A Simple Class of Efficient Compression Schemes Supporting Local Access and Editing

Hongchao Zhou, Da Wang, and Gregory Wornell
Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139
{hongchao, dawang, gww}@mit.edu

***Abstract*—In this paper, we study the problem of compressing a collection of sequences of variable length that allows us to efficiently add, read, or edit an arbitrary sequence without decompressing the whole data. This problem has important applications in data servers, file-editing systems, and bioinformatics. We propose a novel and practical compression scheme, which shows that, by paying a small price in storage space (3% extra storage space in our examples), we can retrieve or edit a sequence (a few hundred bits) by accessing compressed bits close to the entropy of the sequence.**

I. Introduction

Compression is a well studied subject with fruitful results on both fundamental limits and practical algorithms. However, traditional compression research mostly concerns with minimizing the coding rate, while in practice, other performance metrics such as the ease of partial retrieval and editing are also of great importance. For example, social networks and email services need to maintain a long list of sequences (such as user profiles, messages, emails, etc.) for millions of users. These sequences are highly redundant, and hence, compression is beneficial. Meanwhile, it is highly desired to be able to access or edit each individual sequence efficiently. Since the sequences are typically not long, it may not be efficient to manage them with conventional file systems, especially when the changes of sequences are very frequent. In this paper, we address this problem by proposing an architecture that compresses a large number of sequences with support for efficient add, read, or edit (update) operations for arbitrary sequence. While this architecture is general and applied for a wide range of sources, we use simple source models for the purpose of analysis and verify the practical applicability via simulations.

The problem of retrieval-efficient compression is investigated in [1]–[5]. In [1], Bloom filters were introduced as data structures for storing a set in a compressed form that allows membership query in constant time. In [3], a compression technique based on succinct data structure was provided, where source string is partitioned into segments, compressed separately based on arithmetic coding, and concatenated together. Here *succinct data structure* is applied to locate each compressed sequence. Another data structure based on tables for storing the positions of compressed sequences was investigated in [4]. However, their design is unsuitable for updating, because compressed sequences are placed in a concatenated form, and hence it does not support changing the length of compressed sequences, which may happen after updating as the compressed output of the updated sequence may change. The theoretical analysis, or in particular, the bounds on the number of bits that need to be read for retrieving a single source bit, was given in [5].

Another line of work focuses on compression that supports efficient local updating. The constructions are mainly based on sparse linear codes [6] or sparse graph codes. The basic idea of compressing an input string is to compute its syndrome with a sparse parity-check matrix, and the decompression process is to decode the syndrome based on the prior knowledge about the distribution of the input string. In [7], a "counter braids" scheme is developed that enables efficient updating (increment or decrement) for a set of integers (counters). In [8], this framework is extended to support efficient retrieval of an integer. Beyond integer sources, [9] provides construction for i.i.d. sources, but this construction is not very retrieval-efficient. One limitation of these schemes based on sparse graph codes is that it is "lossless" for a vanishing error probability, while in certain data storage applications zero-error compression is required. In addition, while [6], [9] are efficient for updating a bit, they are in general inefficient for updating a sequence, and they do not support other editing operations such as insertions and deletions.

Recently, Jansson et al. [10] and Grossi et al. [11] studied compression schemes that support both local retrieval and updating, which can be tailored to handle the problem of compressing a list of sequences. Attractively, they showed that the reading and updating can be done on blocks of $\log n$ bits in $O(1)$ time per bit while keeping the compression asymptotically optimal. In order to further support insertion and deletion operations, the time is increased to $O(\log n/\log\log n)$ per bit. The limitation of these schemes is that they do not allow the dynamic expansion of the size of the sequence list [10], which is important in web applications. Furthermore, these schemes have to maintain several tables as auxiliary data structure, which affects the overall storage efficiency and retrieving efficiency when the sequences are not long. For example, in experiment 1 of [10], the auxiliary data structure uses about 15% of the storage space, a non-negligible amount. In this paper, we introduce simpler and more practical schemes for compressing a large number of sequences that uses a very small amount extra storage space and allows us to add, read, or edit an arbitrary sequence very efficiently. And, we perform evaluations based on both ideal Bernoulli sources and real data

This work was supported in part by AFOSR under Grant No. FA9550-11-1-0183, and by NSF under Grant No. CCF-1017772.

from the social network Twitter, which achieve 97% and 93% storage efficiencies, respectively, and allow one to read or edit a sequence by accessing the amount of bits a little more than the entropy of the sequence.

