On Optimizing Partitioning Strategies for Faster Inverted Index Compression

Xingshen Song(✉), Kun Jiang, Yu Jiang, and Yuexiang Yang

College of Computer, National University of Defense Technology, Changsha, China
{songxingshen,jiangkun,jiangyu14,yyx}@nudt.edu.cn

Abstract. Inverted index is a key component for search engine to manage billions of documents and fast respond to users' queries. While substantial effort has been made to compromise space occupancy and decoding speed, what has been overlooked is the encoding speed when constructing the index. VSEncoding is a powerful encoder that works by optimally partitioning a list of integers into blocks which are efficiently compressed by using simple encoders, however, these partitions are found by using a dynamic programming approach which is obviously inefficient. In this paper, we introduce compression speed as one criterion to evaluate compression techniques, and thoroughly analyze performances of different partitioning strategies. A linear-time optimization is also proposed, to enhance VSEncoding with faster compression speed and more flexibility to partition an index. Experiments show that our method offers a far more better compression speed, while retaining an excellent space occupancy and decompression speed.

Keywords: Inverted index · Index compression · Optimal partition · Approximation algorithm

1 Introduction

Due to its simplicity and flexibility, inverted index gains much popularity among modern IR systems since 1950s. Especially in large scale search engines, inverted index is now adopted as their core component to maintain billions of documents and respond to enormous queries. In its most basic and popular form, an inverted index is a collection of sorted sequences of integers [10,16,18]. Growing size of data and stringent query processing efficiency requirement have appealed a large amount of research, with the aim to compress the space occupancy of the index and speed up the query processing.

While state-of-the-art encoders do obtain very good space-time trade-offs, we argue that one important evaluation criterion has been neglected is the compression speed of these methods [10,13,17]. The reason can be attributed to the fact that index is always preprocessed offline before deployment, and once being taken into effect, update can be committed in an asynchronous and parallel manner. Therefore index designers usually traverse the sequences more than

O. Gervasi et al. (Eds.): ICCSA 2016, Part IV, LNCS 9789, pp. 246–260, 2016.
DOI: 10.1007/978-3-319-42089-9_18

once to find the optimal parameters for space-efficiency gains, making index construction speed rather slow. However, timely update for unexpected queries is becoming more and more stringent in search engine, especially in twitter and other social network sites. Compression speed should also be an important factor to evaluate an index compression algorithm. Early techniques like Simple-9 and Simple-16 [2,3] evaluate all the possible schemes to decide the best partition, PFOR [9,17,18] splits each sequence into blocks of fixed size (say, 128 integers) and goes over the whole block to decide the exception ratio and block width. In a nutshell, compression speed is compromised to achieve better space occupancy and decoding speed.

Modern encoders are designed to compress lists of integers, that is the input posting list is split into blocks with fixed or variable lengths to be encoded. Intuitively, partitioning posting list aligning to its clustered distribution, can effectively minimize the compressed size while keeping partitions separately accessed. Works from literature [1,6,13] give another perspective on index compression. The integer sequence is considered a particular directed acyclic graph (DAG), the partitioning problem is then treated as optimal path finding problem, unfortunately the DAG is complete with $\frac{n(n+1)}{2} = \Theta\left(n^2\right)$ edges, a trivial traversal may not suffice to obtain an efficient solution for this problem. Scheme from [1] uses a *greedy* mechanism to yield a sub-optimal partition with a small amount of effectiveness exchanged for speed of compression and ease of implementation. AFOR from [6] computes the block partition by using a series of fixed-sized sliding windows over a block and determines the optimal configuration of frames and frame lengths of the current window. VSEncoding from [13] finds the optimal partition by using a *dynamic programming* approach and is said to be able to encode groups of integers beating the entropy of the gaps distribution. However, dynamic programming can be very inefficient since it needs to recalculate all the edges when a new vertex is added in the current graph, to mitigate this problem VSEncoding simply restricts the length of the longest block to h, reducing its time complexity from $O(n^2)$ to $O(nh)$, but barely satisfactory in practice.

Recently, Ottaviano overcomes this drawback by introducing a new compression scheme called Partitioned Elias-Fano Index (PEF) [12], in which a linear-time approximation algorithm is presented to find a solution at most $(1+\epsilon)$ times larger than the optimal one, for any given $\epsilon \in (0,1)$. Its core idea is to generate a pruned graph $\mathcal{G}_\epsilon$ in linear time directly without explicitly constructing the whole graph $\mathcal{G}$ which, otherwise, would require quadratic time. Edges with a heavy weight are dropped according to a predefined pruning policy, reducing the time and space complexities to linear at the cost of finding slightly suboptimal partitions. The same idea is also adopted in [7].

