A Randomly Accessible Lossless Compression Scheme for Time-Series Data

Rasmus Vestergaard, Daniel E. Lucani, and Qi Zhang
DIGIT and Department of Engineering, Aarhus University, Denmark
{rv, daniel.lucani, qz}@eng.au.dk

***Abstract*—We detail a practical compression scheme for lossless compression of time-series data, based on the emerging concept of generalized deduplication. As data is no longer stored for just archival purposes, but needs to be continuously accessed in many applications, the scheme is designed for low-cost random access to its compressed data, avoiding decompression. With this method, an arbitrary bit of the original data can be read by accessing only a few hundred bits in the worst case, several orders of magnitude fewer than state-of-the-art compression schemes. Subsequent retrieval of bits requires visiting at most a few tens of bits. A comprehensive evaluation of the compressor on eight real-life data sets from various domains is provided. The cost of this random access capability is a loss in compression ratio compared with the state-of-the-art compression schemes BZIP2 and 7z, which can be as low as 5% depending on the data set. Compared to GZIP, the proposed scheme has a better compression ratio for most of the data sets. Our method has massive potential for applications requiring frequent random accesses, as the only existing approach with comparable random access cost is to store the data without compression.**

I. Introduction

Research on data compression traditionally focuses on improving the compression ratio, with the goal of pushing it close to Shannon's entropy limit [1]. At this point, compression schemes that achieve or approach the entropy limit are well known, with some of the most typical examples being Huffman codes [2] and the universal Lempel-Ziv (LZ) codes [3], [4]. However, these codes are not sufficient for all scenarios. Importantly, they generally provide poor support for random access, meaning that all of the compressed data must be decompressed to retrieve even a single bit. This can make these codes ill-suited for the requirements of modern data storage systems, where many applications make continuous queries to the data [5]. A suitable compression scheme thus requires the data to be stored in a manner that facilitates such usage. Compression schemes that enable access to the data without complete decompression are thus desirable, since they allow the data to be stored in a compressed format while still enabling applications to make arbitrary queries to the data at a low cost [6].

At the other end of the spectrum, deduplication techniques have in recent years become a popular strategy for large-scale compression in the cloud [7], [8]. These techniques enable local encoding and decoding of files, requiring only access to a common deduplication dictionary. The principle is simple. A client stores a file by uploading it to the cloud. The cloud server checks whether this file already exists, and if it already exists it simply stores a reference to the location where the existing data is found, so no duplicate files are stored in the system. A slightly more sophisticated variant divides files into *chunks*, typically of several KBs each, and ensures that each stored chunk is unique. However, deduplication does not work in all cases and for several data types it achieves no or little compression. This limitation is caused by the fact that chunks can be deduplicated only if they are identical, so two chunks that differ in just a single bit, but are otherwise identical, will both be stored in full.

In this paper, we show how the principle of *generalized deduplication* [9] can be used to build a lossless compression system with desirable random access properties. Generalized deduplication uses a transformation function to split the data chunks into a *base* and a *deviation*, where the base is treated as the chunk in classic deduplication and is deduplicated to be stored only once. The deviation is stored without compression. The transformation function should thus be chosen so that similar chunks will have the same base and so that deviations are "small" compared to the chunk.

Our main contribution is a thorough evaluation of a practical compression scheme for time-series data [10]. It is based on generalized deduplication and provides random access to the data at a low cost. A conceptual overview is shown in Fig. 1. The scheme provides scalable lossless compression and may be implemented in large-scale cloud systems for time-series data storage, although we study only a prototype that operates on the local file system. The scheme can, with only minor modifications, even be used in an online fashion, compressing data as the storage system receives it. An exact evaluation of the random access capabilities of the scheme is provided. Since the scheme is designed for time-series data, we evaluate the scheme's compression performance on several real-world time-series data sets, with applications ranging from e-health, smart grids, and environmental monitoring, to sound recordings. Finally, the compression performance and random access is compared to state-of-the-art universal compression schemes.

II. Related work

In terms of lossless compression designed for time-series data, Sprintz [11] is a state-of-the-art algorithm developed for the Internet of Things. It leverages the temporal correlation of the data to forecast samples based on preceding values. The difference between the forecast and the actual value is then

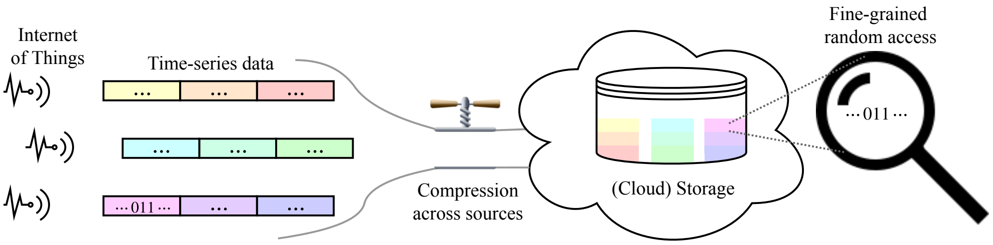

Fig. 1. A conceptual overview of the compression scheme. Data is stored in a compressed format enabling random access.

encoded, using bit-packing, run-length encoding, and entropy coding. While this method achieves both good compression ratios and high speed, it offers no advantages in terms of random access.

There have been several practical studies that report the compression performance for classic deduplication under different use cases [12], [13]. The fact that deduplication has issues with near-identical chunks has been recognized previously. If a file that was previously stored is uploaded with a minor change, e.g., a bit was inserted at the beginning, every chunk may be different from what was previously stored, since each chunk is shifted one bit. Content defined chunking [14] addresses this problem by using Rabin fingerprints [15] to adaptively determine where to create the chunk boundaries. This enables the system to identify redundancy between the file with shifted data and the original. An evaluation and comparison of fixed chunks and content defined chunks is presented in [14]. However, this does not remove the fundamental issue that deduplication is possible only if the data contains sequences that are identical. The work in [16] presents an adaptation of classic deduplication, where the authors attempt to resolve the problem of near-identical chunks being treated independently by adding strategies for detecting similar data chunks. Generalized deduplication is another proposal for dealing with this issue. It was first described in [9] to study its properties as a compression scheme. An example of generalized deduplication for an Internet of Things data model was presented in [17], where a Reed-Solomon code was used for the transformation function, showing that it is able to outperform classic deduplication for a wide range of data configurations. The permute-and-truncate transformation function was presented in [10], where the authors show that a compression scheme based on this transformation can asymptotically achieve the entropy limit for compression. The authors further provide a practical example where it is shown that a small ECG data set can be compressed to the same level that state-of-the-art compressors are able to achieve.

