Lempel-Ziv Compression With Random Access

Shaik Mastan Vali, Srikanth Kamparaju
Supervisor - Dr. Shashank Vatedka

1 Introduction

In the last decade, the amount of data generated has increased dramatically. Data from genetics, video data, virtual reality, and other fields is exploding at an unprecedented rate. This data avalanche has necessitated the development of more effective compression methods for data storage and management. Data storage, on the other hand, is no longer for archival purposes. In modern applications, the majority of the stored data must be retrieved and analysed on a regular basis in order to make statistical choices. This has prompted the development of a new class of data compressors: efficient compressors that allow for computation on the compressed data while remaining rate efficient. shows up when source sequences are very long. In other words, to query a single source symbol,traditional algorithms require accessing a number of stored bits that scale with the length of the sequence. So asymptotically, as the sequence length n tends to infinity, one needs to access a large number of stored bits (scaling with n) to query a single bit.We are interested in constructions that allow close-to-optimal compression with efficient random access. When the compression sizes are quite large, the ability to deal with the compressed data without first decompressing the dataset is especially important (say 100x). We've previously seen how they can help with a variety of issues. In the realm of genomics, for example, much effort has gone into developing compressors that efficiently compress the 2D-tabular variant-call dataset while also allowing for random decompression of any row or column.

We're looking for a way to save a source sequence with the fewest possible bits while yet permitting a query of any place in the sequence using only a few bits of the compressed sequence. To recover a single symbol using classic compressors like Lempel-Ziv based techniques, one must access the complete compressed sequence. This is desired since these algorithms' space efficiency becomes apparent when source sequences are quite long. To put it another way, standard algorithms involve accessing a number of stored bits that grow with the length of the sequence to query a single source symbol.As the sequence length n approaches infinity, accessing a large number of stored bits (scaling with n) is required to query a single bit.We're looking for data structures that allow for near-optimal compression while maintaining efficient random access.

2 Problem Setup

A sequence X^n is generated from a stationary and ergodic source with alphabet x. We would like to losslessly store X^n using the minimum expected number of bits such that every symbol $X_i, \mathrm{i} = 0, \ldots, \mathrm{n}-1$, can be reliably reconstructed by accessing no more than to of the stored bits in the memory. A compression scheme with blocklength n, random access $t^{(n)}$, and the rate $R^{(n)}$ is defined by an encoder:

$$f^{(n)} : \boldsymbol{X}^{(n)} \to \{0,1\}^*$$

And decoder pairs $\left(S_i^{(n)}, g_i^{(n)}\right)$:

$$S_i \subseteq N, \left|S_i^{(n)}\right| = t^{(n)},$$
$$g_i^{(n)} : \{0,1\}^{t^{(n)}} \to x, \forall i = 0, \ldots, n-1$$

such that the following are true:

$$R^{(n)} = \frac{E\left[f^{(n)}\left(X^n\right)\right]}{n},$$
$$g^{(n)}\left(\left\{f^{(n)}\left(X^{(n)}\right)\right\}_{S_i^{(n)}}\right) = X_i$$

Here, $\left\{f^{(n)}\left(X^n\right)\right\}_{S_i^{(n)}}$ denotes the subsequence of $f^{(n)}\left(X^n\right)$ that belongs to the positions in $S_i^{(n)}$. The pair (R,t) is said to be achievable if there exists a sequence of compression schemes (indexed by n) such that

$$\begin{array}{lll} \lim_{n\to\infty} & R^{(n)} & \to R \\ \lim_{n\to\infty} & t^{(n)} & \to t \end{array}$$

In this work, We implement and compare the average random time vs compression size of three different compression schemes such as RLZ , Dense sparse and BV-RLZ

3 Compression Schemes

3.1 RLZ Compression

In this scheme, we divide the file into fixed-size chunks and compress each chunk using lempel-ziv algorithm both with dictionary and without dictionary compression.

Compression: The sequence X is divided into blocks of length b and is compressed block by block. Let the compressed length of the i^{th} block be L_i and the number of blocks be N_b. The header h is stored as follows-

$$h_i = \sum_{j=0}^{i} L_j, \text{ for } 0 \leq i \leq N_b - 1$$

Decompression: To access the i^{th} block, the start position and the compressed size of the block is obtained from the header as follows-

$$start_i = h_i$$
$$size_i = h_{i+1} - h_i, \text{ for } 0 \leq i \leq N_b - 1$$

Using these values, the block can be accessed and decompressed.

3.2 Dense-Sparse stream based compression

In this scheme, the file X is divided into fixed-size chunks and each chunk is stored using two streams: dense stream D and sparse stream S. This scheme is a modified version of [1].