II. Basic Compression Scheme

We assume that all the sequences are drawn independently from a distribution, which can be learned or estimated from enough samples of sequences. We compress each sequence based on this distribution with an efficient compression algorithm, and store compressed sequences in a compact form that allows us to retrieve or update any one of the sequences efficiently. We consider the regime that the length of each sequence is about a few hundred bits or at most several thousand bits. In this regime, little redundancy is introduced by the underlying compression algorithm, e.g., based on arithmetic coding, the expected length of a compressed sequence is at most $H+2$, where H is the entropy of the sequence. Meanwhile, this length allows us to achieve practically good locality: to read or edit a sequence (or part of a sequence), we only need to access a little more than H compressed bits. If the input sequences are too short, we can concatenate multiple sequences to form a new sequence of an appropriate length.

The difficulty of efficient updating comes from the fact that the length of a compressed sequence may change during updating. In contrast to previous schemes that dynamically modify auxiliary data structure to achieve efficient retrieval and updating [10], [11], we consider a very simple idea: we compress independent sequences $X_1, X_2, ..., X_t$ to t compressed sequences of lengthes $l_1, l_2, ..., l_t$, and then we store the t compressed sequences into t blocks with fixed size k.

Although this idea is simple, we show that this idea can lead to surprisingly good performance. In order to achieve the best performance, it requires us carefully designing the storage-management scheme (including the selection of the parameter k), and incorporating the underlying compression algorithm with the storage-management scheme, which is not straightforward.

We define the storage efficiency η_s of a scheme as the ratio between the entropy of the source and the total storage cost. Then

$$\eta_{\text{s}} = \frac{H(X)}{k} = \frac{H(X)}{\mathbb{E}(l)} \cdot \frac{\mathbb{E}(l)}{k}, \tag{1}$$

where $H(X)$ is the entropy of a sequence X, and $\mathbb{E}(l)$ is the expected length of a compressed sequence. It is clear that the upper bound of η_{s} is 1. Specifically, the overall storage efficiency attributes to two terms, $H(X)/\mathbb{E}(l)$ and $\mathbb{E}(l)/k$, corresponding to the efficiencies of the underlying compression algorithm and the storage of the compressed sequences, respectively.

III. Neighbor-Based Scheme

According to (1), in order to make the scheme storage-efficient, we need to utilize each size k storage block efficiently. In this section, we introduce a neighbor-based scheme, which compresses each sequence with length-prefix compression, and stores the overflows efficiently by using the extra space in neighboring blocks due to underflows.

(a) Sequences to store. (b) Storage result.

Fig. 1. Neighbor-based scheme: storage example.

A. Length-Prefix Compression

Here we introduce **length-prefix compression** as the underlying compression algorithm, namely, each sequence X_i is compressed into $f(X_i) = |g(X_i)| \cdot g(X_i)$, i.e., the concatenation of $|g(X_i)|$ and $g(X_i)$, where $g : \mathcal{A}^* \to \{0,1\}^*$ is a variable-length lossless compression without prefix constraints (g is not necessarily a uniquely decodable code), and $|g(X_i)|$ is the binary representation of the length of $g(X_i)$.