This idea of permuting data to make it more compressible is common in other compressors. One such example is the Burrows-Wheeler transform [18], a popular and effective strategy, notably used in the BZIP2 algorithm [19]. In Migratory Compression [20] the authors also propose a method for rearranging data and show that this preprocessing step improves the results of general purpose compressors such as GZIP [21] and 7-zip [22].

The problem of locally decodable compression has received significant attention from a theoretical point of view [5], [6], [23], [24]. These works outline the fundamental principles and limits for the compression schemes that have good random access properties. Notably, the recent work in [5] presents a generic scheme that can be used to systematically convert any universal compressor, e.g., LZ-based schemes, into a scheme that no longer has optimal compression, but instead has constant random access. A greater compromise on the compression ratio can be made to reduce the random access cost as desired.

III. Preliminaries

We let F denote an uncompressed file and F_C a compressed file. Any file can be loaded as a single binary vector. Vector notation is used for chunks of data. We will denote vectors with small boldface letters and their entries with subscripts, i.e., $\boldsymbol{v} = [v_0, v_1, \ldots, v_{\ell(\boldsymbol{v})-1}]$, where $\ell(\boldsymbol{v})$ is the number of entries in $\boldsymbol{v}$. We will use the notation $\boldsymbol{v}_a^b = [v_a, v_{a+1}, \ldots, v_b]$ to denote part of a vector. Letters from the Latin alphabet will denote constant values, whereas Greek letters will be used for functions. $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ denotes the ceiling and floor functions, returning the first integer not smaller than or not greater than, respectively. Sets are represented with big calligraphic letters and $|\cdot|$ denotes set cardinality, the number of elements in the set. Data chunks are denoted $\boldsymbol{z} \in \mathcal{Z}$, where $\mathcal{Z}$ is the set of all possible chunks. In this paper, we will restrict the chunks to be in the finite field $\mathbb{F}_2^n$, which is the set of binary vectors with length n. Generalized deduplication thus operates with a fixed chunk length. We will be forming chunks by concatenating c subsequent samples of time-series data. When multiple vectors associated with a single file, F, are used in the same context, we will discern them by subscripts, e.g., $\boldsymbol{z}_1$, $\boldsymbol{z}_2$. If they are associated with different files $F^{(1)}$, $F^{(2)}$, this will be denoted by their superscripts, i.e.., $\boldsymbol{z}^{(1)}$, $\boldsymbol{z}^{(2)}$.

Generalized deduplication utilizes a transformation function $\phi : \mathcal{Z} \to \mathcal{X} \times \mathcal{Y}$, that takes a file chunk $\boldsymbol{z} \in \mathcal{Z}$ and transforms it to a pair $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{X} \times \mathcal{Y}$. $\boldsymbol{x}$ is called the *base* of $\boldsymbol{z}$ and $\boldsymbol{y}$ is called the *deviation*. To be a valid transformation function, it must be injective, which implies that we can specify an inverse function $\phi^{-1} : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ such that

$$\phi^{-1}(\phi(\boldsymbol{z})) = \boldsymbol{z}, \ \forall \boldsymbol{z} \in \mathcal{Z}. \tag{1}$$

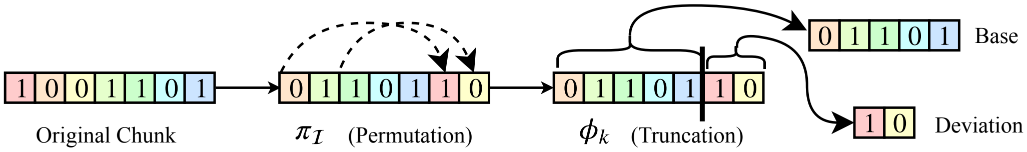

Fig. 2. Using the permute-and-truncate strategy to transform a chunk to base and deviation.

This property must be satisfied to let generalized deduplication be used as a lossless compression scheme.

The storage process is similar to classic deduplication. When a client uploads a file, it is divided into chunks. Now, rather than immediately storing the chunks, the transformation function is applied to each chunk to obtain the corresponding pair of base and deviation. The server applies deduplication to the bases, ensuring that each base is stored only once. The deviation is stored in full alongside the ID of the base. This clearly allows perfect reconstruction of the original chunk.

Generalized deduplication achieves a compression gain if the chosen transformation function transforms several similar chunks to a single base and different (small) deviations. The optimal transformation function will depend on the structure of the data. We will consider a simple transformation function that splits the data in two parts as

$$\phi_k(\boldsymbol{z}) \triangleq (\boldsymbol{z}_0^{k-1}, \boldsymbol{z}_k^{n-1}), \tag{2}$$

where the first k bits of $\boldsymbol{z}$ will be used as a base and the remaining $n - k$ bits will be the deviation. With this transformation function and the fact that $\boldsymbol{z} \in \mathbb{F}_2^n$, we have that $\boldsymbol{x} \in \mathbb{F}_2^k$ and $\boldsymbol{y} \in \mathbb{F}_2^{n-k}$. The transformation is clearly injective and the inverse is a concatenation, denoted as

$$\kappa(\boldsymbol{x}, \boldsymbol{y}) \triangleq [x_0, x_1, \ldots, x_{\ell(\boldsymbol{x})-1}, y_0, y_1, \ldots, y_{\ell(\boldsymbol{y})-1}]. \tag{3}$$

It is immediate that (1) holds with $\phi = \phi_k, \phi^{-1} = \kappa$, so the transform is valid for lossless compression. The transformation function of Eq. (2) does not necessarily cause bases to match. To increase the number of matches, we will employ a permutation $\pi_{\mathcal{I}}$ that moves the indices in the set $\mathcal{I} = \{i_1, i_2, \ldots, i_{n-k}\}$ to the end of a vector, i.e.,

$$\pi_{\mathcal{I}}(\boldsymbol{z}) \triangleq [\ldots, z_{i_1-1}, z_{i_1+1}, \ldots, z_{i_2-1}, z_{i_2+1}, \ldots, z_{i_1}, z_{i_2}, \ldots, z_{i_{n-k}}].$$

This permutation is easily reversed by moving the last $n - k$ elements into their appropriate positions, as specified by $\mathcal{I}$. The complete workflow of transforming a chunk to its base and deviation is shown in Fig. 2

Remark. Classic deduplication is a special case where the transformation function ϕ_n results in $\boldsymbol{x} = \boldsymbol{z}$ ($k = n$) and $\boldsymbol{y}$ is empty.

IV. The compression scheme

We now detail the compression scheme. This is done in three parts: Compression, decompression, and a discussion of the compressed size.

A. *Compression*

The compression procedure consists of two mandatory and one optional step. Each step is detailed in the following.