Compression: Let C_i be the compressed block corresponding to the original block B_i, and L_i be the length of C_i. The typicality of the block is decided using the threshold $T = b(H(X) + \epsilon)$.

- If $C_i \leq T$, then

$$D_i = C_i[0 \dots L_i - 1]$$
$$S_i = 0^b$$

- Else,

$$D_i = C_i[0 \dots T - 1]$$
$$S_i = C_i[T \dots L_i - 1]$$

Each of D_i and S_i are padded with zeroes such that the length of D_i is T and the length of S_i is b. Finally, the sparse stream S is compressed using a sparse bit-vector compressor.

The header h is stored as follows-

$$h_i = \sum_{j=0}^{i} L_j, \text{ for } 0 \leq i \leq N_b - 1$$

Decompression: To decompress the i^{th} block, the LZ-compressed block $C_i^{'}$ needs to be constructed. The start position and the compressed size of the block is obtained from the header as follows-

$$start_i = h_i$$
$$size_i = h_{i+1} - h_i, \text{ for } 0 \leq i \leq N_b - 1$$

The compressed block $C_i^{'}$ is constructed as follows-

- If $size_i \leq T$, then

$$C_i^{'} = D[0 \dots size_i - 1]$$

- Else,

$$C_i^{'} = D[0 \dots T-1] \quad || \quad S[0 \dots size_i - T - 1]$$

where || is the concatenation operator.

The sparse stream values are accessed using the random access operator on the sparse bit-vector. The block $C_i^{'}$ can now be decompressed using the Lempel-Ziv algorithm to obtain B_i.

3.3 BV-RLZ compression

In this scheme, the file is divided into fixed-size chunks and each chunk is compressed, same as in the RLZ scheme. The header is stored as a compressed bit-vector.

Compression: The sequence X is divided into blocks of length b and is compressed block by block. Let the compressed length of the i^{th} block be L_i. The header h is stored using the bit-vector data structure. Let pos be defined as -

$$pos_i = \sum_{j=0}^{i} L_j, \text{ for } 0 \leq i \leq N_b - 1$$

Let $L_X = \sum_{j=0}^{N_b-1} L_i$ be the size of the compressed file. Initially, $h = 0^{L_X}$. The bits of h are set as follows-

$$h_{pos_i} = 1, \text{ for } 0 \leq i \leq N_b - 1$$

The bit-vector h is compressed using the sparse bit-vector compressor.

Decompression: To access the i^{th} block, the start position and the compressed size of the block is obtained from the header as follows-

$$start_i = select_h(i+1)$$
$$size_i = select_h(i+2) - select_h(i+1), \text{ for } 0 \leq i \leq N_b - 1$$

where $select_h(i)$ is the standard select operator on the compressed bit-vector h. Using these values, the compressed block can be constructed and decompressed using the LZ algorithm.

4 Experiments and Results

The three compression schemes were tested on the Wikipedia corpus (enwik8, enwik9), IMDB principal data and a protein FASTA file both with and without using a dictionary. The ZSTD library was used for LZ compression and SDSL library was used for bit-vector compression. The experiments were run on Linux machine having Ubuntu 20.04.3 LTS and AMD EPYC 7452 32-Core Processor with 128GiB RAM. All the methods were implemented in C++14 and compiled using g++ 9.3.0.

The values of chunk size b used were in the range $[fileSize/10000, fileSize/1000]$, where $fileSize$ is the size of the original file. The value of ϵ for the threshold used was 0.01.