In this paper we follow the compression techniques studied in [4,9,15,17]. However, while previous work has focused on improving compression ratio and speed up decoding, we consider compression speed as one criterion. In particular, we extensively study various compression schemes on their space-time trade-offs, and propose our optimization on VSEncoding to achieve a faster compression speed while keeping its space and decoding-time efficiencies, namely substituting

an approximation algorithm for the dynamic programming used by VSEncoding [13] when partitioning the input sequence into blocks, an experiment is also performed on TREC GOV2 collection to validate the proposed method, results show that our method significantly improves the compression speed with a slight loss at index size overhead.

The rest of this paper is organized as follows. Section 2 provides a background on compression techniques and partitioning strategies; Sect. 3 proposes our optimization on speeding up partitioning procedure for VSEncoding; Sect. 4 shows our experimental results and analyses of the original methods and their optimizations; conclusion and future work follow in Sect. 5.

2 Background

2.1 Index Compression

Compressing the index has long been a key issue for researchers to ensure both the time- and space-efficiency of inverted index, various encoders with different properties have been put up to settle this problem, and they can be roughly divided into two classes, namely the *integer-oriented encoders* and the *list-oriented encoders*. The integer-oriented encoders assign an unique codeword to each integer of the input sequence, then the compression procedure turns into a mapping or substitution from the integer space to code space. As they compress integers without considering their neighborings, the integer-oriented encoders are also called oblivious encoders [4], such as *unary code, Elias Gamma/Delta codes* and *Golomb/Rice codes*. Most integer-oriented encoders are hard to decode since they need bitwise operations to cross computer word boundaries, so byte/word-aligned encoders, are proposed to solve this problem, like *Variable Byte* and *Group Varint*, more importantly, they can be further improved by SIMD instructions of modern CPUs [14,15].

List-oriented encoders are designed to exploit the cluster of neighboring integers, each time a fixed-sized or variable-sized group of integers is binary packed with an uniform bit width, providing equivalent compression ratio and faster decoding speed, the technique used by these encoders is called *frame-of-reference* (FOR), or *binary packing* [8]. Basically, their compression ratios are inferior to these of the first category as a batch of integers are encoded indiscriminately, and useless zeros are padded in the codeword to keep word-aligned, however, when decoded, list-oriented encoders can obtain an entire block while the formers just decode one integer at a time. More importantly, with the help of *skip pointers* or *skip list*, it is possible to step along the codewords compressed by list-oriented encoders and stop when the required number of blocks has been bypassed. Examples of these encoders are *Simple Family*, AFOR and *Patched* FOR (PFOR, OptPFOR and FastPFOR).

2.2 Directed Acyclic Graph

One thing to be noted is that list-oriented encoders may cost equivalent time to compress the input sequence as the integer-oriented encoders even they are

designed to compress a list of integers at the same time. Before compressing, a partitioning strategy is needed to traverse the whole input list to search for an optimal partition, in consideration of compression ratio and decoding speed. Even after that, an uniform bit width has to be chosen to fit every element in for each block. As for Simple Family, a descriptor is decided after enumerating all the possible partitioning cases; OptPFOR needs an additional computation to choose the optimal proportion of exceptions in each block in order to achieve a better space efficiency. To speed up the compression speed, a proliferation of partitioning schemes have been seen in the last few years [1,6,12,13].

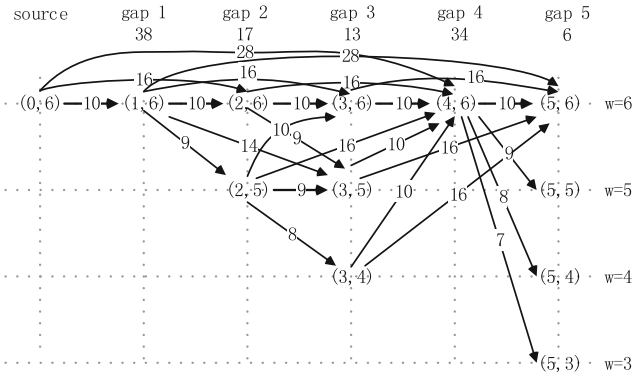


Fig. 1. Here is a DAG for sequence with 6 integers represented using gap (differences between them). Optional bit widths range from 3 to 5, each gap is fitted in the tuple (*position, available-bit-width*), if the available-bit-widths are more than one, they are placed in different rows, the number in the edge denotes the cost for this path, the fixed cost is set to 4.