An optimal variable-length lossless compression without prefix constraints [12], denoted by g^*, lists all possible sequences in decreasing probabilities, and maps them, starting with the most probable one, to binary sequences of increasing lengths $\{\phi, 0, 1, 00, 01, 000, 001, 010, 011, ...\}$. For instance, let $x_0, x_1, x_2, ...$ be all the possible sequences in decreasing probabilities, then x_0 is mapped to ϕ and x_5 is mapped to 000. In [12], [13], Szpankowski and Verdú showed that the minimum expected length of the compressed sequence for a finite-alphabet memoryless source of fixed length m with known distribution is $\min \mathbb{E}(l) = mH - \frac{1}{2}\log m + O(1)$, where H is the entropy of each symbol. For our problem, the length of an input sequence may not be fixed, and we let m be the maximum possible length. In this case, the maximum length of $g^*(x)$ is $\lceil \log_2(\sum_{i=0}^{m} |\mathcal{A}|^i + 1) \rceil - 1$, and the number of bits needed to represent the prefix $|g^*(x)|$ is

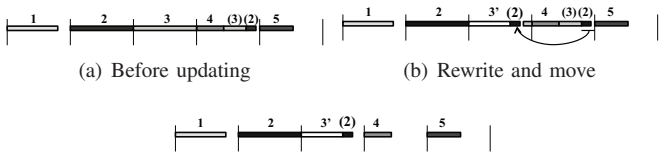
$$s_{\text{prefix}} = \lceil \log_2(\lceil \log_2(\sum_{i=0}^{m} |\mathcal{A}|^i + 1)) \rceil. \tag{2}$$

In the next subsection, we will show how length-prefix compression helps us to store, add, retrieve or update compressed sequences. Note that for some sequences that are not from memoryless sources or Markov sources, it might be computationally difficult to implement g^*. In practice, we can have a length-prefix compression by prepending a length field to an arbitrary source code such as a Lempel-Ziv code or an arithmetic code.

B. Storage of Compressed Strings

Here, we propose the neighbor-based storage scheme, which stores overflow in the neighboring blocks that have space available after storing its own sequence. In this scheme, retrieval and update can be achieved by searching sequentially in the neighboring blocks with the help of length-prefix compression. We store compressed sequences of different lengths within blocks of fixed size according to the following rules, as demonstrated in Fig. 1:

1) If a sequence does not overflow, i.e., is shorter than the block size k, it is placed in the corresponding block directly. See sequence 1, 4 and 5 in Fig. 1.
2) If a sequence overflows, i.e., is longer than the block size k, its first k bits are stored in the current block, and its overflow part is stored in the next few neighboring blocks that have extra storage space (we assume that the



(a) Before updating (b) Rewrite and move

(c) Clean

Fig. 2. Neighbor-based scheme: update (shrink)

blocks are cyclic, i.e., the next block of the end block is the first block).

3) The overflows are stored based on the nearest-neighbor-first order. This rule is important for efficient local updating and retrieval. For instance, for the extra space of the 4th block in Fig. 1, we first store the overflow of sequence 3, denoted by subsequence (3). If there are still extra space afterwards, we store the overflow of sequence 2, denoted by subsequence (2).

In order to support local retrieval, we compress all the original sequences based on length-prefix compression, so that the length of each compressed sequence can be obtained by reading the first s_{prefix} bits (prefix) of its corresponding block.

1) Retrieval: Assume sequence i is stored spanning over block i to block j with $j \geq i$. If we know the lengths of sequence i to sequence j, i.e., $l_i, l_{i+1}, ..., l_j$, then the locations of storing sequence i can be uniquely determined.

To retrieve sequence i, we first get its length l_i by reading its prefix in block i, by which, we determine whether it has overflow. If there is no overflow, we simply retrieve the sequence; otherwise, we move to block $i+1$, read l_{i+1}, and then decide if we need to read data from block $i+1$, which in turn depends on if sequence $i+1$ has overflow or not. Repeating this procedure, eventually all data for sequence i will be read, while we incur the overhead of reading the length prefixes in some later blocks. For the example in Fig. 1, to retrieve sequence 2, we first read l_2 and realize there is overflow and the overflow size is $o_2 = l_2 - k$. Then we move to block 3 and read l_3, and realize there is overflow for sequence 3 as well. At block 4, we read l_4, skip the data for sequence 4 and overflow data for sequence 3, and then read the rest of sequence 2 (o_2 bits) by starting from the $l_4 + o_3 + 1$th bit of block 4.

