Refining SFDC Compression Scheme with Block Text Segmentation*

Simone Faro and Alfio Spoto

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
simone.faro@unict.it

Abstract. The Succinct Format with Direct Accessibility (SFDC) is an encoding scheme originally designed for efficient data compression and quick access to elements within compressed sequences. While SFDC performs well under stable character frequency conditions, its efficacy diminishes in text corpora with high variability in character frequencies, typical of natural language environments. Addressing this limitation, this paper presents three variant of SFDC based on block segmentation methods, each offering unique enhancements over the original SFDC representation. By tailoring the segmentation process to the distribution of characters within the text, these methods aim to optimize compression efficiency and decoding performance. The paper presents experimental results demonstrating the effectiveness of these approaches, highlighting their ability to improve upon the original scheme in several scenarios. The findings underscore the potential of these advanced segmentation strategies to provide superior compression and performance across a range of text datasets.

Keywords: text processing, text compression, computational friendly data structures

1 Introduction

Text compression is the process of reducing the size of a given text to save storage space and speed up its transmission across digital systems. This is crucial for efficient data management and quick access to information, which is especially important in environments like online streaming, real-time communication, and large-scale data storage. Formally, the challenge is to modify the representation of a text y, of length n and whose characters are drawn from an alphabet Σ of size σ, such that the new format optimizes storage space usage. The basic premise in conventional compression is that every symbol in an uncompressed text is represented by $\lceil \log_2 \sigma \rceil$ bits, totaling $n\lceil \log_2 \sigma \rceil$ bits[1]. However, by employing a variable-length encoding, such as Huffman's optimal compression scheme [7], each character $c \in \Sigma$ can be represented by a code $\rho(c)$, with the length depending on the character's frequency $f(c)$ in the text, allowing the text to be encoded in fewer bits. In an optimal compression scheme, [2] like Huffman's, the total number of bits required for the compressed text is $N = \sum_{c \in \Sigma} f(c) \cdot |\rho(c)|$. This total N is minimized in such a way that it is less than or equal to the number of bits required in the conventional representation, i.e., $N \leqslant n\lceil \log_2 \sigma \rceil$ bits. Huffman's algorithm calculates an optimal prefix code based on character frequencies and creates a binary tree (Huffman tree) for efficient decoding. These Huffman trees, while not unique, are pivotal in compression as they simplify

* This work is supported by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, co-founded by the European Union - NextGenerationEU.

[1] Throughout the paper, all logarithms are intended in base 2, unless otherwise stated.

[2] An optimal compression scheme minimizes the number of bits required to represent the text.

Simone Faro, Alfio Spoto: Refining SFDC Compression Scheme with Block Text Segmentation, pp. 27–41.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5 © Czech Technical University in Prague, Czech Republic

the decoding process. The leaves of the tree represent characters, and the branches are labeled in a binary manner, facilitating straightforward codeword retrieval. Despite their efficiency, variable-length codes like Huffman's pose a significant challenge: they do not allow direct access to the i-th codeword in the encoded string without processing previous characters. This limitation hampers quick data retrieval and affects algorithmic computations, especially in environments that benefit from rapid access. Several encoding schemes have been developed to address this issue, providing direct access while trying to maintain compactness. Notable examples include Dense Sampling [5], Elias-Fano codes [4], Interpolative coding [8,9], Wavelet Trees [6], and DACs [1,2]. These schemes vary in their approach and efficiency.

This work is based on the most recent SFDC format, which leverages variable-length codes to enable fast and direct access to any element within a compressed sequence. This format excels under conditions of low variability in character frequencies, providing consistent access times and superior compression ratios compared to other methods. SFDC's design is inherently flexible, making it suitable for a range of applications where the trade-off between efficiency and space consumption is critical. The scheme employs a total of $N + \mathcal{O}\big(n\,(\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1})\,\big) = N + \mathcal{O}(n)$ bits, where λ is parameter involved in the encoding, F_j represents the j-th Fibonacci number, achieving efficient access times that are merely proportional to the length of the character's encoding, with an expected additional time overhead $\mathcal{O}((F_{\sigma-\lambda+3} - 3)/F_{\sigma+1})$.

