A Postprint of an Inverted Index Compression Method

Authors: Bai Fujun, Gao Jianling, Li Wanrong, Siyun He, Xiao Shaowu

Date: 2018-05-20T00:00:00+00:00

Abstract

Efficient access to inverted indexes is critical for search engines to rapidly respond to user queries, and compressing inverted lists represents one of the most important techniques for enhancing search engine performance. This paper investigates the Adaptive Segmentation Compression Scheme (ASCS) algorithm. To address the suboptimality of the uniform segmentation method employed in ASCS, we propose optimizing the segmentation strategy within ASCS using the Artificial Bee Colony algorithm. To remedy the issue that ASCS considers overly simplistic factors influencing sequence space occupancy, we propose an improved algorithm incorporating multiple factors. For long sequences with non-uniform distribution that yield unsatisfactory compression rates under ASCS, we propose performing sorting and differential encoding operations prior to compression with the ASCS algorithm. Comparative experiments demonstrate that the optimized algorithm can significantly compress inverted indexes.

Full Text

Preamble

Method of Inverted Index Compression

Bai Fujun[1], Gao Jianling[1], Li Wanrong[2], He Siyun[1], Xiao Shaowu[1]
([1]College of Big Data & Information Engineering, [2]Archives, Guizhou University, Guiyang 550025, China)

Abstract: Efficient access to inverted indexes is crucial for search engines to achieve fast query response times, and compressing posting lists represents one of the most important methods for improving search engine performance. This paper investigates the Adaptive Segmentation Compression Scheme (ASCS) algorithm and proposes three key improvements: First, to address the suboptimal uniform segmentation approach in ASCS, we optimize the segmentation method using the Artificial Bee Colony (ABC) algorithm. Second, to remedy ASCS' s

overly simplistic consideration of factors affecting sequence space consumption, we propose an improved algorithm under multiple influencing factors. Third, to improve compression ratios for long sequences with uneven distributions under ASCS, we propose performing sorting and differential encoding operations before applying ASCS compression. Comparative experiments demonstrate that the optimized algorithm significantly improves inverted index compression.

Keywords: search engine; inverted index; index compression; artificial bee colony algorithm; ASCS algorithm

0 Introduction

With the rapid development of the Internet, particularly mobile Internet, network information volume has grown increasingly massive, posing significant challenges for information storage and retrieval. Efficiently locating and searching for target information within vast information collections has become an urgent problem to solve. The inverted index [1] serves as the modern search engine' s indexing model, where each term corresponds to a posting list containing document IDs (DocIDs), frequencies, and positions where the term appears. An inverted index file is essentially a collection of terms and their posting lists, with the structure shown in Table 1 . Compressing an inverted index means compressing both terms and posting lists. For better storage and compression, the DocIDs, frequencies, and positions for each term are typically stored separately. Thus, compressing posting lists essentially involves compressing integer sequences composed of DocIDs, frequencies, and positions.

For each DocID, its disk space consumption depends on the maximum document ID in the collection, which obviously wastes substantial storage resources. Inverted index compression technology is currently widely applied, with numerous algorithms and encoding schemes adopted in search engines. A common practice involves storing d-gaps [2] (differential values) of DocIDs and positions rather than their actual values to reduce storage space. Current compression algorithms can be categorized based on their encoding properties [3] into bitwise compression, byte-aligned compression, word-aligned compression, oblivious compression, and list-adaptive compression. Oblivious compression encodes values based solely on themselves, considering only individual information while ignoring neighboring and global context.

Unary [4] encodes a positive integer n as n-1 ones followed by a zero (e.g., 5 is encoded as 11110), making it suitable for small integers but inefficient for large ones. Elias Gamma [5] encodes integer n into two parts: a length portion and a data portion. The length portion uses Unary encoding of n' s binary length, while the data portion consists of n' s binary representation without the most significant bit. For example, 6 is encoded as 110|10. Similarly, Gamma coding is unsuitable for large integers. Elias Delta [6] also encodes integer n into length and data portions, but the length portion uses Elias Gamma encoding of n' s

binary length, with the data portion identical to Elias Gamma' s. For instance, 6 is encoded as 1101|10, which is more efficient for large integers.