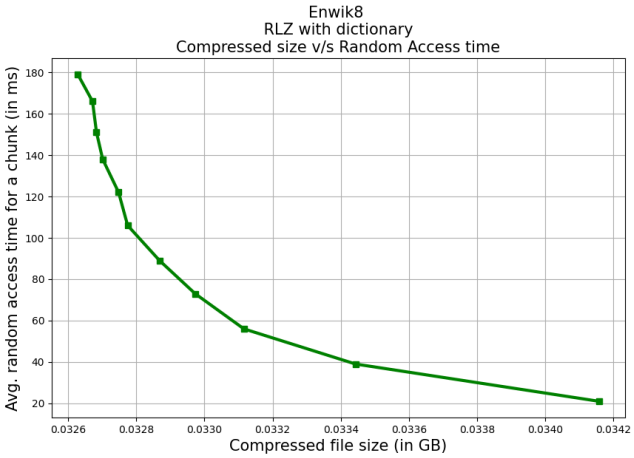

Figure 1: Compressed size v/s random access time for RLZ scheme with dictionary

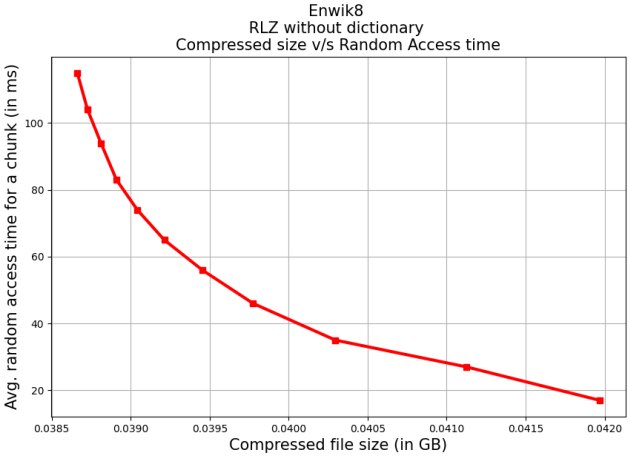

Figure 2: Compressed size v/s random access time for RLZ scheme without dictionary

From fig-1 and fig-2 it can be observed that the RLZ scheme using a dictionary gives a smaller compressed file, but consumes more time for random access due to the additional overhead.

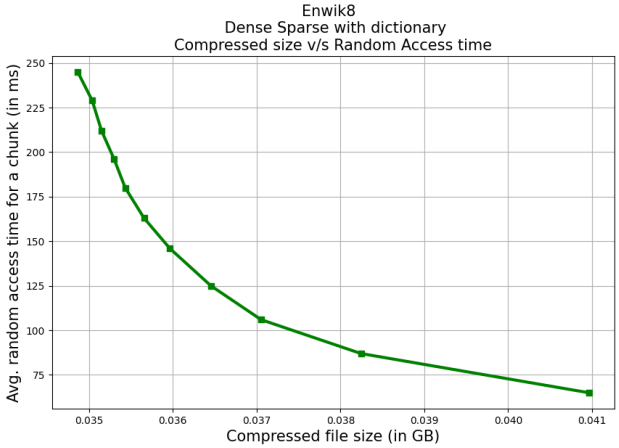

Figure 3: Compressed size v/s random access time for Dense-sparse scheme with dictionary

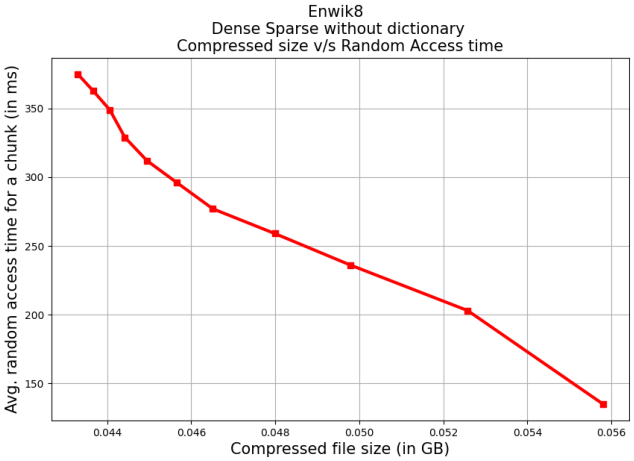

Figure 4: Compressed size v/s random access time for Dense-sparse scheme without dictionary

From fig-3 and fig-4 it can be observed that the dense-sparse scheme using a dictionary gives a smaller compressed file and consumes less time for random access, because of the smaller compressed size of each chunk and thereby less

random access time.

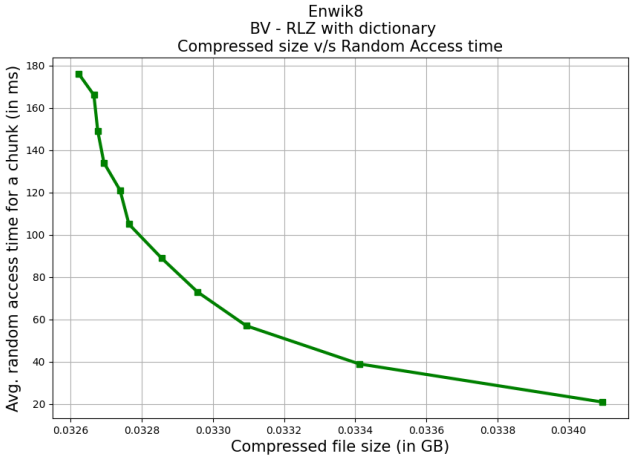

Figure 5: Compressed size v/s random access time for BV-RLZ scheme with dictionary