1) Preprocessing: The intent of this stage is to determine the key parameters for the compression. This will be done using training data at the beginning of the data, e.g., the first file in a data set. Clearly, the training data must be somewhat representative of the data's structure. We assume that the first file contains N_1 samples, which will be used for our parameter estimation. If $N_1 > 10^6$ samples, we limit it to this amount to keep the preprocessing stage computationally tractable. Each sample consists of n_s bits.

The first key parameter is the number of samples to concatenate to form each data chunk, denoted by c. The optimal amount of samples to concatenate will depend on how much correlation there is between adjacent samples. The ID of the bases will use fewer bits per sample than would be the case without concatenation, reducing the overhead associated with their representation. On the other hand, concatenating too many samples reduces the probability of matching bases, as they need to be equal in a larger number of bits, so it is likely that there will be fewer matches overall. Further, when each chunk is formed from a larger number of samples, a fixed number of samples will result in a smaller number of chunks. However, if adjacent samples are highly correlated, many chunks will still have matching bases after the concatenation. The right number of samples to concatenate is thus clearly dependent on the data set and should be inferred from it. To infer what this parameter, c, should be, we propose the heuristic in Algorithm 1 based on our experience with the system. The procedure is a iterative, where larger and larger amounts of samples are concatenated until almost all chunks are unique. A small fraction, d, of duplicate chunks are allowed. We use $d = 0.01$ for our experiments. When the number of concatenated samples per chunk has been determined to be c, the concatenated chunk length becomes $n \triangleq cn_s$ bits.

Next, the number of bits to use for the deviation must be determined, i.e., the value of k for the transformation function ϕ_k in Eq. (2). To maximize the number of matching bases, we want the deviation to contain the bits that are least correlated with the rest of the chunk. Ideally, we would capture this by estimating the mutual information between sets of bits, but estimating this value reliably is not always easy [25]. Instead, we use the training data to find the Pearson correlation between pairs of bits. Although this only captures linear and pairwise

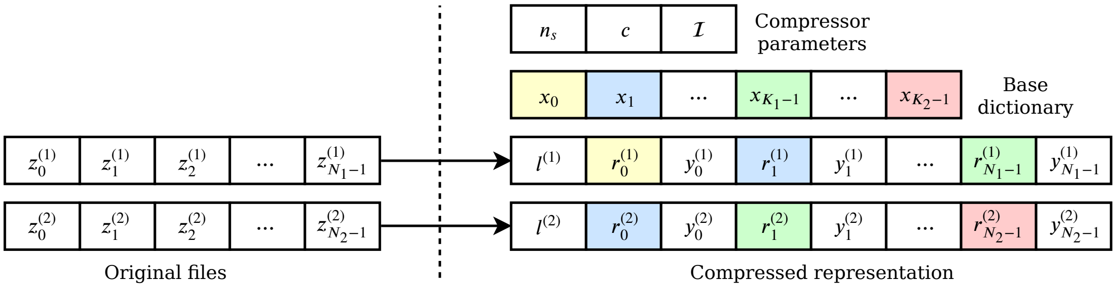

Fig. 3. Visualization of the original and compressed representation of two files. An ID's color shows which base it refers to. (Blocks are not to scale.)

```
Algorithm 1 Determining number of samples to concatenate
Input: Training file with $N_T$ samples of $n_s$ bits, parameter $d$.
Output: Number of samples to concatenate, $c$.
$c \leftarrow 0$, $U \leftarrow 0$                        //Initialize values
while $U < (1-d)N_c$ do
    $c \leftarrow c + 1$                 //Try next value of concatenations
    $N_c \leftarrow N_T/c$               //Calculate new number of chunks
    Reshape to $N_c$ chunks of $cn_s$ bits   //Construct chunks
    $U \leftarrow$ Number of unique chunks        //Count unique
end while
return $c$
```

correlations, it captures much of the same information as the mutual information does [25]. Having estimated the correlation matrix, the bits of the chunk is sorted in a descending order according to the maximum absolute correlation. Ignoring overhead and storage of parameters, the cost of storing the training data with bases of size k is

$$S_k \triangleq Kk + N(\lceil \log_2 K \rceil + (n-k)), \tag{4}$$

where K is the number of unique bases and N is the number of chunks. A more precise storage cost is given in Section IV-C. This cost is first found where the entire chunk is considered the base, i.e., $k = n$, by counting the number of unique bases and evaluating S_n. We then remove the least correlated bit, which the previous sorting of the chunk conveniently has caused to be located at the end. This changes the base length to $n-1$. The number of unique bases is counted anew, and the cost of storing the training data with the new base length, S_{n-1}, is found. This process is repeated, until termination when $S_{n-i} \leq S_{n-i-1}$, i.e., until the first local minimum of the storage cost for the training data is found. We then have the base length $k = n - i$ and the deviation length $n - k = i$. Finally, the original indices of the $n-k$ bits with the smallest maximum absolute correlation are collected to form the set $\mathcal{I}$, which specifies which bits the compressor will use as a deviation. To conclude, the values n_s, c and the set $\mathcal{I}$ are stored, as shown in Fig. 3, to allow for later decompression.

Remark. If the event $S_n < S_{n-1}$ occurs, the generalized deduplication approach performs as the special case of classic deduplication. We regard this event as a *fall back* to classic deduplication and use $n = k = n_s$, i.e., $c = 1$, $n - k = 0$.

2) Estimation of base usage statistics (OPTIONAL): This step involves going through all files to count the number of times that each base is used. This allows us to estimate the base usage distribution by calculating the frequency of each base. This can be used to define an entropy encoding scheme for the base IDs, i.e., encoding frequently used bases with fewer bits [26]. This can further improve the overall compression, as less storage is needed for the base IDs. However, it will impact the overall compression speed and compromise random access performance, so this step should only be used if the overall compression rate is more important than the random access performance. In general, this step improves the compression the most if the usage distribution of bases is far from uniform, since heavily used bases will get a short ID and infrequent bases will get a long ID. If the distribution is uniform, no gain is possible. In order to decode, the assigned codes for each base also need to be stored. Huffman coding [2] could be used, letting the Huffman table be compactly stored in a manner similar to what is done in the DEFLATE algorithm [27].