Golomb [7] coding divides integer n by divisor d to obtain quotient q and remainder r. For compression sequences, the divisor is typically set as $d \approx 0.69 \times \text{avg}$, where avg is the sequence average. Rice [8] coding is an upgraded version of Golomb coding that restricts the divisor to powers of two, enabling efficient shift operations instead of division and faster encoding. Extended Golomb [9] coding repeatedly divides integer n by divisor d k times until quotient q=0. Subsequent improvements include Fast Extended Golomb [10] (FEGC), Re-Ordered FEGC [11] (RFEGC), Block RFEGC [11] (BRFEGC), Re-Ordered Fast Modified Extended Golomb [12] (RFMEGC), Run-Length Encoding RFEGC [12] (RLETFEGC), and Dynamic Programming RFEGC [12] (DPRFEGC).

PackedBinary [13] is a position-based encoding that processes one integer at a time, selecting the effective bit width w of the maximum integer in the sequence and encoding all integers using w bits. Variable Byte [14] is a byte-aligned method that uses the lower 7 bits of each byte to represent integer n' s binary portion, with the most significant bit indicating whether encoding is complete (1) or continues (0). For example, 201 is encoded as 10000001 01001001. The Simple [15] family compresses integer sequences by packing multiple integers into a single word. Frame of Reference [16] (FOR) encodes blocks of integers (e.g., 128) by storing the minimum value m in binary and other integers as differences from m, with each integer occupying $b = \lfloor \log_2(M - m) \rfloor + 1$ bits.

Patched Frame of Reference [17] (PFOR) addresses FOR' s poor compression when large outliers M cause excessive b values. PFOR selects parameter b for most integers (e.g., 90%), treating integers requiring more than b bits as exceptions. It uses FOR for normal values and stores the position of the next exception in the b bits originally allocated for exceptions, storing actual exception values uncompressed after the FOR output. When b bits cannot represent exception intervals, b is increased, reducing compression ratio. To enhance PFOR, NewPFD [18], OptPFD [18], and FastPFOR [19] were subsequently proposed.

To improve inverted index compression ratios, this paper investigates the Adaptive Segmentation Compression Scheme (ASCS) algorithm [20]. ASCS has three main deficiencies: (a) its uniform segmentation approach is suboptimal, resulting in poor compression; (b) it considers only a single factor affecting sequence space consumption, limiting effective data compression; and (c) it achieves unsatisfactory compression ratios for long sequences with uneven distributions. To address these limitations, we propose: optimizing ASCS segmentation using the Artificial Bee Colony algorithm, developing a comprehensive improvement under multiple factors, and applying sorting and differential encoding before ASCS compression. Experiments demonstrate that the improved algorithm significantly enhances inverted index compression.

---

1 Related Algorithms

1.1 CSN Encoding

For a positive integer sequence $X = \{x_1, x_2, \dots, x_n\}$, a unique Gödel number [21] can represent it. Gödel numbers are based on prime factorization encoding systems with the formula:

$$\mathrm{enc}(x_1, x_2, \dots, x_n) = p_1^{x_1} \cdot p_2^{x_2} \cdot \dots \cdot p_n^{x_n}$$

where p_i are consecutive increasing primes. This encoding grows exponentially with integer sequence values and length, making it unsuitable for long integer sequence compression. Based on Gödel numbers, the CSN-1 number [20] encoding is defined as:

Definition 1. For an integer sequence $X = \{x_1, x_2, \dots, x_n\}$ of length n with minimum element 1, the CSN number encoding can be obtained through the recurrence relation (2):

$$\begin{cases} a_0 = 1 \\ a_k = T \cdot a_{k-1} + x_k, \quad 1 \le k \le n \\ \mathrm{CSN} = a_n \end{cases}$$

where $T = \max(x_1, x_2, \dots, x_n) + 1$.

This compresses an integer sequence into a single CSN number. However, CSN-1 becomes inapplicable when $x_i \le 0$ or the minimum element is not 1. Therefore, CSN-2 improves upon CSN-1:

Definition 2. For an integer sequence $X = \{x_1, x_2, \dots, x_n\}$ of length n, the CSN number encoding can be obtained through recurrence relation (3):

$$\begin{cases} a_0 = 1 \\ a_k = T \cdot a_{k-1} + x_k - m + 1, \quad 1 \le k \le n \\ \mathrm{CSN} = a_n \end{cases}$$

where $m = \min(x_1, x_2, \dots, x_n)$ and $T = \max(x_1, x_2, \dots, x_n) - m + 2$.