The common foundation for the abovementioned methods is to recast the integer sequence $\mathcal{S}[0,n]$ to a particular DAG $\mathcal{G}$, each integer is represented by a vertex, plus a dummy vertex marking the end of the sequence, the graph $\mathcal{G}$ is complete, which means that for any i and j with $i < j \leqslant n$, there exists an edge connecting v_i and v_j, denoted as (v_i, v_j). In fact, the edge is an exact correspondence of a partition in the sequence $\mathcal{S}[i,j]$, the problem of fully partitioning $\mathcal{S}$ is converted to finding a path π in $\mathcal{G}$, for instance, $\pi = (v_0, v_{i_1})(v_{i_1}, v_{i_2}) \ldots (v_{i_{k-1}}, v_n)$ with k edges corresponds to the partition $\mathcal{S}[0, i_1 - 1]\mathcal{S}[i_1, i_2 - 1] \ldots \mathcal{S}[i_{k-1}, n-1]$ of k blocks. The weight of an edge in the graph is equal to the cost in bits consumed by the partition. Thus, the problem of optimally partitioning a sequence is reduced to the problem of Single-Source Shortest Path (SSSP) Labeling, as shown in Fig. 1. An intuitive way to solve this is to firstly set the cost of each vertex in $\mathcal{G}$ to $+\infty$, then an iteration starts from the left vertex to the rightmost, when it comes to a vertex v_j with $0 \leqslant j < n$, a subproblem of find the optimal path from v_j to v_n shows up, assuming the optimal path from v_0 to v_j has been correctly computed. Each

edge (v_j, v') outgoing from v_j will be assessed and cost of vertex v' is updated if it becomes smaller. As can be seen, the time complexity of this algorithm is proportional to the number of edges in $\mathcal{G}$.

However, the $\mathcal{G}$ transformed from integer sequence $\mathcal{S}$ is complete with $\Theta\left(n^2\right)$ edges, especially some posting lists for popular terms will be quite large, finding their optimal partitions will be intolerable. Since dynamic programming is inefficient and greedy mechanism is too coarse, an elaborate approximation algorithm which reduces the time and space complexities to linear at the cost of finding slightly suboptimal solutions will be feasible.

3 Optimizing Partitioning Strategy via Pruning DAG

While PEF using Elias-Fano code gets an impressive compression performance, we are aiming at revitalizing encoders using binary packing with optimizations. Since Elias-Fano code compresses integer in its complete form which may leads to poor space efficiency, binary packing uses gapped integer instead will obtain better compression ratio, it is still an promising method with potential for faster compression speed.

3.1 VSEncoding

VSEncoding is similar to PFOR, however, it neither applies a fixed-sized block length nor appends a patch for outliers at the end of blocks. In order to maximize the compression while retaining simple and fast decompression, VSEncoding partitions each posting list into blocks of variable length, and binary packs the integers inside of each block with the number of bits, say b, required to encode the largest one. Finally the the value of b and the length of the block, k, are encoded using distinct encoders $\mathcal{M}_1$ and $\mathcal{M}_2$. The basic form of each block can be seen as $\{< k_i, b_i >: data_i\}_{block_i}$, $data_i$ is the k_i integers of $block_i$ using b_i bits each.

Given a sequence $\mathcal{S}$ and a vector of partitions, the number of bits required to encode $\mathcal{S}$ can be computed in constant time, this quantity is calculated by summing up the costs of all the blocks, as the length of each block is variable, the problem of minimizing bits used relies on the problem of finding the optimal partitions. Let P denote the vector of m vertexes indicating the boundary of each block, with $P[0] = 1$, and $P[m] = n$. The problem can be represented as follow:

$$\min_{P \in \mathcal{S}} \sum_{i=0}^{m-1} c\left(P[i] - 1, P[i+1]\right)$$

where $c\left(P[i] - 1, P[i+1]\right) = |\mathcal{M}_1\left(b_i\right)| + |\mathcal{M}_2\left(k_i\right)| + k_i b_i$, namely the cost to encode i-th block.

VSEncoding adopts a dynamic programming algorithm to obtain the optimal partitions, each time a subproblem t consists in encoding in the best way all the integers starting from 0 to t, with a memo to look up when deciding whether to

merge or split current partition. The whole procedure starts by setting $t = 0$ and goes down to $t = n$, which represents the solution to the original problem. In order to implement a faster algorithm, the block lengths are restrained to some small constants between 16 and 64, say h, thus the time complexity drops from $O\left(n^2\right)$ to $O\left(n \log^2 h\right)$.

To further improve decompression speed and keep the block representations word aligned, VSEncoding reorganizes the layout of blocks, first the description parts are stored together in their order (i.e., $b_0 k_0, b_1 k_1 \ldots, b_m k_m$), then the data parts are written separately into each group: first the values that have to be represented with 1 bit, then with 2 bits, and so on. The decompression procedure is done by binary-unpacking and permuting the data parts into the correct order according to description parts.