2) Update: The update process is further divided into shrinking and expanding, which involve different operations. In order to support efficient local updating, we further modify the storage structure described above: if a block is full, we set its last bit as 1; if a block is not full, we fill its empty part with $100...0$, hence, its last bit is naturally 0. Note that with this modification, the available storage space of each block becomes $k-1$.

Fig. 2 demonstrates the shrinking process (Expanding a sequence i is the inverse process of shrinking it). To shrink a sequence i, we need not only to rewrite the content of sequence i, but also to handle the overflows that come from sequences before sequence i. Fortunately, these overflows can be located based on the lengths of a few sequences after i and the storage statuses (full or non-full) of their corresponding blocks. In Fig. 2, to update sequence 3 to sequence $3'$ with a shorter length, we can locate subsequence (3) with the length of sequence 3 and the length of sequence 4. We can further locate (2) if we know the length of the empty part in block 4 (or the first non-full block if block 4 is full). This can be achieved by reading inversely from the end of block 4 until meeting the first 1, since the empty part is in the form of $100..0$. After locating (3) and (2), we rewrite sequence 3 with sequence $3'$, and move the overflow part (2) forward according to the amount shank. Finally, we clean the original space of storing sequence 3 and overflow (2), so that the empty part of each block is in the form of $100..00$.

3) Add: The process of adding a new sequence is very similar to the process of updating a compressed sequence perfectly fits a block to a new sequence.

C. Analysis

We now study the cost of retrieving a sequence or updating a sequence based on the storage mechanism above, and investigate the effect of the block size k.

We define the cost of retrieving a sequence as the total number of bits to read. The cost of retrieving sequence i is

$$\gamma_{\text{retrieve}}(i) = l_i + (w_i - 1)s_{\text{prefix}}. \tag{3}$$

where l_i is the length of the compressed sequence for sequence i, and w_i is the number of blocks that sequence i spans over. For instance, in Fig. 1, sequence 2 spans over 3 blocks and sequence 3 spans over 2 blocks. We let l be the length of a randomly selected compressed sequence and w be the number of blocks it spans over, then the expected cost of retrieving a sequence is

$$\gamma_{\text{retrieve}} = \mathbb{E}(l) + (\mathbb{E}(w) - 1)s_{\text{prefix}}, \tag{4}$$

where $\mathbb{E}(w)$ is a non-increasing function of k.

Assume that the lengths of compressed sequences are i.i.d. distributed, we define $L_i = \sum_{j=1}^{i} l_i$ as the total length of i compressed sequences, then

$$\mathbb{E}(w) \quad \leq \quad 1 + \sum_{j=1}^{\infty} P(L_j > j(k-1)), \tag{5}$$

which can be computed based on the length distribution of compressed sequences. If the sequence distribution is given, we can compute the length distribution of compressed sequences:

$$P(l) = \sum_{i=2^{(l-s_{\text{prefix}})}-1}^{2^{(l-s_{\text{prefix}}+1)}-1} \mathbb{P}\left[x_i\right], \tag{6}$$

where $x_0, x_1, ..., x_{2^m-1}$ are all possible input sequences in decreasing probabilities, and $\mathbb{P}\left[x_i\right]$ is the probability of the sequence x_i.

In our scheme, to update a sequence, the number of bits to read is typically much less than the number of bits to write. In addition, in storage systems such as HDD and SSD, the latency and power consumption of write operations are higher than those of read operations. Hence, we consider the cost of updating a sequence as the total number of bits to write.