This enhances CSN encoding' s adaptability. For CSN decoding, which is the inverse process, the recurrence relation (4) is:

$$\begin{cases} a_{k-1} = \left\lfloor \frac{a_k - 1}{T} \right\rfloor \\ x_k = \mathrm{rem}\left(\frac{a_k - 1}{T}\right) + m - 1 \end{cases}$$

where rem denotes remainder operation and $\lfloor \cdot \rfloor$ denotes floor division. When $a_k = 0$, the recursion ends, and reversing the sequence X restores the original encoded sequence.

1.2 Adaptive Segmentation ASCS Algorithm

Decoding a CSN encoding requires two parameters: T and m. Therefore, compressing a length-n integer sequence requires storing three parameters: CSN, T, and m. The space calculation for CSN number encoding uses the general formula (5):

$$\text{CSN} = a_n = T^n \cdot a_0 + T^{n-1} \cdot (x_1 - m + 1) + T^{n-2} \cdot (x_2 - m + 1) + \ldots + T \cdot (x_{n-1} - m + 1) + (x_n - m + 1)$$

Thus, the space needed to store one integer sequence is:

$$S = \log_2(\text{CSN}) + \sigma(T) + \sigma(m)$$

where σ represents the space occupied by storing one integer.

From equation (6), factors affecting an integer sequence' s space consumption include not only the sequence' s values and length but also value distribution and min/max values. Parameter T has a power function relationship with space S, while parameter m has a linear relationship. Sequences with concentrated values achieve better compression.

Although segmentation compression ASCS can reduce T values and sequence lengths for small segments, it may increase each segment' s minimum value m, yielding unsatisfactory compression for long integer sequences. From equation (7), reducing segment parameters T, m, and sequence values x_i can effectively improve ASCS compression ratios.

The adaptive segmentation ASCS algorithm aims to minimize the average T value of the long sequence being compressed. The algorithm proceeds as follows:

a) Initialize segment count $b = 1$, optimal segment count $p = 1$ (no segmentation), and calculate the T value.

b) Calculate each segment' s T value and the long sequence' s average T value t. Continue if $b < n - 1$ and $t < T$, otherwise terminate.

c) Increase segment count: $b = b + 1$ and return to step b) for iterative computation.

After these steps, the segment count minimizing the long sequence' s average T value can be found, achieving better compression. The space after segmentation compression is the sum of each segment' s space:

$$S = \sum_{j=1}^{p} \left(\log_2 \left(\sum_{i=1}^{n_j} T_j^{n_j - i} \cdot x_{j,i} \right) + \sigma(T_j) + \sigma(m_j) \right)$$

where p is the segment count, T_j is parameter T for segment j, n_j is the number of integers in segment j, $x_{j,i}$ is the i-th element in segment j, and m_j is parameter m for segment j.

---

2 Optimization and Improvement of ASCS Algorithm

2.1 Reordering and Differential Encoding of Sequences

In inverted indexes, DocIDs typically appear in ascending order without repetition in posting lists. Leveraging this characteristic, we first reorder the sequence to be compressed in ascending order, then perform differential encoding by subtracting each element from its successor. These two operations effectively reduce sequence values, decreasing segment parameters m and x_i under the same segmentation approach, thereby improving encoding effectiveness.

For example, sequence $X = \{2, 4, 5, 8, 12, 13, 20, 22, 25\}$ becomes $X' = \{2, 2, 1, 3, 4, 1, 7, 2, 3\}$ after differential encoding. The original sequence can be segmented as $X = \{2, 4, 5\}|\{8, 12, 13\}|\{20, 22, 25\}$ with parameters $(m_1, T_1) = (2, 5)$, $(m_2, T_2) = (8, 7)$, $(m_3, T_3) = (20, 7)$. The differential sequence segments as $X' = \{2, 2, 1\}|\{3, 4, 1\}|\{7, 2, 3\}$ with improved parameters $(m'_1, T'_1) = (1, 3)$, $(m'_2, T'_2) = (1, 4)$, $(m'_3, T'_3) = (2, 6)$. The new sequence significantly reduces values and improves segment parameters m and T, making this improvement essential.