3.2 Optimized Partitioning Strategy

Next we describe our modification to VSEncoding that achieve significant improvements over its original version in [13]. As mentioned before dynamic programming used in optimal partitioning costs too much time of index compression, to overcome this problem, we present a new partitioning scheme, which uses approximation algorithm in place of dynamic programming, thus reducing the time and space complexities to linear by finding a suboptimal partition. The partitioning problem can be reduced to SSSP over a DAG $\mathcal{G}$ with $\Theta\left(n^2\right)$ edges, and its time complexity is proportional to the number of edges, our aim is to design a pruning strategy removing edges of large costs while retaining edges which costs no more than $(1+\epsilon)$ times what the shortest paths do, for any given $\epsilon \in (0, 1)$.

We use $c\left(v_i, v_j\right)$ to denote the cost function of the edge $\left(v_i, v_j\right)$, which is also the cost of a partition $\mathcal{S}\left[i, j-1\right]$, U as the upper bound of cost by representing $\mathcal{S}$ as a single partition, and F as the fixed cost of each partition(e.g. the descriptor). There is an obvious fact that, given any $0 \leqslant i < j < k \leqslant n$, it is $0 < c\left(v_i, v_j\right) \leqslant c\left(v_i, v_k\right) \leqslant \ldots \leqslant c\left(v_i, v_{n+1}\right)$.

By adopting a nontrivial pruning strategy a subgraph $\mathcal{G}_\varepsilon$ of the original $\mathcal{G}$ is produced, in which the shortest path and the suboptimal path which increases a little are preserved. Any edge $\left(v_i, v_j\right)$ in $\mathcal{G}_\varepsilon$ follows at least one of the following conditions: (1) there exits a positive integer k such that $c\left(v_i, v_j\right) \leqslant F\left(1+\varepsilon\right)^k < c\left(v_i, v_{j+1}\right)$; (2) $j = n+1$. These edges of $\mathcal{G}_\varepsilon$ are called $\varepsilon - maximal$ edges. As we have set the upper bound to U, there exists at most $\log_{1+\varepsilon} \frac{U}{F}$ possible values for k, thus for each vertex $\mathcal{G}_\varepsilon$ has at most $\log_{1+\varepsilon} \frac{U}{F}$ outgoing $\varepsilon - maximal$ edges. [7] has proved that the shortest path distance on $\mathcal{G}_\varepsilon$, which is at most $(1+\varepsilon)$ times larger than the one in $\mathcal{G}$, can be computed in $O\left(n \log_{1+\varepsilon} \frac{U}{F}\right)$ time (Fig. 2).

The above procedure is like the trimming scheme for the subset-sum problem [5], which is to choose a representative for a compact range of its neighbor. In other words, we are sparsifying the complete graph $\mathcal{G}$ by quantizing its edge costs into classes of cost between $(1+\epsilon)^i$ and $(1+\epsilon)^{i+1}$, for each cost class of each node, only one $\epsilon - maximal$ edge is retained. If to prune $\mathcal{G}$ in a more coarse-grained but faster way, we can further remove edges which span too many vertex,

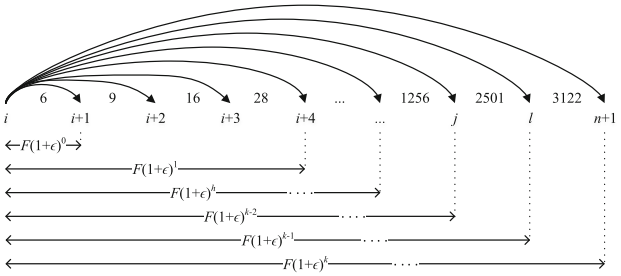


Fig. 2. Curves represent outgoing edges from vertex v_i, the number under each curve is the cost of it, postulating the fixed cost is $F = 4$. Lines below are cost classes with different k, for each class we can choose one edge as $\epsilon - maximal$ edge. This process is called sparsification.

intuitively, long edges are more vulnerable and sensitive to outliers, resulting in high cost and poor efficiency, thus are less likely to be enrolled in optimal partitioning. At the very beginning, we set the upper bound U to be the cost of representing the whole sequence $\mathcal{S}$ as a single block, however this is an asymptotic upper bound which might never be reached, by refining U to a more compact bound we can further lower the time complexity to linear without losing the approximation guarantees. Before giving our U we first state the following proposition to be based on:

Proposition 1. *For any* $0 \leqslant i < j < k \leqslant n+1$*, the weighting function satisfies* $c(v_i, v_k) + c(v_k, v_j) \leqslant c(v_i, v_j) + F + 1$.