3) File compression: The final step is to actually perform the compression of files. The files are compressed one by one, with a single pass over each file. The process is detailed in Algorithm 2. Each file is processed in a stream fashion, from the beginning to the end, one chunk at a time. Zero padding is used to form the final chunk if the file cannot be divided into an integer number of chunks. For each chunk, the base and deviations are first identified, as shown in Fig. 2. The ID of that particular base is then found. If the base does not already exist, it is first added to the base dictionary, $\mathcal{D}$, and assigned the next available integer, starting from 0, which will serve as its ID. Thus, the first base gets the integer 0, the second gets 1, and so forth. Once the ID has been determined and the dictionary is updated, the next chunk is processed. After all chunks in the original file have been processed, we determine how many bits are needed to represent the base IDs. The base IDs can be represented with

$$l^{(i)} \triangleq \lceil \log_2 K^{(i)} \rceil \tag{5}$$

bits, where $K^{(i)}$ is the number of bases after the bases of file i has been added to the dictionary. Finally, the compressed output is generated. It starts by having an Elias-gamma encoding [28] of $l^{(i)}$, which will be required to decode. It then follows an alternating structure, where for each chunk the

Algorithm 2 Compression of a single file

```
Input: $F^{(i)}$ to compress.
Compressor state: $n$, $k$ $\mathcal{I}$, $\mathcal{D}$ are initialized.
Output: Compressed file $F_C^{(i)}$
$\boldsymbol{f} \leftarrow$ Data from file $F^{(i)}$
$N_c \leftarrow \ell(\boldsymbol{f})/n$    //The number of chunks in file
for $j = 0, 1, \dots, N_c - 1$ do
    $\boldsymbol{z}_j \leftarrow \boldsymbol{f}_{jn}^{(j+1)n-1}$    //Extract chunk
    $\boldsymbol{z}'_j \leftarrow \pi_{\mathcal{I}}(\boldsymbol{z}_j)$    //Permute
    $(\boldsymbol{x}_j, \boldsymbol{y}_j) \leftarrow \phi_k(\boldsymbol{z}'_j)$    //Separate base and deviation
    if $x_j \notin \mathcal{D}$ then
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{\boldsymbol{x}_j\}$    //Add if not duplicate
    end if
    $r_j \leftarrow$ ID for the base $\boldsymbol{x}_j$    //Get the ID
end for
$K \leftarrow |\mathcal{D}|$    //Find the number of bases
$l \leftarrow \lceil \log_2 K \rceil$    //Find number of bits for IDs
$\boldsymbol{l} \leftarrow$ Elias-Gamma($l$)    //Binary encoding of $l$
$\boldsymbol{f}_C \leftarrow \boldsymbol{l}$    //Initialize output vector
for $j = 0, 1, \dots, n_c - 1$ do
    $\boldsymbol{r}_j \leftarrow \mathrm{Bin}(r_j, l)$    //ID's $l$-bit binary representation
    $\boldsymbol{f}_C \leftarrow \kappa(\boldsymbol{f}_C, \boldsymbol{r}_j)$    //Concatenate base ID to output
    $\boldsymbol{f}_C \leftarrow \kappa(\boldsymbol{f}_C, \boldsymbol{y}_j)$    //Concatenate deviation to output
end for
Store compressed data $\boldsymbol{f}_C$ in $F_C^{(i)}$
return $F_C^{(i)}$
```

$l^{(i)}$-bit base ID $\boldsymbol{r}_j^{(i)}$ is followed by the $(n-k)$-bit deviation, $\boldsymbol{y}_j^{(i)}$. The procedure for compressing a single file is shown in detail in Algorithm 2. The dictionary $\mathcal{D}$ must also be stored. It is stored such that the first k bits of the file contains the base with ID 0, the following k bits contains the base with ID 1, and so on. Thus, the compressed structure of the data stored in the system becomes as visualized in Fig. 3.

Remark. To use the scheme in an online fashion, the dictionary can be stored on the fly while data is being received. There are two simple ways to achieve this. One is to periodically break the stream into files, which are handled separately. The other is to set the length of the IDs as a global parameter, so that IDs can be saved immediately when they are identified.

B. Decompression

Decompression is a two-step procedure. First, the compressor state is restored. Then, files can be decoded.

1) Restoring compressor state: The first step is to restore the final state of the compressor. That is, we start by loading the parameters stored during the preprocessing stage, n_s, c, and $\mathcal{I}$. The chunk length n must then be cn_s bits. We can also immediately infer that $n-k = |\mathcal{I}|$, implying that $k = n - |\mathcal{I}|$. As a final step, the dictionary $\mathcal{D}$ is prepared. The organization of the bases is known, and the j-th k-bit chunk in the dictionary file corresponds to the base with ID $j-1$. Either the dictionary is loaded into memory in its entirety or the desired bases are retrieved directly from the dictionary file when required during the procedure.

Algorithm 3 Decompression of a single file

```
Input: Compressed file $F_C^{(i)}$ to decompress.
Compressor state: $n$, $k$ $\mathcal{I}$, $\mathcal{D}$ are initialized.
Output: Decompressed file $F^{(i)}$
$\boldsymbol{f} \leftarrow []$    //Initialize empty output
$\boldsymbol{f}_C \leftarrow$ Data from file $F_C^{(i)}$
$m \leftarrow$ Number of zeros the file starts with
$\boldsymbol{l} \leftarrow (\boldsymbol{f}_C)_{m+1}^{2m+1}$    //Extract binary representation
$l \leftarrow \mathrm{Int}(\boldsymbol{l})$    //Convert binary vector to integer
$\boldsymbol{f}_C \leftarrow (\boldsymbol{f}_C)_{2m+2}^{\ell(\boldsymbol{f}_C)-1}$    //Discard Elias-gamma encoding of $l$
$n_c = l + n - k$    //Encoded length of chunks
$N_c \leftarrow \ell(\boldsymbol{f}_C)/n_c$    //The number of data chunks
for $j = 0, 1, \dots, N_c - 1$ do
    $\boldsymbol{r}_j \leftarrow (\boldsymbol{f}_C)_{jn_c}^{jn_c+l-1}$    //Extract $l$-bit base ID
    $r_j \leftarrow \mathrm{Int}\left(\boldsymbol{r}_j\right)$    //Convert base ID to integer
    $\boldsymbol{x}_j \leftarrow \mathcal{D}[r_j]$    //Get base from the dictionary
    $\boldsymbol{y}_j \leftarrow (\boldsymbol{f}_C)_{jn_c+l}^{(j+1)n_c-1}$    //Extract $(n-k)$-bit deviation
    $\boldsymbol{z}'_j \leftarrow \kappa(\boldsymbol{x}_j, \boldsymbol{y}_j)$    //Concatenate base and deviation
    $\boldsymbol{z}_j \leftarrow \pi_{\mathcal{I}}^{-1}(\boldsymbol{z}'_j)$    //Reverse the permutation
    $\boldsymbol{f} \leftarrow \kappa(\boldsymbol{f}, \boldsymbol{z}_j)$    //Concatenate reconstructed chunk
end for
Store decompressed data $\boldsymbol{f}$ in $F^{(i)}$
return $F^{(i)}$
```

2) Decompressing files: Files are decompressed one by one. All files start with an Elias-gamma encoding [28] of the number of bits used for the base representation, $l^{(i)}$. Although this encoding has variable length, it is easily decoded by first counting the number of consecutive zeros, m. The following $m+1$ bits are then the binary encoding of $l^{(i)}$. Once that value is known, the decoder knows that the compressed sequence contains pairs of $l^{(i)}$-bit IDs of bases and $(n-k)$-bit deviations. With knowledge of $\mathcal{I}$, this is enough to enable perfect reconstruction of the original file. The procedure is shown in detail in Algorithm 3.