In addressing the limitations inherent to the traditional SFDC approach, particularly in scenarios with high variability in character frequency along the text, in this paper we introduce three block segmentation strategies within the SFDC framework. The first approach segments the text into fixed-length blocks and uses a single Huffman tree constructed from the character frequencies of the entire text. This method is straightforward but may not adapt well to varying character distributions within the text. The second approach also uses fixed-length blocks but constructs adaptively a separate Huffman tree for each block. By tailoring the Huffman tree to each block's character frequencies, this method aims to improve the compression factor. The third approach segments the text into variable-length blocks based on the occurrences of infrequent characters. This method adapts to the text's natural structure, potentially enhancing compression efficiency. All three approaches reduce the average delay in the SFDC scheme, mitigating its specific limitations. Their performance varies depending on the text characteristics. Our extensive experiments evaluate the performance of these approaches, demonstrating that the choice of segmentation strategy should be based on the specific text properties to optimize compression and decoding.

The structure of this article is as follows. Section 2 provides a comprehensive description of the SFDC representation. In Section 3, we examine a specific phenomenon that adversely affects the performance of the SFDC scheme. Section 4 introduces three new variants of the SFDC approach, which are based on block text segmentation. In Section 5, we present experimental results to assess the performance of these new variants. Finally, Section 6 offers our conclusions.

2 An Overview of the SFDC Representation Scheme

This section summarizes the SFDC representation [3], a recently introduced text compression scheme. SFDC is based on Huffman's algorithm to create optimal prefix

binary codes for characters of an alphabet Σ of size σ, with each character $c \in \Sigma$ represented by a binary code $\rho(c)$. Here, we assume characters are ordered in a non-decreasing frequency, with c_0 being the least frequent and $c_{\sigma-1}$ the most frequent. The SFDC scheme encodes a text y of length n, over an alphabet Σ, into an ordered collection of λ binary strings, where λ is a fixed integer satisfying $2 \leqslant \lambda \leqslant \max\{|\rho(c)| : c \in \Sigma\}$ and represents the size of the SFDC representation. The parameter λ is upper bounded by the length of the codeword of the alphabet character Σ, which is the longest among all characters, i.e., $\lambda \leqslant \max\{|\rho(c)| : c \in \Sigma\}$. This setup includes $\lambda - 1$ *fixed layers* and one additional *dynamic layer*, enhancing the encoding performance.

Each of the first $\lambda - 1$ layers, denoted by $\widehat{Y}_0, \widehat{Y}_1, \ldots, \widehat{Y}_{\lambda-2}$, contains the i-th bits of the encodings of the characters in y, arranged in the order they appear:

$$\widehat{Y}_i \coloneqq \big\langle \rho(y[0])[i],\ \rho(y[1])[i],\ \ldots,\ \rho(y[n-1])[i] \big\rangle,$$

where $\rho(y[j])[i] = 0$ if $i \geqslant |\rho(y[j])|$ for $0 \leqslant j < n$.

The final, dynamic layer, $\widehat{Y}_D \coloneqq \widehat{Y}_{\lambda-1}$, gathers all remaining bits of the character encodings exceeding $\lambda - 1$ in length, following a last-in first-out (LIFO) scheme. This layer adheres to specific storage rules to ensure efficient encoding and decoding:

1. Bits from the same encoding are stored sequentially from left to right.
2. Pending bits from earlier in the text precede those later in the text.
3. Pending bits are stored in the leftmost available position, respecting 1. and 2.

As proved in [3], this approach effectively manages the varying length of Huffman codes, ensuring efficient data compression and accessibility.

