

© 2015 Information Processing Society of Japan

Table 2 Search performance time (in *milliseconds*) for TREC data collections.

Compression Techniques	100000 docid's			10000 docid's		
	Gov2	Clueweb	Yandex	Gov2	Clueweb	Yandex
Optimal Fast PFOR						
Disk Access Time (AT)	11762	7379	5429	218	281	312
Decoding Time (DT)	78	46	31	15	15	15
Search Time (ST)	**11840**	**7425**	**5460**	**233**	**296**	**327**
Fast PFOR						
Disk Access Time (AT)	12527	7425	5647	218	281	312
Decoding Time (DT)	63	31	47	16	15	16
Search Time (ST)	12590	7456	5694	234	296	328
New PFD						
Disk Access Time(AT)	14305	8159	7504	234	296	562
Decoding Time (DT)	63	31	47	15	16	15
Search Time (ST)	14368	8190	7551	249	312	577
Opt PFD						
Disk Access Time(AT)	12730	7612	5944	234	296	343
Decoding Time (DT)	62	62	47	15	16	16
Search Time (ST)	12792	7674	5991	249	312	359
RFEGC						
Disk Access Time(AT)	12074	8253	8767	343	343	405
Decoding Time (DT)	125	93	78	31	15	31
Search Time (ST)	12199	8346	8845	374	358	436
FEGC						
Disk Access Time(AT)	12074	8253	8767	343	343	405
Decoding Time (DT)	202	140	156	32	16	31
Search Time (ST)	12276	8393	8923	375	359	436
Rice						
Disk Access Time(AT)	38501	29921	30560	3213	2964	3291
Decoding Time (DT)	140	125	125	31	31	31
Search Time (ST)	38641	30046	30685	3244	2995	3322
VBC						
Disk Access Time(AT)	23681	21653	22511	1997	1997	2106
Decoding Time (DT)	78	78	78	15	31	15
Search Time (ST)	23759	21731	22589	2012	2028	2121

Table 1 shows the bit rates achieved by optimal FastPFOR and other encoding methods. From the results, Optimal FastPFOR achieves 4.8%, 8.5% and 3.7% gain in compression compared to the recent FastPFOR technique for Gov2, Clueweb09 and Yandex collections, respectively. Our code also gives better results compared to other existing methods.

In addition to the compression performance measurement, the search performance of each coding method is measured and the results are tabulated in **Table 2**. The search performance (or query processing time/decompression performance) is measured using the Eq. (3).

$$\text{Search Time (ST)} = \text{Disk Access Time (AT)} + \text{Decoding Time (DT)} \quad (3)$$

In order to measure the search performance, a set of random queries is generated for each collection. We used 100 random queries for each data set to evaluate the search performance. For each query, approximately 100,000 docids are retrieved and decoded. Then, the search performance is measured by adding the time taken to retrieve the data from the disk and the time taken to decode the data.

The search performance measurements are shown in Table 2. From the results, for retrieving the large number of docids, compared to all other methods, Optimal FastPFOR has given the better result for query processing time because it has the lower bit rate, so it takes lesser time for accessing data from the disk. For small number of docids, optimal FastPFOR gives competitive results. As a conclusion, it can give better or equivalent query processing time depending on the size of the data to be retrieved. The good compression technique must be characterized by minimum bit rate and minimum decompressing time. So, among other techniques, optimal FastPFOR is the better compression technique for IR applications.

5. Conclusion

In this paper, we have proposed a new novel patched coding, Optimal FastPFOR which is based on FastPFOR. FastPFOR technique is a recent fast decoding technique, but compression performance is not as high compared to OptPFD technique. OptPFD technique gives the better compression rate, but it does not decompress fast. Our technique achieves better compression performance and decompression performance compared to FastPFOR, OptPFD and other techniques. Through achieving good compression, we can process the query and retrieve the data quickly. From the experimental results it is observed that our code yields better performance in both fast querying and space efficient indexing.

© 2015 Information Processing Society of Japan

References

[1] Kobayashi, M. and Takeda, K.: Information retrieval on the web, *ACM Computing Surveys*, Vol.2, No.2, pp.144–173 (2000).

[2] Williams, H.E. and Zobel, J.: Indexing and retrieval for genomic databases, *IEEE Trans. Knowledge and Data Engineering*, Vol.14, No.1, pp.63–78 (2002).

[3] Zobel, J., Moffat, A. and Ramamohanarao, K.: Inverted files versus signature files for text indexing, *ACM Trans. Database Systems*, Vol.23, No.4, pp.453–490 (1998).

[4] Faloutsos, C.: Access methods for text, *ACM Computing Surveys*, Vol.17, No.1, pp.49–74 (1985).

[5] Morrison, D.R.: PATRICIA – Practical algorithm to retrieve information coded in alphanumeric, *Journal of Association for Computing Machinary*, Vol.15, No.4, pp.514–534 (1968).