C. Compressed size

The size of the compressed data is now evaluated. We choose to use the Elias-gamma code [28] for representing integers, since it is a simple and effective prefix-free code. Under this code, any integer a will be represented with

$$\gamma(a) \triangleq 2\lfloor \log_2 a \rfloor + 1 \tag{6}$$

bits. The parameters required for restoring the compressor state is the number of bits per sample, n_s, the number of samples per chunk, c, and the set of indices for bits going to the deviation, $\mathcal{I}$. The total number of bits required for storing these parameters is

$$\begin{aligned} S_{\text{params}} &\triangleq \gamma(n_s) + \gamma(c) + \sum_{i \in \mathcal{I}} \gamma(i) \\ &\leq \gamma(n_s) + \gamma(c) + |\mathcal{I}|\gamma(n) \\ &= \gamma(n_s) + \gamma(c) + (n-k)\gamma(n). \end{aligned} \tag{7}$$

The compressed size of an individual file, $F^{(i)}$, is related to the number of chunks in the original file, N_i, and the length of base identifiers in the compressed representation, $l^{(i)}$, which we recall is derived from the number of bases in the dictionary at the time of compression, K_i, by a logarithm. Thus, the number of bits in the compressed representation is

$$S_{F^{(i)}} \triangleq \gamma(l^{(i)}) + N_i(l^{(i)} + (n - k)), \tag{8}$$

where the first term is the cost of representing $l^{(i)}$, the size of base IDs used in the file, and the second term is the compressed representation of each chunk in the original file. Finally, the cost of representing the dictionary of bases, $\mathcal{D}$, must be included. The total number of bits used for representing the dictionary, which contains $K = \max_i K_i$ bases of k bits each, is

$$S_{\mathcal{D}} \triangleq |\mathcal{D}|k = Kk. \tag{9}$$

Jointly, these three expressions specify the total compressed size in bits,

$$S_{\text{tot}} \triangleq S_{\text{params}} + S_{\mathcal{D}} + \sum_i S_{F^{(i)}}. \tag{10}$$

For interesting data sets the cost of the parameters in Eq. (7) is negligible, and the real challenge is in choosing the key parameters that optimize the trade off between Eqs. (8) and (9), which is the goal of the heuristics presented in Section IV-A. If too many chunks are concatenated or if the deviation is chosen to be too small, then there will be no or few base matches. This will cause the dictionary to be so large, and contain so many bases, that no compression is achieved, since the dictionary itself will essentially be the size of the original data. On the other hand, if too few samples are concatenated or if the deviations are large compared to the base, then the ID of the base and its deviation will essentially combine to be the same size as the original chunk, reducing the possibility of compressing the data.

V. Random Access

The generalized deduplication compression scheme presented in the previous section allows low cost random access to the data. In this section, we detail procedures for accessing arbitrary locations of the data and the cost thereof. The compressor state must be partially restored in the first of a series of accesses to the data. That is, the parameters stored during the preprocessing stage, n_s, c, and $\mathcal{I}$, are loaded. n and k are calculated from these, since they are vital for making sense of the compressed representation. The number of bits that must be accessed for restoring these values is exactly S_{params} bits, as given by Eq. (7). This value is independent of the amount of data stored in the system. Costs stated in this section mostly assume the worst-case scenario, where only one access is made to the data. The overhead will clearly be smaller for subsequent accesses, since state from a previous access can be reused.

A. Accessing chunks

The process for accessing any specific chunk in its entirety is given in this section. Assume that the desired chunk is $z_j^{(i)}$, which we recall is the j-th chunk of file $F^{(i)}$. The first step is to load $l^{(i)}$, the number of bits used for representing the base IDs in the compressed file $F_C^{(i)}$. The Elias-gamma encoding of the value is located at the beginning of $F_C^{(i)}$ and takes up $\gamma(l^{(i)})$ bits, all of which must be accessed to decode the value. Each chunk then has a compressed representation using exactly $l^{(i)} + (n - k)$ bits, and chunks appear in the same order as in the original file. Thus, chunk j starts at bit

$$\alpha(j) \triangleq \gamma(l^{(i)}) + j(l^{(i)} + (n - k)), \tag{11}$$

and ends at bit

$$\begin{aligned}\beta(j) &\triangleq \alpha(j + 1) - 1\\ &= \gamma(l^{(i)}) + (j + 1)(l^{(i)} + (n - k)) - 1.\end{aligned} \tag{12}$$

These $l^{(i)} + n - k$ bits must be accessed. The reference to the base of the desired chunk, $r_j^{(i)}$, is defined by the first $l^{(i)}$ bits and the remaining $n - k$ bits are the deviation, $\boldsymbol{y}_j^{(i)}$. The next step is to retrieve $\boldsymbol{x}_j^{(i)}$, which is the base with ID $r_j^{(i)}$, from the dictionary. It occupies the k bits starting from bit $kr_j^{(i)}$, ending at $k(r_j^{(i)} + 1) - 1$, since bases are stored sequentially according to their IDs, with the first ID being 0. This involves accessing a total of k bits. With $\boldsymbol{x}_j^{(i)}$ and $\boldsymbol{y}_j^{(i)}$ now available, we may reconstruct the desired chunk by inserting the deviation bits at the appropriate locations, as specified by $\mathcal{I}$.

In total, the number of bits that must be accessed to retrieve a chunk is

$$\begin{aligned}t_{\text{chunk}} &\triangleq S_{\text{params}} + \gamma(l^{(i)}) + l^{(i)} + (n - k) + k\\ &= S_{\text{params}} + \gamma(l^{(i)}) + l^{(i)} + n,\end{aligned} \tag{13}$$

which is an additional $S_{\text{params}} + \gamma(l^{(i)}) + l^{(i)}$ bits compared to accessing the n bits from an uncompressed file. When the compressor state is already in place, the overhead is only $\gamma(l^{(i)}) + l^{(i)}$ bits.