2.1 Encoding and Decoding procedures

During encoding, each of the n characters of the text is processed using a Last-In-First-Out (LIFO) strategy implemented with a stack $\mathcal{S}$ to manage the pending bits in the dynamic layer $\widehat{Y}_D$. At each iteration i, the algorithm places up to $\lambda - 1$ bits of $\rho(y[i])$ into the fixed layers. If $|\rho(y[i])| < \lambda - 1$, then $\lambda - |\rho(y[i])| - 1$ bits in the fixed layers remain unused. We refer to such positions as *idle bits* in the representation. If the encoding length of character $y[i]$ exceeds $\lambda - 1$, the excess bits (*pending bits*)

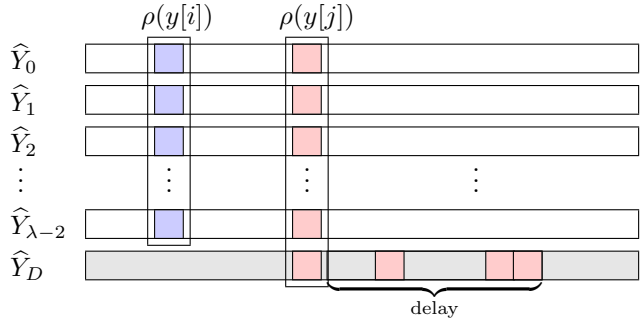

Figure 1. Examples of the reorganization of the bits of two character's code: $\rho(y[i])$ has length $\lambda - 1$ and fits within the λ layers of the representation; $\rho(y[j])$ has length $\lambda + 3$ and its 3 pending bits are arranged along the dynamic layer.

char	code	length
s	001	3
e	01	2
n	010	3
p	0110101	7
m	101	3
C	1100011010	10
o	1100111	7
i	11010	5
r	11101	5
_	00001	5

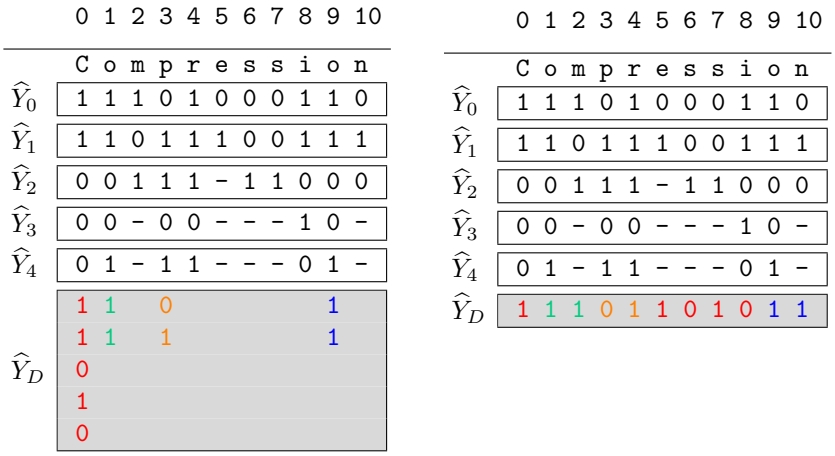

Figure 2. The SFDC representation of the string `Compression` with $\lambda = 6$ layers (5 fixed layers and an additional dynamic layer).

are pushed onto a stack $\mathcal{S}$ in reverse order. Once the bits are processed, the top bit of the stack is then stored into the dynamic layer $\widehat{Y}_D$. If the stack $\mathcal{S}$ is empty at any iteration, the corresponding bit position in $\widehat{Y}_D$ will remain idle. After all characters are processed, any remaining bits in the stack are sequentially placed into the dynamic layer, potentially extending its length beyond that of the fixed layers.

The total execution time of the encoding algorithm for a text y of length n effectively amounts to $\mathcal{O}(n+N)$, where N is the total length of the encoded text. This accounts for the time to read the input text, which is $\mathcal{O}(n)$, and the time to generate the encoded text, which is $\mathcal{O}(N)$. Note that for highly compressible texts, N can be significantly smaller than n.