The cost of updating sequence i with compressed sequence from length l_i to l_i' is

$$\gamma_{\text{update}}(i) = \max[l_i, l_i'] + r_i + \xi_i, \tag{7}$$

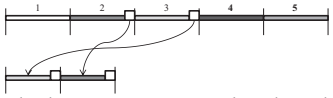


Fig. 3. Pointer-based scheme: storage example, where the white squares are pointers.

where r_i is the length of total overflows that come from sequences before i and stored at blocks i and after, and ξ_i is the number of blocks that change status during the updating process, either from non-full state to full state or from full state to non-full state. In general, ξ_i is much smaller than the other two terms, hence, we ignore it in our analysis. As a result, the expected cost of updating a sequence is

$$\gamma_{\text{update}} \simeq \mathbb{E}(\max[l, l']) + \mathbb{E}(r). \tag{8}$$

Assume that the lengths of compressed sequences are i.i.d. distributed, then

$$\mathbb{E}(r) = \max[\mathbb{E}\max_{i=1}^{\infty}(L_i - i(k-1)), 0]. \tag{9}$$

Based on the above analysis, we can compute $\gamma_{\text{retrieve}}, \gamma_{\text{update}}$, i.e, the expected costs to retrieve or update a sequence.

IV. Pointer-Based Scheme

As a comparison, we consider another natural idea of implementing the basic compression scheme that is based on fixed-length source coding, i.e., mapping each sequence into a binary sequence of fixed length. Hence, we can store each compressed sequence into a block respectively, and it allows us to retrieve or update every sequence very efficiently. Here, we consider an optimal fixed-length source code that is constructed as follows. Let $x_0, x_1, ...$ be all the possible input sequences in decreasing probabilities. Then a sequence x_i is encoded as the binary representation of i if $i < 2^{k-1}$; otherwise, it is an exception. Clearly, not all possible input sequences can be mapped into compressed sequences of length $k-1$, hence, we need to handle the exceptions, which appear with a probability ϵ. It is easy to get that

$$\epsilon = 1 - \sum_{i=0}^{2^{(k-1)}-1} \mathrm{P_r}(x_i). \tag{10}$$

Let m be the maximum length of the binary representation of the input sequences. Given a sequence X and a fixed-length source coding, if X can be mapped into a compressed sequence of length $k-1$, we store the compressed sequence in the corresponding block of length k with an additional bit '0' as the prefix. If X is an exception, we store X directly in the corresponding block with an additional bit '1' as the prefix. However, $1X$ is too big to be filled into the block, so we store a part of it in the block of length k, and store the rest in an additional block of length k', where the two blocks are linked to each other with pointers.

Fig. 3 demonstrates a storage example of the pointer-based scheme, where the t sequences are stored in t blocks of length k, and t' $(t' < t)$ additional blocks of length k'. If a sequence is stored in a block with index i and an additional block with index i', then the pointers record the index i' and the index i respectively. With this data structure, it is very easy to add or retrieve a sequence. For example, to add a new sequence

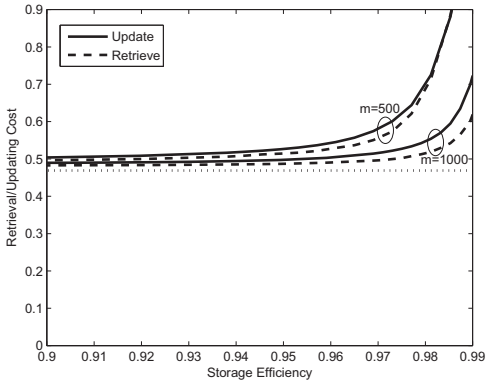


Fig. 4. The unit retrieval/updating cost versus the storage efficiency for Bernoulli-0.1 sources of length m with the neighbor-based scheme. The horizontal dotted line is $H(0.1)$, which is the theoretical lower bound for the unit retrieval/updating cost.

X that is an exception, we store it into a block of length k with index $t+1$, and a block of length k' with index $t'+1$. Updating a sequence is also not difficult. For example, in Fig. 3, assume that we change sequence 3 to a typical sequence that can be stored within a single block of length k, then we can release the storage space of the 1st additional block. For the convenience of management, we move the last additional block to the released additional block, and update the pointer linked to it (since its index is changed).