B. Accessing individual bits

The compressed format also allows for more fine-grained random access patterns. It is possible to access any specific bit from the original data. The cost of this access depends on whether the bit is encoded as part of a chunk's base or deviation. Both scenarios are now evaluated. Assume that we want to know the value of bit a in the i-th original file, $F^{(i)}$. As with accessing an entire chunk, it is necessary to first load $l^{(i)}$, the number of bits used for representing the base IDs in the compressed file $F_C^{(i)}$. This still requires accessing $\gamma(l^{(i)})$ bits, the Elias-gamma encoding of the value. The compressed size of each chunk in the file, $l^{(i)} + (n - k)$ bits is now known. In the original file, bit a belongs to the $\lfloor a/n \rfloor$-th chunk, $z_{\lfloor a/n \rfloor}^{(i)}$, which must be located in the compressed file. It is found at the same position as in the previous section, starting in $\alpha(\lfloor a/n \rfloor)$ and ending at $\beta(\lfloor a/n \rfloor)$, as defined in Eqs. (11) and (12). Instead

TABLE I
The time-series data sets that we use in this article

Abbrev.	Dataset	Size	Reference
ECG0	MIT-BIH Arrhythmia Database, ECG (Lead 1 of 2)	46.8 MB	[29], [30]
ECG1	MIT-BIH Arrhythmia Database, ECG (Lead 2 of 2)	46.8 MB	[29], [30]
Pow	Power grid total load for Germany, day ahead forecast and actual	1.1 MB	[31], [32]
T-DK	Daily maximum temperature at 8 weather stations in Denmark	843 KB	[33], [34]
T-EU	Daily maximum temperature at 100 weather stations across Europe	9.7 MB	[33], [34]
Mic	CMU-MMAC Database, microphone recording	36.4 MB	[35], [36]
Acc	CMU-MMAC Database, accelerometer data (single axis)	872 KB	[35], [36]
Elec	Electricity load for 10 households	5.6 MB	[37]

of loading the entire chunk, as in the previous section, we determine where bit a is located in this chunk. In the original chunk, bit a was at location $(a \mod n)$. The next step depends on whether the bit is used as part of the deviation or base in the compression.

1) Accessing a deviation bit: If $(a \mod n) \in \mathcal{I}$, then the bit is stored in the deviation. The order of the elements in $\mathcal{I}$ reveals bit a's location. If $(a \mod n)$ is the j-th element of $\mathcal{I}$, then the bit of interest can likewise be found as the j-th element of the deviation in the compressed chunk. This means that the bit is located exactly at position $\alpha(\lfloor a/n \rfloor) + l^{(i)} + j$ and can be immediately retrieved. In the worst case, the total number of bits that must be accessed to retrieve the bit is

$$t_{\text{bit, deviation}} \triangleq S_{\text{params}} + \gamma(l^{(i)}) + 1. \tag{14}$$

2) Accessing a base bit: If $(a \mod n) \notin \mathcal{I}$, then the bit is compressed as part of a base. To determine the bit's value, the first step is to access the $l^{(i)}$ bits at the beginning of the chunk to find the value of the ID, $r^{(i)}_{\lfloor a/n \rfloor}$. The ID reveals which base is associated with that particular chunk. The base with identifier $r^{(i)}_{\lfloor a/n \rfloor}$ is the k bits located from bit $kr^{(i)}_{\lfloor a/n \rfloor}$ to bit $k(r^{(i)}_{\lfloor a/n \rfloor} + 1) - 1$ in the file containing the bases. To find the exact location, we cannot just access one specific bit yet, since some of the bits in the original chunk were moved to the end. To finally locate the bit of interest, we determine how many places it has moved due to the permutation:

$$\delta_{\mathcal{I}}(a) \triangleq \sum_{i \in \mathcal{I}} \mathbf{1}\{i < (a \mod n)\}, \tag{15}$$

where $\mathbf{1}\{\cdot\}$ denotes the indicator function, taking the value of 1 if the statement is true and 0 otherwise. It is now known that the bit of interest is the bit located at position $kr^{(i)}_{\lceil a/n \rceil} + (a \mod n) - \delta_{\mathcal{I}}(a)$, so it can be retrieved. In total, the number of bits that must be accessed to retrieve the bit is

$$t_{\text{bit, base}} \triangleq S_{\text{params}} + \gamma(l^{(i)}) + l^{(i)} + 1, \tag{16}$$

i.e., an additional $l^{(i)}$ bits compared to accessing a bit that is located in the deviation.

VI. Performance evaluation

We compare the performance of the generalized deduplication-based compression scheme to three popular, high quality universal compression algorithms. The workings of each are detailed in [38]. The algorithms are:

- GZIP [21], an implementation of the DEFLATE algorithm [27], which uses LZ77 [3] and Huffman coding [2].
- BZIP2 [19], an implementation of the Burrows-Wheeler transform [18].
- 7-zip (7z) [22], which uses LZMA, an extension of LZ77.

We evaluate the compression performance of our scheme on a number of signals from time-series data sets. The data sets are briefly described in Table I. Before we compress, we extract the raw signal of interest from each data file in the data set and pack it into a binary file with a small header. The sizes listed in Table I reflect the cumulative sizes of the binary files generated from the data set. To evaluate our compression method, we operate on these binary files. The general purpose compression methods are supplied a folder containing the binary files, allowing them to jointly compress the files. Our compressor based on generalized deduplication receives the binary files one by one, using the first file supplied to it as the training data for estimating the parameters. The parameter $d = 0.01$ is chosen for Algorithm 1. We implemented the scheme in Python and did not optimize for speed. Thus, we will not compare our compression and decompression speed with that of the general purpose methods. Instead, we focus on (i) the compression ratio and (ii) the random access cost as quality indicators for the compression. The compression ratio is defined as

$$\text{CR} \triangleq \frac{\text{compressed size}}{\text{uncompressed size}}, \tag{17}$$

which is desired small. The sizes of all the binary files in a data set are aggregated to get the real uncompressed size, shown in Table I. The compressed size includes all files stored by the compressor. This provides a real measurement of the total storage cost, S_{tot}, given in Eq. (10), including all overhead incurred.

A. Compression of several data sets

The results of applying our compressor to the data sets listed in Table I is now studied. In Section IV-A it is mentioned that the compression ratio may be improved if entropy encoding of the base IDs is used. To estimate the effect of performing this step, the number of times that each base is used in the compressor is logged. This statistic allows calculation of how this step could improve the compression performance. The simple Shannon-Fano-Elias entropy code [26] is used as an example encoding. With this code, $\lceil \log_2 1/f_i \rceil$ bits are used