The decoding of characters from a text encoded with the SFDC representation allows direct access to the start position of the encoding, but decoding time is not necessarily constant. When decoding a character $y[i]$, additional characters might need to be decoded due to the *decoding delay*, which is the additional effort required to complete the decoding of $y[i]$. This occurs if the last bit of the encoding of $y[i]$ is located at position j in the dynamic layer, where $j \geqslant i$, necessitating the decoding of all characters from position i to j, including $y[j]$. In this context, the *decoding delay* is thus defined as the need to decode additional $j - i$ characters to fully decode $y[i]$.

Assume we want to decode the window of the text $y[i\,..\,j]$. Each character $y[i]$ is decoded by traversing the Huffman tree from the root to a leaf, selecting the left or right subtree based on the bits of $\rho(y[i])$. Each node x in the tree has a Boolean value $x.\mathsf{leaf}$ indicating whether it is a leaf, and if so, $x.\mathsf{symbol}$ provides the associated character. A stack is used to manage the pending bits from the dynamic layer, storing tree nodes related to partially decoded characters. Nodes in the stack are associated with their respective positions in the text, allowing efficient positioning once decoded. The stack follows a LIFO strategy, where nodes are added or removed based on their proximity to being fully decoded.

The procedure iteratively scans each layer from $\widehat{Y}_0$ to $\widehat{Y}_D$, attempting to decode each character. If a leaf is reached early, the character is immediately decoded; otherwise,

the node is placed on the stack for further processing. The loop continues until the stack is empty and $y[j]$ is successfully decoded. The complexity of decoding a single character is $\mathcal{O}(|\rho(y[i])| + d)$, and for a window $y[i..j]$, it is $\mathcal{O}(\sum_{k=i}^{j} |\rho(y[k]|) + d)$, where d represents the decoding delay. Estimates of the expected decoding delay can be computed through simulations, adjusting λ to achieve acceptable performance levels. Of course, the time complexity of these simulations is $\mathcal{O}(n+N)$, with N being the total length of encoded text.

2.2 Performances and Evaluation

Regarding space efficiency, Cantone and Faro proved in [3] that SFDC uses a total of $N + \mathcal{O}(n)$ bits, optimizing both space and access time in the encoded representation. More formally, assuming the worst case scenario, the expected number of idle bits in a SFDC encoding using λ layers can be estimated by the following formula $\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1}$, where F_j is the j-th Fibonacci number. As the size of the alphabet σ increases, it is easy to verify that the function quickly converges to the value $\lambda - 2.618$. Consequently, if we assume that the value λ represents a constant implementation-related parameter, the total space used by SFDC for encoding a sequence of n characters is equal to $N + \mathcal{O}\left(n\left(\lambda - (F_{\sigma+3} - 3/F_{\sigma+1})\right)\right) = N + \mathcal{O}(n)$.

Regarding the expected value of the decoding delay, whch is the number of additional characters that need to be decoded in order to obtain the full encoding of a character of the text, the authors proved in [3] that, for an alphabet of σ characters, the expected delay of our SFDC encoding with λ layers (where $4 \leqslant \lambda \leqslant \sigma - 1$) can be estimated, in the worst case, by $(F_{\sigma-\lambda+3} - 3)/F_{\sigma+1}$, which is constant.

From the experimental results it emerges that the SFDC representation scheme offers a compression very close to the optimal values and offers in many cases an extremely low average delay, showing a certain competitiveness with the best compression schemes that offer direct access. However, from experimental results it turns out also that the SFDC scheme suffers particularly when the text contains portions in which the character frequency diverges significantly from the overall frequency of the text. This is the case, for instance, of datasets consisting of the union of several natural language texts, with slightly different character frequencies, or in the case of datasets containing portions in which infrequent characters are found in contiguous sequences. These circumstances lead to a phenomenon that we call *LIFO Delay Amplification*. Such phenomenon was neither identified nor analyzed by Faro and Cantone. In this paper we highlight how the LDA effect can significantly impact the performance of the SFDC encoding by increasing the decoding delay for characters preceding rare characters in the text. By bringing attention to this previously unrecognized issue, we provide a deeper understanding of the challenges associated with the SFDC scheme and propose strategies to mitigate its effects.