The storage efficiency of the pointer-based scheme is

$$\eta_s = \frac{H(X)}{k + \epsilon k'}, \tag{11}$$

where $H(X)$ is the entropy of a sample sequence X. The expected cost of retrieving a sequence is

$$\gamma_{\text{retrieve}} \simeq k + \epsilon k', \tag{12}$$

and the expected cost of updating a sequence is

$$\gamma_{\text{update}} \simeq (1-\epsilon)^2 k + (2\epsilon - \epsilon^2)(k + k'). \tag{13}$$

V. Performance Evaluation

The compression schemes proposed in this paper can work for very general sources. In this section, we evaluate the performance of these schemes based on both ideal Bernoulli-p sources and real data from Internet.

First, we consider a simple case: all the sequences are Bernoulli-p sequences of length m. We are interested in the tradeoff between the storage efficiency and the retrieval/update efficiency. Here, $u_{\text{retrieve}} = \gamma_{\text{retrieve}}/m$ is the cost per source bit when retrieving a sequence. It reflects the retrieval efficiency of our compression scheme, and we define it as the *unit retrieval cost*. Similarly, $u_{\text{update}} = \gamma_{\text{update}}/m$ is the cost per source bit when updating a sequence to another one of the same length, and we define it as the *unit update cost*. The theoretical lower bound of both u_{retrieve} and u_{update} is $H(p)$, which is the entropy of a Bernoulli-p bit.

Fig. 4 shows the trade-off between the unit retrieval/updating cost and the storage efficiency for Bernoulli-0.1 sequences with the neighbor-based scheme. Note that the storage efficiency is proportional to the inverse of the block length k, so we can get k from the storage efficiency. When

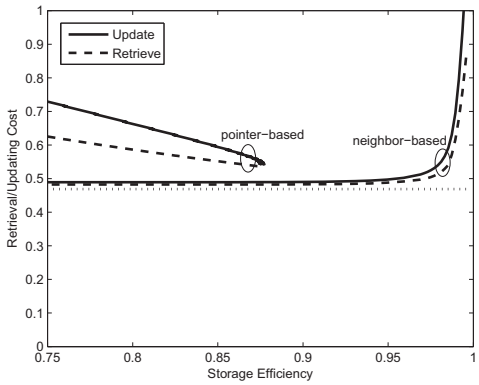


Fig. 5. The unit retrieval/updating cost versus the storage efficiency for Bernoulli-0.1 sequences of length $m = 1000$ bits. Here, we assume that the length of each pointer is 10 bits.

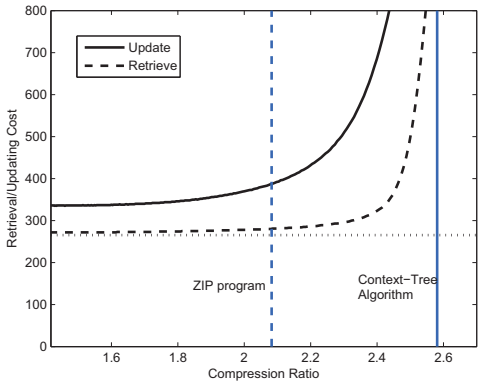


Fig. 6. The cost to retrieve/update a tweet versus the compression ratio for storing $70,000$ random tweets from Twitter with the neighbor-based scheme. The vertical lines are for the compression ratios of the ZIP program and the Rissanen's context-tree algorithm, when we compress all the tweets without considering the locality requirement. The horizontal dotted line is the estimated entropy of a tweet based on the context-tree model.

the length of each sequence is $m = 1000$ bits, the proposed neighbor-based scheme can reach 98% storage efficiency while the unit retrieval/updating cost is below 0.6 reads/writes per source bit. Note that this high storage efficiency already takes the loss introduced by the underlying compression algorithm into account. For this example, updating a sequence of 125 bytes, we only need to change at most 75 bytes. If we reduce the sequence length to $m = 500$ bits, the compression scheme still can reach 97% storage efficiency while supporting efficient local operations.