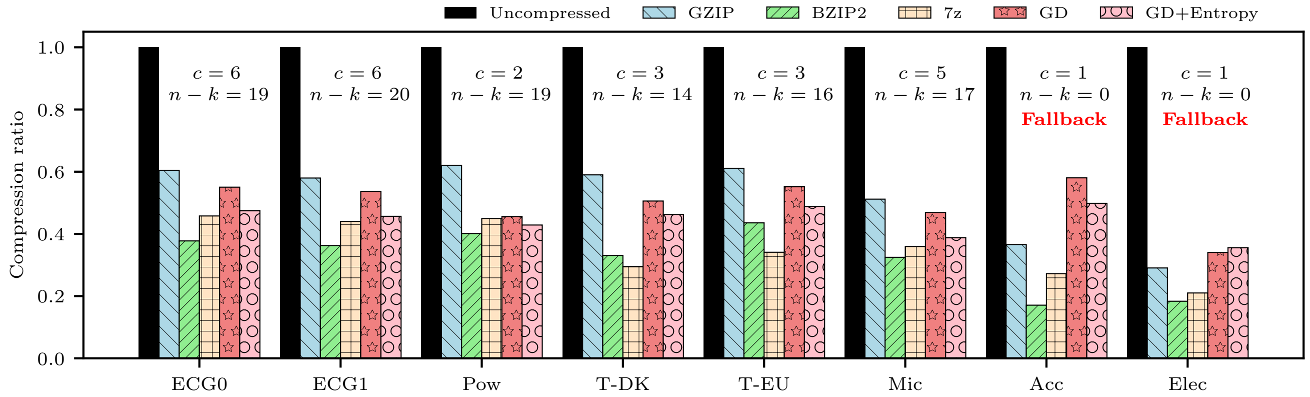

Fig. 4. Comparison of compression performance on several data sets. The generalized deduplication configuration selected by the heuristic is indicated.

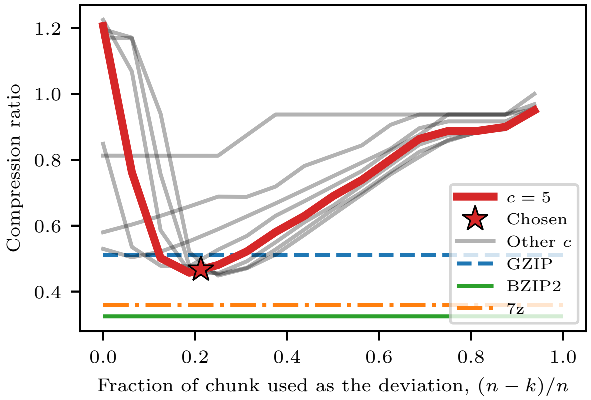

Fig. 5. Comparison of heuristically chosen parameters $c = 5, n - k = 17$ for Mic data set

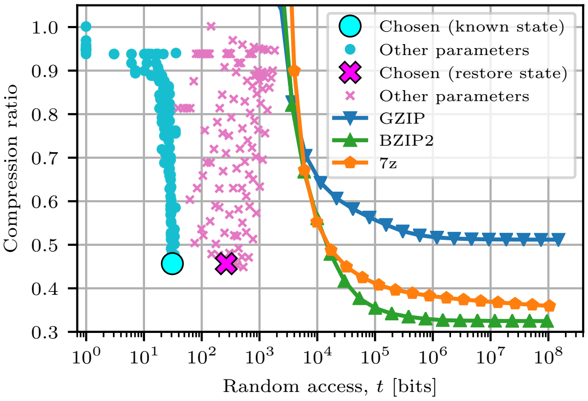

Fig. 6. Trade-off between compression ratio and cost of accessing a single bit in Mic data set.

to refer to a base with frequency f_i. The compression ratios achieved by the three general purpose algorithms and our method, with both the basic and the entropy-based encoding of the base IDs, are compared in Fig. 4. The figure also states the value of parameters c and $n - k$ found using the training data. For the first six of the eight signals, our method achieves a smaller compression ratio than what GZIP achieves. Both BZIP2 and 7z are able to achieve a better compression for all the data sets. If entropy encoding of IDs is employed, the gap between our method and BZIP2 and 7z is reduced, as expected. For the remaining two data sets, `Acc` and `Elec`, the parameter heuristic falls back to classic deduplication with no concatenation. This is indeed the best possible configuration for our compressor on these signals, but the general purpose methods all achieve a better compression.

B. The parameter choice

Since the parameters of the compressor were chosen based on a training data set according to the heuristic approach described in Section IV-A, the chosen parameters are unlikely to be the optimal configuration. Fig. 5 shows the compression ratio achieved on the full `Mic` signal with parameters $c \in \{1, 2, \ldots, 8\}$ and varying deviation lengths. The configuration chosen by the heuristics is $c = 5$, $n - k = 17$, as indicated by the star. It is seen that the chosen configuration is very close to the global minimum, validating that the heuristic approach is reasonable. The performance for other data sets is similar and the achieved compression is close to the global minimum.

C. Random access

An obvious advantage of our scheme is that it makes random access to the data less costly than for the general purpose compression schemes. To reduce the random access cost of general purpose compressors, one approach is to split the files and then perform the compression on these smaller parts, trading off compression performance for random access cost [5], since this allows a bit to be accessed by decompressing only a single part instead of the entire file. For the general purpose methods, this is how we can provide lower random access cost. Random access in our scheme is performed as described in Section V. The trade-off between compression ratio and the number of bits that must be accessed to retrieve a single bit is visualized in Fig. 6 for the `Mic` data set. The figure shows the cost of retrieving a single base bit both in the worst case, where the compressor state must be restored, and in the event the decompressor knows the parameters already. If the state must be restored, the number of bits to be accessed is at most $t_{\text{bit, base}}$, as given by (16). The cost of accessing subsequent bits can be much lower for both our method and general purpose methods, depending on the access pattern. With the heuristically chosen parameters for this particular data set, up to 266 bits must be accessed to retrieve any single bit of the uncompressed data. Most of this cost is associated with restoring the compressor state, which involves accessing as much as 235 bits, as detailed in Eq. (7). Thus, if the compressor state is already restored, any individual bit can be retrieved by accessing 31 bits if it is a base bit or 10 bits if it is a deviation bit. Compared to BZIP2 and 7z, our scheme thus allows random access to an arbitrary bit with approximately two orders of magnitude fewer bit accesses, while having a similar compression ratio. Our scheme outperforms GZIP on both compression ratio and random access capabilities. Thus, our scheme is very effective to compress the data if there are strict random access requirements. The trade-off between compression ratio and random access is similar for the other data sets, and our method is in general able to allow random access at a much smaller cost. This advantage comes at the expense of slight performance loss in the compression ratio compared to state-of-the-art lossless compressors, as Fig. 4 shows. This trade-off will be acceptable if low random access cost is important, since the only other way to provide comparable random access to our method is to store the data in an uncompressed format, which requires much larger amounts of storage space.