The following section deals with this phenomenon in detail while the rest of the paper proposes solutions to significantly mitigate it.

3 Understanding LIFO Delay Amplification in SFDC

The SFDC representation scheme employs variable-length encodings for characters. In this framework, frequently occurring characters are assigned shorter codewords,

whereas characters that appear rarely in the text are given longer codewords. For the purposes of this paper, we define *rare markers* as those characters with very low relative frequency. In this context, the *LIFO Delay Amplification* (LDA) phenomenon in SFDC refers to the unintended increase in decoding delay for characters appearing in a block that precedes a rare character in the text. This phenomenon actually occurs because rare characters can induce significant delay to characters preceding them due to their long decoding paths. Such behavior is exacerbated by the LIFO strategy adopted in the SFDC scheme which assigns decoding priority to the rightmost characters in the text.

When a rare character is encoded, all preceding characters waiting in the stack for some pending bits must wait for the rare character's encoding to complete. And this occurs also for those characters with shorter codewords that wait for a single bit on the stack. Consequently, characters that normally would require minimal delay to decode suddenly experience an increased waiting time, thus amplifying the overall decoding delay for the block. Consider the example illustrated in Fig. 3, which demonstrates the application of the SFDC representation to the text "`poor poor Compression`", assuming the character encodings provided in Figure 2. It also indicates the length of the codeword for each character in the text and its corresponding decoding delay. In this instance, the rare marker is the character "`C`" located at position 10. It is noteworthy that the block of characters preceding the "`C`" is primarily composed of characters whose codeword length is equal to or slightly exceeds the number of layers. Despite this, the delay associated with these characters increases linearly as the distance from the rare character increases, illustrating the potentially adverse effects of the LDA phenomenon. Note that the block involved in the LDA phenomenon has the shape of a ladder that descends from left to right ending, at the far right, at the position of the rare marker.

In the example of Fig. 3, the average delay of each character is just over 4. However, it is curious to observe that if we move the character "`C`" in the last position of the text the average delay would suddenly drop below 1. This highlights how the primary implication of LDA is a decrease in the efficiency of the decoding process, particularly in texts where rare characters are unevenly distributed. This can lead to significant variability in the performance of the SFDC encoding scheme, affecting both its speed and its predictability. For applications requiring consistent and rapid decoding, such as real-time data processing, this can pose considerable challenges.

This phenomenon can be observed in the experimental results shown by Faro and Cantone in [3], where a particularly inefficient behavior of the SFDC representation scheme is highlighted in the case of the ENGLISH dataset, represented by a collection of texts written in natural language, the total length of which is 100 million characters. In this case, in fact, the average delay obtained on the dataset is particularly high, equal to approximately $44,300$ characters. Analyzing the performance of the algorithm in

$y[i]$	p	o	o	r	_	p	o	o	r	_	C	o	m	p	r	e	s	s	i	o	n
$\lvert\rho(y[i])\rvert$	7	7	7	5	6	7	7	7	5	6	10	7	3	7	5	2	3	3	5	7	3
$d(i)$	23	21	1	0	15	13	11	1	0	8	7	1	0	1	0	0	0	0	0	1	0

Figure 3. An example of an SFDC scheme in which the LDA phenomenon occurs: the rare marker "`C`" in position 10 induces a significant decoding delay in the entire block of text preceding it.

more detail for this dataset, it is highlighted that the maximum delay obtained on the text is equal to approximately $17,400,000$ characters. Assuming that such high values could have been generated by a single LDA phenomenon on the text, it would be sufficient to have a rare marker whose delay is equal to just over 17 million. If this marker is capable of amplifying the delay on a block of 260 thousand characters that precede it (just 0.26% of the length of the text), the observed values would be obtained. The histogram shown in Fig. 4 confirms our analysis. It shows the delay values associated with all 100 million characters of the ENGLISH dataset. It is easy to locate a block of considerable height whose structure is typical of LDA. The histogram associated with the PROTEIN dataset does not show the occurrence of this phenomenon.