In Fig. 5, we further compare the performance of the neighbor-based scheme and that of the pointer-based scheme when the sequences are Bernoulli-0.1 sequences of length $m = 1000$ bits. It is a little surprising that there is a big performance gap between them, revealing the significance of the neighbor-based scheme: the neighbor-based scheme can reach 98% storage efficiency while the pointer-based scheme can only reach 88% storage efficiency (the optimal ϵ is 0.0173). One of the reasons is that optimal variable-length source coding is typically much more efficient than optimal fixed-length coding in the (short) finite block-length regime. Although the performance of the pointer-based scheme could be further improved via introducing more levels of fixed-length block storage (currently we use two levels), this makes the scheme more complex, and sensitive to the distribution of the sequences, and hence less practical.

We also implement the neighbor-based scheme on real data from the social network Twitter, where we are trying to compress $70,000$ random English tweets, which are short message with mean length 91.8 characters (each character consists of 8 bits) and maximum length 164 characters, including the user names. In this implementation, we use the Rissanen's context-tree algorithm [14] to compress each tweet and prepend a length field to the compressed sequence, as the underlying length-prefix compression. The context-tree model is created based on all the tweets. Fig. 6 shows the cost (the number of bits to read/write) to retrieve or update a tweet versus the overall compression ratio. Here, updating a tweet means that a tweet is changed to another randomly selected tweet. From the figure, we see that the neighbor-based scheme can achieve a compression ratio 2.4 (defined as the ratio between the number of the original data bits and the compressed data bits) while each tweet can be retrieved or updated by accessing a few hundred bits. By contrast, if we compress all the tweets together as a single string without considering the locality requirement, the widely used ZIP program can only achieve a compression ratio 2.1, and the Rissanen's content-tree algorithm can achieve a compression ratio slightly smaller than 2.6. Therefore, the neighbor-based scheme operating at compression ratio 2.4 has a 7% storage-efficiency loss. Note that this storage-efficiency loss is higher than that for the Bernoulli-0.1 sequences, because the variance of the tweet length is much larger.

REFERENCES

[1] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] D. Belazzougui, F. Botelho, and M. Dietzfelbinger, "Hash, displace, and compress," in *Algorithms - ESA 2009*, vol. 5757, 2009, pp. 682–693.

[3] M. Patrascu, "Succincter," in *Proc. 49th Annual IEEE FOCS*, Oct. 2008, pp. 305–313.

[4] P. Ferragina and R. Venturini, "A simple storage scheme for strings achieving entropy bounds," *Theoretical Computer Science 372(1)*, pp. 115–121, 2007.

[5] M. M. A. Makhdoumi, S.-L. Huang and Y. Polyanskiy, "On locally decodable source coding," *arXiv:1308.5239*, 2013.

[6] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. on Information Theory*, vol. 45, no. 2, pp. 399–431, 1999.

[7] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," in *Proc. ACM SIGMETRICS*, New York, NY, USA, 2008, pp. 121–132.

[8] V. Chandar, D. Shah, and G. Wornell, "A locally encodable and decodable compressed data structure," in *Proc. 47th Annual Allerton Conference on Communication, Control, and Computing*, 2009, pp. 613–619.

[9] A. Montanari and E. Mossel, "Smooth compression, gallager bound and nonlinear sparse-graph codes," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Jul. 2008, pp. 2474–2478.

[10] W. S. J. Jansson, K. Sadakane, "CRAM: Compressed random access mem- ory," *ICALP (1). Volume 7391 of LNCS*, pp. 510–521, 2012.

[11] S. R. R. Grossi, R. Raman and R. Venturini, "Dynamic compressed strings with random access," *ICALP*, pp. 504–515, 2013.

[12] W. Szpankowski and S. Verdú, "Minimum expected length of fixed-to-variable lossless compression without prefix constraints," *IEEE Trans. on Information Theory*, vol. 57, no. 7, pp. 4017–4025, 2011.

[13] S. Verdú and I. Kontoyiannis, "Lossless data compression rate: Asymptotics and non-asymptotics," in *46th Annual Conference on Information Sciences and Systems, Princeton University*, 2012.

[14] J. Rissanen, "A universal data compression system," *IEEE Trans. on Information Theory*, vol. 29, no. 5, pp. 656–663, 1983.