It is easy to prove as we split one edge into two shorter edges, the correspondent interval in $\mathcal{S}$ is also partitioned. The worst case is the split cuts out nothing but only adds cost of one descriptor. For any edge (v_i, v_j) in $\mathcal{G}_\epsilon$, we have $c(v_i, v_j) \leqslant \overline{U}$, if $c(v_i, v_j) > \overline{U}$, then they are pruned and replaced by sub-edges in $\mathcal{G}_\epsilon$. These sub-edges can be found in a greedy way, in which the cost of each edge equals $\overline{U}$ (optimal ones are probably better), thus the number of edges cannot be larger than $\frac{c(v_i,v_j)-F}{\overline{U}-F} + 1$. Postulating all the sub-edges are the worst cases which only add costs, the overall cost follows the inequality:

$$\sum_{i \leqslant k < l \leqslant j} c(v_k, v_l) \leqslant c(v_i, v_j) + \left(\frac{c(v_i, v_j) - F}{\overline{U} - F} + 1 \right) (F + 1) \tag{1}$$

our ultimate goal is to keep the shortest path distance in $\mathcal{G}_\epsilon$ no more than $(1 + \epsilon)$ times the optimal one in $\mathcal{G}$, if (v_i, v_j) is one edge of the optimal path, we get the following inequality:

$$\sum_{i \leqslant k < l \leqslant j} c(v_k, v_l) \leqslant (1 + \epsilon) \cdot c(v_i, v_j) \tag{2}$$

combining these two we get an inequality for $\overline{U}$:

$$\overline{U} \geqslant F + \frac{F+1}{\epsilon} \tag{3}$$

One thing to be noted is that, even parameters ϵ and F are predefined by user, there is an implicit criterion that for any edge (v_i, v_j) the cost function must satisfy, since the correctness of pruning strategy relies on it: $\epsilon \cdot c(v_i, v_j) - F - 1 > 0$, which is not mentioned in [12]. Different lower bound of $c(v_i, v_j)$ determines the minimum of ϵ, for instance, if $c(v_i, v_j) \geqslant 2(F+1)$, then $\epsilon \in [0.5, 1)$. Finally we can set our $\overline{U}$ to $F + \frac{F+1}{\epsilon}$, reducing the time complexity from $O\left(n \log_{1+\epsilon} \frac{U}{F}\right)$ to $O\left(n \log_{1+\epsilon} \frac{1}{\epsilon}\right) = O(n)$, more importantly, the approximation guarantee is preserved.

Even providing the edge cost can be computed in constant time, we cannot check every edge of the complete graph $\mathcal{G}$ to determine whether it is $\epsilon - maximal$ or not, since it would take $\Theta\left(n^2\right)$ time. To construct the pruned graph $\mathcal{G}_\epsilon$ on the fly we need to deploy a dynamic data structure that maintains a set of *sliding windows* over $\mathcal{S}$ denoted by $\omega_0, \omega_1, \ldots, \omega_{\log_{1+\epsilon} \frac{1}{\epsilon}}$, each of them represents a cost class of $F(1+\epsilon)^k$, starting from the same vertex v_i but covering a different range of $\mathcal{S}$. For each vertex v_i, each sliding window ω_j begins to expand its size from the starting position to the position where the cost is larger than $F(1+\epsilon)^j$ for the first time, thus we generate all the $\epsilon - maximal$ edges outgoing from v_i. By performing a scan on $\mathcal{S}$ for each sliding window, we get and evaluate all the $\epsilon - maximal$ edges on-the-fly. Thus, the algorithm returns the optimal partition in $O\left(n \log_{1+\epsilon} \frac{1}{\epsilon}\right)$ time.

4 Experiments

4.1 Experimental Setup

In our experiments, we use the posting lists extracted from the TREC GOV2 collection, which consists of 25.2 million web pages and about 32.8 million terms in the vocabulary crawled from the gov Internet domain. The uncompressed size of these web pages is 426 GB. Also, all the terms have the Porter stemmer applied, and stopwords have been removed, docids are assigned in two ways: according to the lexicographic order of their URLs or to the order that they appear in the collection, thus we can see how docid reordering influence indexing performance. Then the docids and term frequencies are extracted from the collection in a non-interleaved way and applied with compression methods separately. To highlight the improvements between the original methods and our optimizations, the compression methods used in experiments are AFOR, VSEncoding via Dynamic Programming (VSE-DP) and VSEncoding via Optimal Partitioning (VSE-OP), we do not compare other methods like the Simple Family or PFOR in our benchmark as they have been throughly studied in the literature [4,9,11,17].

All the implementations are carried out on an Intel(r) Xeon(r) E5620 processor running at 2.40 GHz with 128 GB of RAM and 12,288 KB of cache. The default physical block size is 16 KB, algorithms are implemented using C++ and compiled with GCC 4.8.1 with O3 optimizations. In all our runs, the whole

inverted index is completely loaded into main memory, in order to warm up the execution environment, each query set is run 4 times for each experiment, and the response times only measured for the last run. Our implementations are available at https://github.com/Sparklexs/myVS.

4.2 Indexing Performance

The performance of indexing is based on the index size and compression speed. Before comparing the spaces obtained by different methods, we first set the parameters needed by AFOR, as mentioned before, to keep byte-aligned, we set the frame length to be (8, 16, 32), these configurations give the best balance between performance and compression ratio. Offering additional frame lengths will slightly increase the compression ratio, at the cost of linearly decreasing the compression speed.