LDA is a notable challenge in the SFDC encoding approach, particularly affecting its application in environments where decoding speed is crucial. Understanding and addressing this phenomenon is essential for optimizing the performance of the SFDC scheme and ensuring their practical utility in diverse computational contexts. Mitigating the effects of LIFO Delay Amplification involves strategic block management and possibly adjusting the Huffman coding process to minimize the occurrence of lengthy rare character codes at critical points within the text. One approach could be the introduction of adaptive Huffman trees that adjust more dynamically to the text's character frequency distribution, thereby potentially reducing the impact of rare characters on overall block decoding times. In the following sections we will try to mitigate the phenomenon through the use of a text block segmentation strategy.

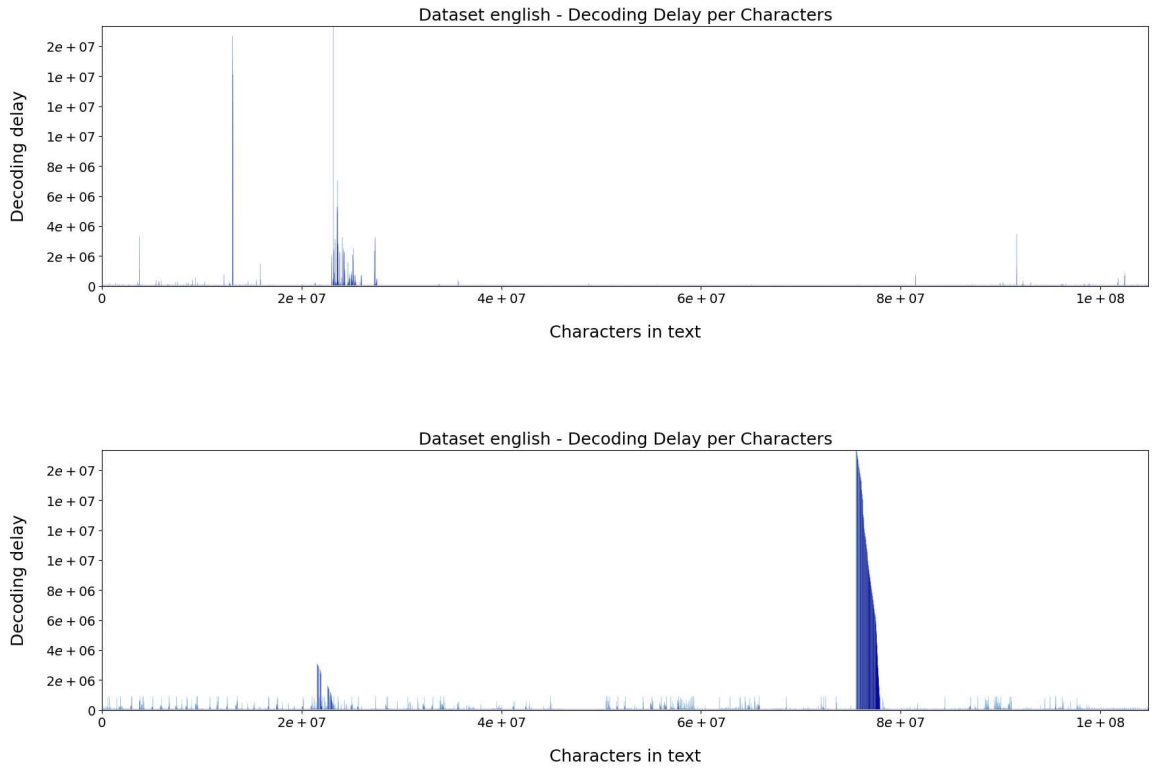

Figure 4. Histograms of the delay values associated with the 100 million characters of the PROTEIN and the ENGLISH datasets, respectively, obtained by the SFDC representation.