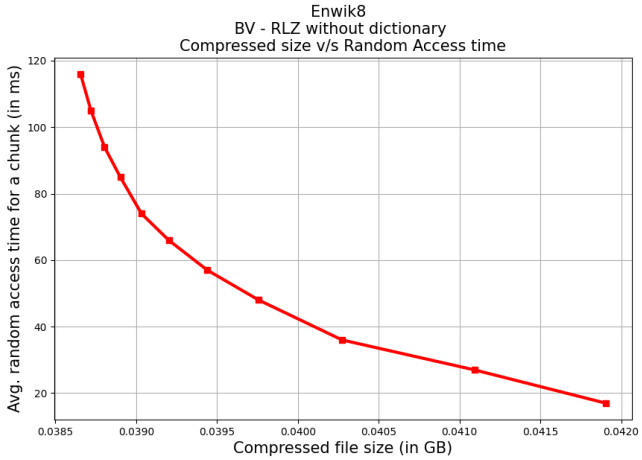

Figure 6: Compressed size v/s random access time for BV-RLZ scheme without dictionary

From fig-5 and fig-6 a conclusion similar to the RLZ scheme can be derived.

Further, a custom method `get(start, length)` was implement to optimize the random access time in dense-sparse scheme. The method returns the subsequence $Y[start \dots start + length - 1]$ from C_Y, where C_Y is the compressed bit-vector corresponding to the bit-vector Y. Suppose that we need to access a subsequence of length k in the compressed bit-vector C_Y. Traditionally, using the random access operator `[]` pre-defined in the SDSL library, the number of computations required is $\mathcal{O}(k(t_{select} + n/m))$, where m is the number of zeroes in the original bit vector Y. The custom method achieves this using $\mathcal{O}(t_{select} + kn/m)$ computations by using a single `select()` method call.

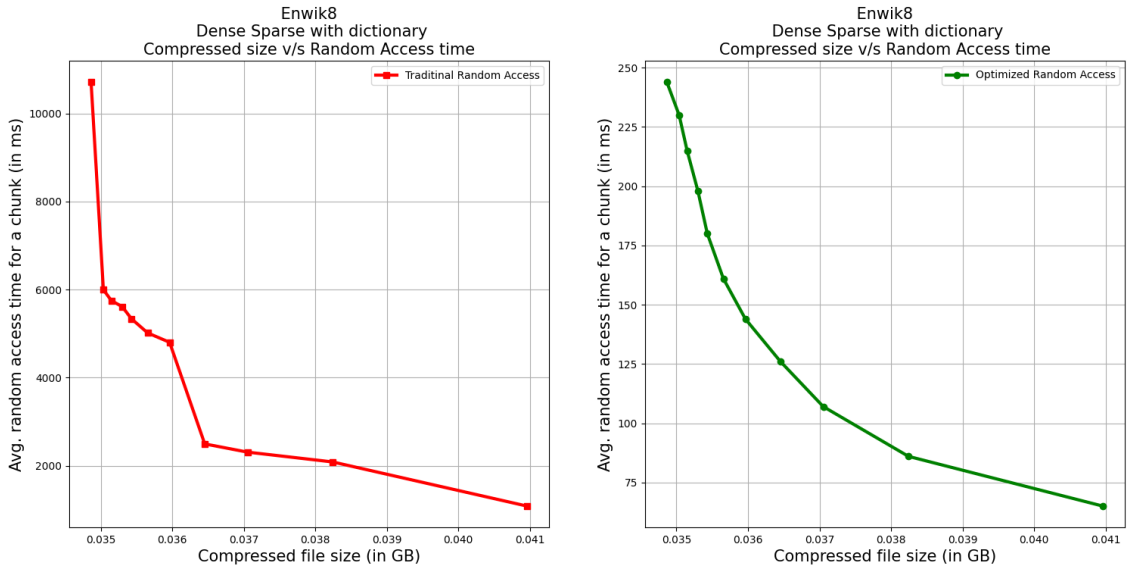

Figure 7: Compressed size v/s random access time for dense-sparse scheme using the traditional and optimized random access

From fig-7, a significant improvement in random access time can be observed using the optimized random access, while the compressed file size remains same.

The source codes, instructions to run and text file links are in the following repository

5 References

1. K.Tatwawadi, Shirin Saeedi Bidokhti, Tsachy Weissman, "On Universal Compression with Constant Random Access"
2. Thomas M Cover, Joy A Thomas, "Elements of Information Theory" `enumerate` environment.
3. Kewen Liao, et al, "Effective Construction of Relative Lempel-Ziv Dictionaries"
4. Zstandard - Real-time data compression algorithm
5. Getting Started - Zstd dev
6. SDSL library