Table 1. Total Size in GB, and corresponding average bits per integer (bpi) for docid and frequency, compressed by different methods on GOV2

methods	original				reordered			
	docid	bpi	freq	bpi	docid	bpi	freq	bpi
AFOR	16.30	30.17	8.10	14.95	9.78	18.05	6.39	11.81
VSE-DP	4.13	7.64	2.08	3.84	1.97	3.65	1.89	3.50
VSE-OP	4.22	7.80	2.25	4.17	2.04	3.76	2.04	3.77

Table 1 shows the index space, it is divided into two classes, the original and the reordered, the former denotes docids are assigned in the order they appear and the latter denotes docids are assigned in the lexicographic order of their URLs. The overall size and bits per integer of docid and frequency are shown separately. To facilitate reading, we fill docid-related cells with gray, so are the other tables below. We can observe that VSE-OP gets slightly worse result than its initial version, since we are focusing on accelerating compression speed, the partitioning scheme we adopt in VSE-OP is suboptimal, but it is still competitive compared with the optimal one. We can also observe VSEncoding achieves far more better compression ratio than AFOR, no matter for docid or frequency, the size of AFOR is nearly 4 times larger than VSEncoding, which demonstrates that a partitioning strategy, which offers more optional frame lengths, can more effectively utilize the distribution of integers to compress the sequence. Also note that docid after reordering is half the size it is ordered by the sequence of appearance, while frequency stays less sensitive to reordering, this can be explained by the fact that docid is stored in ascending order and reordering by URL further narrows the gaps between consecutive docids, however the frequency is aligned

with docid and stored in an unordered way, thus reordering does not make too much difference. Also the docids are quite sparse while the frequencies are fairly concentrated, rendering the docid's compression ratio twice larger than the frequency's.

Table 2. Total time elapsed in seconds and performance in million integers per second (mis) when compressing docid and frequency

methods	original				reordered			
	docid	mis	freq	mis	docid	mis	freq	mis
AFOR	78.98	58.95	72.56	64.17	75.43	61.72	66.92	69.58
VSE-DP	8303.96	0.56	8005.42	0.58	7849.07	0.59	7966.30	0.58
VSE-OP	5982.24	0.78	5819.72	0.80	5601.14	0.83	5648.14	0.82

Table 2 shows the compression speed of different methods when constructing index for docid and frequency. Combining with the index size shown in Table 1, we can find that there is a clear trade-off between compression efficiency and effectiveness, while index compressed by AFOR is quadruple the size of the ones compressed by VSEncoding, however, its compression speed is two orders of magnitude faster than VSEncoding. Due to a lack of attention, the construction time of experimented methods are rarely mentioned in the literature, we cannot find their performances on other platforms to compare. In our experiment, the optimized method outperforms its original version, the average time saved constitutes nearly 20 % of the time cost by the original. We can notice that the performance gap between docid and frequency is not very apparent, for that they contain the same number of integers. Also, the compression methods traverse the lists in a fixed way, which is irrelevant to the symbols the lists may contain. The only difference exists is that docid is sparser and larger than frequency, thus leading to the result that compressing docid is a little more time-consuming than compressing frequency. One exception is that VSE-OP partitions a list under the influence of the predefined parameter ϵ and the upper bound of representing the list in a single block, we can observe that compression speed between docid and frequency using VSE-OP is quite different. Another thing to be noted is that compressing reordered lists cost less time than the original lists both for docid and frequency.

Table 3 further details the performance of different methods. As is shown, Table 3 lists the number of partitioning schemes evaluated when traversing the integer sequences, comparing these different encoders, we can see that the partitioning step is crucial for compression speed. The optimized method sharply reduces the calculation needed to partition a posting list. By pruning edges with large cost, more than half of the calculation is saved by VSE-OP. We can also

note that all methods maintain the same calculation in the four columns except VSE-OP, it is easy to explain by the fact that time complexities of the first two methods are only relevant to the length of input sequences, while VSE-OP is relevant to both the length and the upper bound of the sequences. Calculation of frequency for VSE-OP stays the same after reordering because the changed integer distribution makes no difference to the upper bound of the sequence. However calculation for docid is slightly reduced as upper bound of the sequence is narrowed.