4 Evaluating Block Text Segmentation in SFDC

In this section we discuss some approaches based on text segmentation to address the challenges faced by LDA, especially in the context of datasets which exhibit substantial internal variability of character frequencies, like those consisting of various texts written in natural language. The text segmentation approach partitions the text into smaller blocks and compresses each block separately using the SFDC method. As a general effect, dividing the text into blocks can mitigate the effects of the LDA phenomenon by allowing the pending bits in the stack to be processed in advance. Therefore, closing a block enables the placement of all pending bits, thereby reducing the waiting times for the characters in the stack.

However, a text segmentation can be implemented in various ways. In this paper, we evaluate the following three primary segmentation strategies:

- Fixed Length Block Segmentation (Section 4.1);
- Adaptive Huffman Encoding in Fixed Length Block Segmentation (Section 4.2);
- Rare Markers Block Segmentation (Section 4.3).

4.1 Fixed Length Block Segmentation

The Fixed Length Block (FLB) is a segmentation strategy designed to divide text into blocks of a fixed length. This approach employs a single Huffman tree that is constructed over the entire dataset to define the codeword set used across all the blocks. By referncing this single Huffman tree, the encoding process remains consistent throughout the entire text. This method is straightforward and adheres closely to the original SFDC scheme, with the primary requirement being the division of text into blocks of fixed length. The use of a single Huffman tree for encoding characters within all blocks ensures that the compression factor remains equivalent to that achieved by the original SFDC. During the text encoding phase, the process under FLB is nearly identical to that of SFDC, with one key difference: at the end of each fixed-length block, there is an opportunity to empty the stack, thereby processing all pending bits of characters that are left waiting. This capability is advantageous for the performance of the representation scheme, as it tends to reduce the average delay experienced by characters within that block. As a result, a reduction in the overall average delay can be anticipated, with this decrease being more pronounced when smaller block sizes are considered. In the decoding phase of FLB, no significant modifications to the main algorithm are necessary. When decoding a specific character, it is sufficient to accurately identify the block in which the character's encoding resides. This identification process can be performed in constant time, ensuring efficiency in the decoding phase. FLB segmentation can be interpreted as a variant of the SFDC scheme. It introduces specific positions at regular intervals within the text where the stack of pending bits can be completely emptied. This insertion of emptying points is a highly efficient mechanism for addressing the challenges posed by LDA.

To summarize, the Fixed Length Block segmentation strategy simplifies the text processing procedure by utilizing fixed-length blocks and a single Huffman tree. This approach ensures consistent compression factors and offers an efficient mechanism for reducing delays in character processing, making it a robust and effective variant of the SFDC scheme in handling large datasets.

4.2 Adaptive Huffman Encoding in Fixed Length Block Segmentation

The idea of Adaptive Huffman Encoding in FLB (AFLB) Segmentation is to create a new Huffman tree for each block obtained from the segmentation of the text. This strategy ensures that the frequency function used for tree construction more accurately reflects the character frequencies within that specific block, thereby enabling more efficient character encoding and consequently reducing the average delay within the block. On the other hand, this method introduces the need to maintain multiple Huffman trees, one for each text block created by the partitioning process. This requirement can potentially complicate the management of resources and increase the computational overhead involved in the compression process.

In AFLB segmentation, let n represent the length of the text in characters, and k denote the block size. The text is divided into $\lfloor n/k \rfloor$ blocks, each containing k characters, with a final block containing the remaining $n \mod k$ characters if n is not divisible by k. Assuming the text uses the ASCII character set (256 elements) and that the average code length for Huffman encoding is approximately 4 bits, the memory requirement for one Huffman tree is around 1KB. This accounts for the storage of each character's Huffman code and the tree structure. The space overhead for maintaining a separate Huffman tree for each block is directly proportional to the number of blocks, amounting to approximately n/k KB. This relationship highlights the additional space required for separate Huffman trees, indicating the space efficiency and scalability of the compression method. However, this overhead can be significantly high in certain scenarios, potentially undermining the effectiveness of the compression scheme.