VII. Conclusion

A lossless compressor for time-series data utilizing the concept of generalized deduplication has been detailed and evaluated. First, we detailed the compressor in terms of compression and reconstruction algorithms and we further defined a simple heuristic for selecting an appropriate parameter configuration based on a training data set. This heuristic approach is found to achieve a compression close to the global optimum among all configurations. Second, a comprehensive evaluation of the method is provided, where the method is applied to eight different time-series signals from various domains. The achieved compression is compared to the general purpose compressors GZIP, BZIP2, and 7z. In six out of these eight data sets, our compressor achieves a better compression than GZIP, but not quite as low as BZIP2 and 7z. The remaining two data sets cause the compressor based on generalized deduplication to operate at the special case of classic deduplication, which does not reach the compression ratio of the general purpose algorithms. However, the value of our method is evident when random access capabilities are taken into account. While BZIP2 and 7z can achieve the lowest compression ratio, they also make access to the data difficult, requiring the entire chunk to be decompressed to access even a single bit. To access any bit compressed with our compressor, the number of bits to be accessed can be orders of magnitude fewer than would be required if the data was compressed with BZIP2 or 7z. The proposed method thus provides a good trade-off between achieving a good compression ratio and low random access cost. This makes the scheme well suited for storage of large amounts of time-series data that frequently needs to be accessed and/or if access to storage is costly and needs to be minimized.

Acknowledgments

This work was partially financed by the SCALE-IoT project (Grant No. DFF - 7026-00042B) granted by the Danish Council for Independent Research, the Aarhus Universitets Forskningsfond Starting Grant Project AUFF-2017-FLS-7-1, and Aarhus University's DIGIT Centre.

References

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, jul 1948.

[2] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, sep 1952.

[3] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, may 1977.

[4] ——, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. Inf. Theory*, vol. 24, no. 5, pp. 530–536, sep 1978.

[5] K. Tatwawadi, S. S. Bidokhti, and T. Weissman, "On Universal Compression with Constant Random Access," in *IEEE ISIT*, jun 2018, pp. 891–895.

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

[7] W. Xia, H. Jiang, D. Feng *et al.*, "A Comprehensive Study of the Past, Present, and Future of Data Deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[8] R. Kaur, I. Chana, and J. Bhattacharya, "Data deduplication techniques for efficient cloud storage management: a systematic review," *J. Supercomputing*, vol. 74, no. 5, pp. 2035–2085, may 2018.

[9] R. Vestergaard, Q. Zhang, and D. E. Lucani, "Generalized Deduplication: Bounds, Convergence, and Asymptotic Properties," in *IEEE GLOBECOM*, dec 2019.

[10] ——, "Lossless Compression of Time Series Data with Generalized Deduplication," in *IEEE GLOBECOM*, dec 2019.

[11] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time Series Compression for the Internet of Things," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 3, sep 2018.

[12] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Trans. Storage*, vol. 7, no. 4, pp. 1–20, 2012.

[13] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary Data Deduplication—Large Scale Study and System Design," in *USENIX ATC*, Boston, MA, 2012, pp. 285–296.

[14] C. Policroniades and I. Pratt, "Alternatives for Detecting Redundancy in Storage Systems Data," in *USENIX Annual Technical Conference*. Boston, MA: USENIX, 2004.

[15] M. O. Rabin, "Fingerprinting by random polynomials," Tech. Rep., 1981.

[16] W. Xia, H. Jiang, D. Feng, and L. Tian, "Combining Deduplication and Delta Compression to Achieve Low-Overhead Data Reduction on Backup Datasets," in *Data Compression Conf.*, mar 2014, pp. 203–212.

[17] R. Vestergaard, D. E. Lucani, and Q. Zhang, "Generalized Deduplication: Lossless Compression for Large Amounts of Small IoT Data," in *European Wireless Conf.*, may 2019.

[18] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep., 1994.

[19] J. Seward, "bzip2." [Online]. Available: sourceware.org/bzip2/

[20] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility," in *Proc. USENIX Conf. File and Storage Technologies*, Santa Clara, CA, USA, 2014, pp. 256–273.

[21] P. Deutsch, "GZIP File Format Specification Version 4.3," United States, 1996.

[22] I. Pavlov, "7-zip." [Online]. Available: 7-zip.org

[23] A. Makhdoumi, S. Huang, M. Médard, and Y. Polyanskiy, "On locally decodable source coding," in *IEEE ICC*, jun 2015, pp. 4394–4399.

[24] A. Pananjady and T. A. Courtade, "The Effect of Local Decodability Constraints on Variable-Length Compression," *IEEE Trans. Inf. Theory*, vol. 64, no. 4, pp. 2593–2608, apr 2018.

[25] A. Kraskov, H. Stögbauer, and P. Grassberger, "Estimating mutual information," *Physical Review E*, vol. 69, no. 6, p. 66138, jun 2004.

[26] T. M. Cover and J. A. Thomas, *Elements of information theory*. Wiley-Interscience, 2006.

[27] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, may 1996.

[28] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inf. Theory*, vol. 21, no. 2, pp. 194–203, mar 1975.

[29] A. Goldberger, L. Amaral, L. Glass *et al.*, "PhysioBank, PhysioToolkit, and PhysioNet : Components of a New Research Resource for Complex Physiologic Signals," *Circulation*, vol. 101, pp. E215–20, 2000.

[30] G. B. Moody and R. G. Mark, "The impact of the MIT-BIH Arrhythmia Database," *IEEE Engineering in Medicine and Biology Magazine*, vol. 20, no. 3, pp. 45–50, may 2001.

[31] European Network of Transmission System Operators for Electricity (ENTSO-E), "ENTSO-E Transparency Platform." [Online]. Available: https://transparency.entsoe.eu/

[32] L. Hirth, J. Mühlenpfordt, and M. Bulkeley, "The ENTSO-E Transparency Platform – A review of Europe's most ambitious electricity data platform," *Applied Energy*, vol. 225, pp. 1054–1067, sep 2018.

[33] A. M. G. Klein Tank, J. B. Wijngaard, G. P. Können *et al.*, "Daily dataset of 20th-century surface air temperature and precipitation series for the European Climate Assessment," *Int. J. Climatology*, vol. 22, no. 12, pp. 1441–1453, 2002.

[34] The ECA&D Project, "European Climate Assessment & Dataset." [Online]. Available: ecad.eu

[35] F. De la Torre, J. Hodgins, J. Montano, S. Valcarcel, R. Forcada, and J. Macey., "Guide to the Carnegie Mellon University Multimodal Activity (CMU-MMAC) Database," Carnegie Mellon University, Tech. Rep., 2009.

[36] "Carnegie Mellon University Multimodal Activity (CMU-MMAC) Database." [Online]. Available: kitchen.cs.cmu.edu

[37] A. Trindade, "Electricity Load Diagrams 2011-2014 Data Set." [Online]. Available: archive.ics.uci.edu/ml/datasets/ElectricityLoadDiagrams20112014

[38] D. Salomon and G. Motta, *Handbook of Data Compression*. Springer, London, 2010.