Table 3. Number of evaluated partitions by different methods

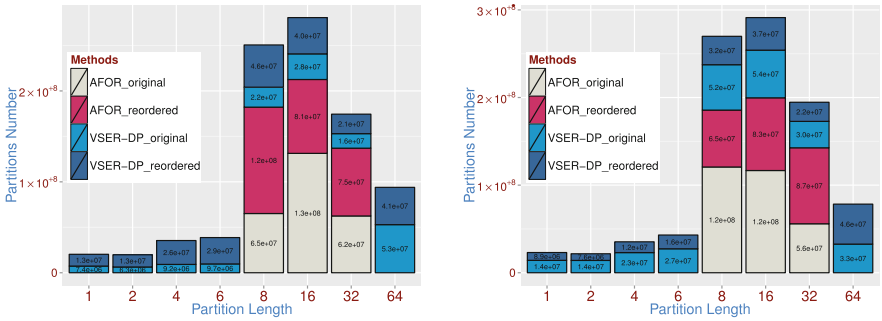
methods	original		reordered	
	docid	freq	docid	freq
AFOR	718,920,570	718,920,570	718,920,570	718,920,570
VSE-DP	293,805,525,502	293,805,525,502	293,805,525,502	293,805,525,502
VSE-OP	128,325,159,726	126,661,641,588	111,438,988,684	126,661,641,588

4.3 Decompression Performance

In this subsection, we are going to discuss the decompression performance of different methods, before reporting our results on the decompression speed, we first display the distribution of partition lengths, which can help us in understanding the differences among these partitioning schemes, and the correlation between clusters in a collection and partitions produced on it by variant methods.

As is discussed before, AFOR and VSE-DP use a partitioning strategy with fixed-sized lengths (8, 16, 32 for AFOR, and 1, 2, 4, 6, 8, 16, 32, 64 for VSE-DP), while VSE-OP uses a more flexible partitioning strategy, its partition lengths are determined by a series of sliding windows when traversing the sequence. To show the distribution clearly, we display the first two methods in histograms and VSE-OP in scatter plots as shown in Figs. 3 and 4.

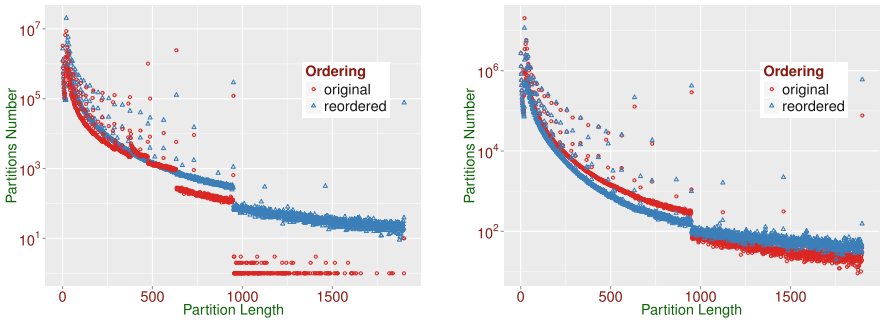
There is one important proposition we need to declare: there exists a compromise among partition length, compression ratio, compression and decompression speed. All the codewords that fall into one partition share the same bit width, thus the smaller the partition length is, the less bits will be wasted, however, smaller partition lengths also produce more fragmentations in one sequence, resulting in more CPU cycles and disk I/O needed to write and read these partitions. Vice versa, larger partition lengths are easier to access but less space-friendly. In Fig. 3, each bar indicates number of specified lengths used by one or more methods in pairs, each two adjacent blocks in one bar indicate the different number before and after reordering for one method, more exactly, the pale color represents the original and the dark color represents the reordered.



(a) docid (b) frequency

Fig. 3. Distribution of partition lengths produced by AFOR and VSE-DP

Figure 3(a) shows that after reordering these two methods produce more blocks with small lengths than before, so the compressed size contracts sharply. However, these methods produce more large lengths on frequency as shown in Fig. 3(b), resulting an inferior compression ratio than that on docid. When it comes to decompression speed (which will be listed below in Table 4), larger partitions imply faster access speed for a list of integers. From Fig. 3, we can find that AFOR has larger partition length than VSE-DP, so the disk I/O for it decreases orderly, however this only includes the disk transfer time, their time complexities on calculation do not change. Overall, these methods do not have a large variation in terms of partition lengths, which shows that they are able to adapt to the skewness of a dataset.



(a) docid (b) frequency

Fig. 4. Distribution of partition lengths produced by VSE-OP

Things go quite different for VSE-OP: as shown in Fig. 4, the partition lengths vary from 0 to larger than 1500 rather than being confined to a small range. In order to improve the interpretability of the result, we apply *log transformation*

to the original graph. As for the docid on the left hand, we can observe that partitions shorter than 500 compose the main proportion of the whole distribution, at the point of 1000, there exists a sharp segmentation, specially for the docid without reordering, number of such partitions drops to 1. However, reordering increases the number by almost one order of magnitude. With much more long partition allocated, VSE-OP still keeps its compression ratio close to VSE-DP, this illustrates its superiority on choosing partitions while avoiding wasting space caused by outliers. Also, different from Fig. 3, with quantity of longer partitions growing, size of VSE-OP after reordering decreases to half of the original as shown in Table 1, which again confirms that long partitions chosen by VSE-OP do not add load on space occupancy. Figure 4(b) for frequency shows similar result with that for docid, the only difference is that reordering does not affect the distribution as acutely as that in Fig. 4(a), this can be explained by the fact that reordering has no effect on the value range of frequency sequences.