[6] Chan, C.-Y. and Ioannidis, Y.E.: Bitmap index design and evaluation, *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pp.355–366 (1998).

[7] Scholer, F., Williams, H.E., Yiannis, J. and Zobel, J.: Compression of inverted indexes for fast query evaluation, *Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Tampere, Finland, pp.222–229 (2002).

[8] Anh, V.N. and Moffat, A.: Inverted index compression using word-aligned binary codes, *Information Retrieval*, Vol.8, No.1, pp.151–166 (2005).

[9] Zobel, J. and Moffat, A.: Inverted files for text search engines, *ACM Computing Surveys*, Vol.38, No.2, pp.1–56 (2006).

[10] Trotman, A.: Compressing inverted files, *Information Retrieval*, Vol.6, No.1, pp.5–19 (2003).

[11] Zhang, J., Long, X. and Suel, T.: Performance of compressed inverted list caching in search engines, *Proc. 17th International Conference on World Wide Web*, WWW '08, ACM: New York, NY, USA, pp.387–396 (2008).

[12] Golomb, S.W.: Run length encoding, *IEEE Trans. Inf. Theory*, Vol.12, No.3, pp.399–401 (1966).

[13] Rice, R.F.: *Some practical universal noiseless coding techniques* (*Tech. Rep.*), JPL Publication Pasadena, CA: Jet Propulsion Laboratory (1979).

[14] Elias, P.: Universal codeword sets and representations of the integers, *IEEE Trans. Inf. Theory*, Vol.21, No.2, pp.194–203 (1975).

[15] Salomon, D.: *Variable-length codes for data compression*, Springer-Verlag (2007).

[16] Domnic, S. and Glory, V.: Inverted file compression using EGC and FEGC, *Proc. International Conference on Communication, Computing and Security*, pp.735–742 (2012).

[17] Glory, V. and Domnic, S.: Re-Ordered FEGC and Block Based FEGC for Inverted File Compression, *International Journal of Information Retrieval Research*, Vol.3, No.1, pp.71–88 (2013).

[18] Moffat, A. and Stuiver, L.: Binary interpolative coding for effective index compression, *Information Retrieval*, Vol.3, No.1, pp.25–47 (2000).

[19] Goldstein, J., Ramakrishnan, R. and Shaft, U.: Compressing relations and indexes, *Proc. 14th International Conference on Data Engineering, ICDE '98, IEEE Computer Society*: Washington, DC, USA, pp.370–379 (1998).

[20] Ng, W.K. and Ravishankar, C.V.: Block-oriented compression techniques for large statistical databases, *IEEE Trans. Knowledge and Data Engineering*, Vol.9, No.2, pp.314–328 (1997).

[21] Zukowski, M., Heman, S., Nes, N. and Boncz, P.: Super-scalar RAM-CPU cache compression, *Proc. 22nd International Conference on Data Engineering, ICDE '06, IEEE Computer Society*: Washington, DC, USA, pp.59–71 (2006).

[22] Yan, H., Ding, S. and Suel, T.: Inverted index compression and query processing with optimized document ordering, *Proc. 18th International Conference on World Wide Web, WWW '09*, ACM: New York, NY, USA, pp.401–410 (2009).

[23] Lemire, D. and Boystov, L.: Decoding billions of integers per second through vectorization, *Software: Practice and Experience* (2013).

[24] Somasundaram, K. and Domnic, S.: Extended golomb code for integer representation, *IEEE Trans. Multimedia*, Vol.9, No.2, pp.239–246 (2007).

[25] Lemire, D. and Boystov, L.: FastPFOR Java code (2013), available from ⟨https://github.com/lemire/JavaFastPFOR⟩.

[26] Boystov, L.: Clueweb09 posting list data set (2012), available from ⟨http://boytsov.info/datasets/clueweb09gap/⟩.

[27] Silvestri, F. and Venturini, R.: Gov2 & Yandex Dataset, available from ⟨http://integerencoding.isti.cnr.it/⟩.

V. Glory received her B.Sc. degree in Computer Science from Madurai Kamaraj University and M.C.A degree from Kalasalingam University, India in 2007 and 2010, respectively. Currently she is a pursuing Ph.D. in Department of Computer Applications at National Institute of Technology, Tiruchirappalli, India. Her research Interests is in Information Retrieval.

S. Domnic received his B.Sc. degree in physics and M.C.A degree from Bharathidasan University, India, in 1998 and 2001, respectively, and the Ph.D. degree from Gandhigram Rural University, Gandhigram, India in 2008. He is presently working as an assistant professor in the Department of Computer Applications, National Institute of Technology, Tiruchirappalli, India. His current research interests are in Data Compression, Image Compression and Information Retrieval.

© 2015 Information Processing Society of Japan