2.2 Artificial Bee Colony Algorithm for Optimizing Sequence Segmentation

When T values are too large, compression becomes inefficient. The ASCS algorithm segments long sequences to reduce T values and lengths, achieving better compression per segment. However, ASCS uses uniform segmentation, which is not necessarily optimal. This paper employs the Artificial Bee Colony (ABC) algorithm [22] to optimize segmentation for a given segment count, using the long sequence' s average T value as the objective function.

The ABC optimization process is as follows:

a) Initialize NP food sources $X = \{X_1, X_2, \ldots, X_{NP}\}$, where each source corresponds to a feasible solution (a segmentation pattern represented by segmentation coordinates). Each source is a d-dimensional vector $X_i = \{x_{i1}, x_{i2}, \ldots, x_{id}\}$ where $d = p - 1$ (segment count minus 1). Food sources are initialized using equation (8):

$$x_{ij} = \text{randInt}(1, n-1), \quad x_{ik} = \text{randInt}(1, n-1), \quad j, k \in \{1, 2, \ldots, d\}, j \neq k, x_{ij} \neq x_{ik}$$

Each dimension is a random integer between 1 and $n-1$, with all dimensions distinct.

b) Employed bees calculate their source' s fitness and search for new sources using equation (10):

$$V_{ij} = X_{ij} + \text{randInt}(-a, a)$$

where a is a range control parameter. If a better source (higher fitness) is found, the bee moves; otherwise, the source remains unchanged and its non-update counter increments.

c) Onlooker bees probabilistically select sources using roulette wheel selection based on equation (11):

$$P_i = \frac{\text{fit}_i}{\sum_{i=1}^{NP} \text{fit}_i}$$

They update sources using the same method as employed bees. Unupdated sources have their counters incremented.

d) When a source' s non-update count reaches threshold limit, the employed bee becomes a scout, abandoning the local optimum and generating a new random source using equation (8), resetting its counter.

e) Save the current global best solution and check termination. If iterations reach maxCycle, output the optimal segmentation; otherwise, return to step b).

2.3 Improvement to ASCS Algorithm

ASCS focuses on minimizing the average T value but ignores other influential parameters m, x, and p from equation (6), leaving room for compression improvement. Combined with ABC optimization, we propose a comprehensive improvement that considers all factors affecting space consumption, using equation (7) as the objective function. This modifies the fitness calculation from equation (9) to equation (12):

$$\text{fit}_i = \frac{1}{\frac{1}{p}\sum_{j=1}^{p}\left(\log_2\left(\sum_{i=1}^{n_j} T_{j,i}^{n_j-i} \cdot x_{j,i}\right) + \sigma(T_j) + \sigma(m_j)\right)}$$

During each ASCS iteration with fixed segment count p, each source' s parameters T, m, and segment values are fixed, enabling fitness calculation via equation (12) to achieve efficient long integer sequence compression.

2.4 Program Flowchart of Improved ASCS Algorithm

The pseudocode for the improved ASCS segmentation compression algorithm is as follows:

Input: Integer sequence to be compressed
Output: CSN sequence and related parameters T, m

The flowchart is shown in Figure 1 [Figure 1: see original paper].

---

3 Experiments and Results Analysis

This experiment comprehensively compares compression results across different storage formats: integer storage (Int), string storage (String), CSN encoding, differential CSN (dCSN), CSCA encoding, differential CSCA (dCSCA), ABC-optimized CSCA (CSCAABC), differential CSCAABC (dCSCAABC), and comprehensive improvement combining all three approaches (dCSCAABCP). Compression ratio is defined as:

$$\text{rate} = \frac{S_{\text{before}} - S_{\text{after}}}{S_{\text{before}}} \times 100\%$$

where S represents sequence space in bits.

We selected nine integer sequences of varying lengths as experimental data, with detailed descriptions in Table 2 . The ABC algorithm parameters were set as: food source range control parameter $a = 2$, limit $= 5$, and maxCycle $= 250$. Table 3 and Figure 2 [Figure 2: see original paper] show each sequence' s space consumption and compression ratios relative to integer storage.

Table 2 shows that space consumption varies significantly across encoding methods. Uncompressed string storage occupies much more space than integer storage, making integer format the practical choice for uncompressed sequences.