Table 4. Total time elapsed in seconds and performance in million integers per second (mis) when decompressing docid and frequency

methods	original				reordered			
	docid	mis	freq	mis	docid	mis	freq	mis
AFOR	36.53	127.04	31.24	148.55	33.01	140.56	28.77	161.36
VSE-DP	69.09	67.17	71.97	64.48	52.98	87.60	70.11	66.19
VSE-OP	48.24	96.20	49.81	93.16	40.15	115.58	48.56	95.56

Table 4 reports the results on decompression speed, again our optimized method outperforms its original version. The decompression speed are much faster than the compression speed, however, with more partitions included, VSEncoding will encounter more skips when decompressing, which slows down its speed; on the other hand, long partitions also enable decompressing more integers in bulk. In a nutshell, the speed achieved by VSEncoding, specially by VSE-OP, is still competitive with AFOR, taking the compression ratio into account. Restrained by the configuration of our platform, the results in our experiment are quite different from that in [13], where VSE-DP is said to reach a speed of 450 ± 20 mis.

5 Conclusion and Future Work

In this paper we introduced and motivated the study of shortening compression time of inverted index via optimizing partitioning strategies. We first summarized a series of compression techniques which fall into the same category by treating the partitioning problem as SSSP over a DAG. Then we presented

our optimization on VSEncoding with a better space-time trade-off, namely to enhance its partitioning procedure with more flexibility and faster speed, while keeping its compression ratio and decompression speed competitive. At last, an extensive experimental analysis was given, which showed that our optimization significantly improve the compression speed as well as the decompression speed, with a little loss at in space efficiency.

There are still many open problems and opportunities for future research, since our solution only focus on optimizing VSEncoding, further experiments will investigate the consequence of optimizing other compression techniques and design a compression that offers better space-time trade-offs. An interesting problem would be using SIMD instructions to further accelerate the compression and decompression speed, also it would be promising to decrease the time complexity of the approximation algorithm adopted to compute optimal partitions.

References

1. Anh, V.N., Moffat, A.: Index compression using fixed binary codewords. In: Proceedings of the 15th Australasian Database Conference, vol. 27, pp. 61–67. Australian Computer Society, Inc. (2004)
2. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retr. **8**(1), 151–166 (2005)
3. Anh, V.N., Moffat, A.: Index compression using 64-bit words. Softw. Pract. Exp. **40**(2), 131–147 (2010)
4. Catena, M., Macdonald, C., Ounis, I.: On inverted index compression for search engine efficiency. In: de Rijke, M., Kenter, T., de Vries, A.P., Zhai, C.X., de Jong, F., Radinsky, K., Hofmann, K. (eds.) ECIR 2014. LNCS, vol. 8416, pp. 359–371. Springer, Heidelberg (2014)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
6. Delbru, R., Campinas, S., Samp, K., Tummarello, G.: Adaptive frame of reference for compressing inverted lists. Technical report, DERI-Digital Enterprise Research Institute, December 2010
7. Ferragina, P., Nitto, I., Venturini, R.: On optimally partitioning a text to improve its compression. Algorithmica **61**(1), 51–74 (2011)
8. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: Proceedings of 14th International Conference on Data Engineering, pp. 370–379. IEEE (1998)
9. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw. Pract. Exp. **45**(1), 1–29 (2015)
10. Manning, C.D., Raghavan, P., Schütze, H., et al.: Introduction to Information Retrieval, vol. 1. Cambridge university press, Cambridge (2008)
11. Ottaviano, G., Tonellotto, N., Venturini, R.: Optimal space-time tradeoffs for inverted indexes. In: Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, pp. 47–56. ACM (2015)
12. Ottaviano, G., Venturini, R.: Partitioned elias-fano indexes. In: Proceedingsof the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval, pp. 273–282. ACM (2014)

13. Silvestri, F., Venturini, R.: Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, pp. 1219–1228. ACM (2010)
14. Stepanov, A.A., Gangolli, A.R., Rose, D.E., Ernst, R.J., Oberoi, P.S.:SIMD-based decoding of posting lists. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, pp. 317–326. ACM (2011)
15. Trotman, A.: Compression, SIMD, and postings lists. In: Proceedings of the 2014 Australasian Document Computing Symposium, p. 50. ACM (2014)
16. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, San Francisco (1999)
17. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proceedings of the 18th International Conference on World Wide Web, pp. 401–410. ACM (2009)
18. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. (CSUR) **38**(2), 6 (2006)