From Table 3, the comprehensive improvement (dCSCAABCP) achieves the best compression ratio across all sequences. Figure 2 demonstrates that different sequences exhibit varying compression rates under the same encoding, depending on sequence characteristics. For the same sequence, improved versions (dCSN, CSCA) outperform basic CSN, and further improvements (dCSCA, CSCAABC) outperform CSCA. The comprehensive dCSCAABCP achieves the highest compression ratio.

---

4 Conclusion

This paper preprocesses data through sorting and differential encoding to reduce values and increase value concentration, optimizes CSCA segmentation using the Artificial Bee Colony algorithm, and employs equation (6) as the segmentation criterion for holistic sequence compression. Experiments show that the improved dCSCAP algorithm achieves excellent compression ratios for integer sequences.

While effective, dCSCAP' s iterative optimization makes it less suitable for real-time indexing and retrieval scenarios. However, it shows high applicability for periodically updated index search engines. For massive datasets, distributed computing frameworks could enable parallel encoding to improve speed. For decoding, dCSCAP only adds a differential step requiring one additional subtraction compared to CSCA, minimally impacting decoding performance. In summary, the proposed improvements hold practical significance.

References

[1] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. An Introduction to Information Retrieval [M]. Cambridge: Cambridge University Press, 2009: 67-68.

[2] Hersh W. Managing gigabytes—compressing and indexing documents and images (2nd Edition) [J]. Information Retrieval, 2001, 4 (1): 79-80.

[3] Ounis I, Amati G, Plachouras V, et al. Terrier: a high performance and scalable information retrieval platform [C]// Proc of the Osir Workshop. World Wide Web. 2009: 401-410.

[4] Elias P. Universal codeword sets and representations of the integers [M]. [S.l.]: IEEE Press, 1975.

[5] Manber U, Myers G. Suffix arrays: a new method for on-line string searches [J]. Siam Journal on Computing, 1990, 22 (5): 319-327.

[6] Golomb S W. Run-length encodings [J]. IEEE Trans on Information Theory, 1966, 12 (3): 399-401.

[7] Rice R, Plaunt J. Adaptive variable-length coding for efficient compression of spacecraft television data [J]. IEEE Trans on Communication Technology, 2003, 19 (6): 889-897.

[8] Somasundaram K, Domnic S. Extended GOLOMB code for integer representation [J]. IEEE Trans on Multimedia, 2007, 9 (2): 239-246.

[9] Domnic S, Glory V. Inverted file compression using EGC and FEGC [J]. Procedia Technology, 2012, 6 (4): 493-500.

[10] Domnic S, Glory V. Re-ordered FEGC and block based FEGC for inverted file compression [J]. International Journal of Information Retrieval Research,

2013, 3 (1): 71-88.

[11] 毛福林. 倒排索引压缩算法研究 [D]. 北京: 北京交通大学, 2015.

[12] 闫宏飞, 张旭东, 单栋栋, 等. 基于指令级并行的倒排索引压缩算法 [J]. 计算机研究与发展, 2015, 52 (5): 995-1004.

[13] Williams H E, Zobel J. Compressing integers for fast file access [J]. Computer Journal, 1999, 42 (3): 193-201.

[14] Anh V N, Moffat A. Inverted index compression using word-aligned binary codes [J]. Information Retrieval, 2005, 8 (1): 151-166.

[15] Goldstein J, Ramakrishnan R, Shaft U. Compressing relations and indexes [C]// Proc of International Conference on Data Engineering, 1998: 370-379.

[16] Zukowski M, Heman S, Nes N, et al. Super-scalar RAM-CPU cache compression [C]// Proc of International Conference on Data Engineering. Washington DC: IEEE Computer Society, 2006: 59.

[17] Yan H, Ding S, Suel T. Inverted index compression and query processing with optimized document ordering [C]// Proc of International Conference on the World Wide Web. 2009: 401-410.

[18] Lemire D, Boytsov L. Decoding billions of integers per second through vectorization [M]. Hoboken: Wiley, 2012.

[19] 特日跟, 江晟, 李雄飞, 等. 基于整数数据的文档压缩编码方案 [J]. 吉林大学学报, 2016, 46 (1): 228-234.

[20] Gödel K. Uber Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter System, I [J]. Monatshefte Für Mathematik, 1931, 38 (1): 173-198.

[21] 秦全德, 程适, 李丽, 等. 人工蜂群算法研究综述 [J]. 智能系统学报, 2014, 9 (2): 127-135.

Note: Figure translations are in progress. See original paper for figures.