To mitigate this issue, we propose a strategy that allows for the reuse of the same Huffman tree across two or more adjacent blocks, provided the tree structures obtained from these blocks are identical or sufficiently similar. More formally, this involves constructing the tree for a new block and then checking whether its structure is similar to that of the tree from the previous block. If they are similar, the same tree can be reused; otherwise, a new tree must be constructed. This approach could significantly reduce the number of trees utilized during the compression phase, enhancing the efficiency and manageability of the SFDC encoding scheme in practical applications. This refined approach aims to balance the need for accurate, context-sensitive encoding with the practical considerations of computational efficiency and resource management, making it particularly well-suited for large-scale text analysis and data compression tasks.

In this context, we adopt the *cosine distance metric* to compute the similarity between the trees of two adjacent blocks. Cosine similarity is a critical measure in data analysis, used to determine the similarity between two non-zero vectors in an inner product space. This similarity is defined as the cosine of the angle between the vectors. Due to its ability to measure vector orientation without being affected by vector magnitude, Cosine similarity is essential in several fields. In information retrieval and text mining, it measures document similarity based on content orientation, independent of length. In data mining, it assesses cluster cohesion, replacing Euclidean distance in methods like k-means, especially for text clustering. In machine learning, it aids pattern recognition, classification, and neural network training by determining data object similarity. In the context of AFLB, cosine distance is employed to evaluate the similarity between Huffman trees derived from continuous text blocks. By trans-

forming these tree structures into vector forms we can then apply cosine distance to quantify how similar two trees are.

More formally, assume y_a and y_b are two adjacent text blocks for which we want to compute the similarity of their Huffman trees, T_a and T_b, respectively. Assume also that f_a and f_b are the two character frequency functions of y_a and y_b, respectively, and on which the Huffman trees T_a and T_b are built. The character frequency functions f_a and f_b can be interpreted as vectors over the ordered alphabet Σ. For example, if $\Sigma = \{a, b, c\}$, and the text block y_a contains the characters $\{a, a, b\}$, while y_b contains the characters $\{a, c, c\}$, then the frequency functions f_a and f_b would be defined as $f_a = (2, 1, 0)$ and $f_b = (1, 0, 2)$, where each position in the vectors corresponds to the characters in Σ in order. The distance, denoted as δ, is then expressed as:

$$\delta(T_a, T_b) = \frac{f_a \cdot f_b}{||f_a|| \times ||f_b||}$$

Here, $f_a \cdot f_b$ represents the dot product of the vectors f_a and f_b, and $||f_a||$ denotes the Euclidean norm of the vector f_a. Observe that, in this case, the value of such distance lies in the range $[0, 1]$, where 1 indicates that the vectors are proportional (identical direction), and 0 indicates that they are orthogonal (no similarity).[3] We say that two Huffman trees are similar if their distance is below or equal to a threshold γ, which in our paper is identified by the average cosine distance between all adjacent text blocks. Assuming the block size is K and $N = n/K$ is the number of blocks, we have

$$\gamma = \frac{1}{N-1} \sum_{i=0}^{N-2} \delta(T_i, T_{i+1})$$

where T_i and T_{i+1} are the Huffman trees for the i-th and $(i + 1)$-th text blocks, respectively.

It is important to consider that the cosine similarity measure can only be applied if the two blocks refer to the same character alphabet. If the second block contains characters that do not appear in the first block and, consequently, do not have defined paths in the Huffman tree, reuse of the tree is not feasible. In such circumstances, a new Huffman tree must be constructed to accommodate the unique characters present in the second block. The Adaptive Fixed-Length Block segmentation approach presented in this section, while requiring the maintenance of multiple Huffman trees, offers the potential for tailored compression for each block of text. This could lead to a reduction in the total space required for text compression. However, it is crucial to identify an optimal balance between the size of the blocks (and thus their number in the segmentation) and the overall space needed to maintain the Huffman

[3] In this paper, we evaluate the similarity between Huffman trees using cosine distance based on the frequency function of characters, rather than the more common approach of using codeword lengths. This method directly reflects the character distribution within the text and relates closely to compression efficiency, as Huffman trees are designed to minimize the weighted path length according to character frequencies. While this frequency-based approach provides a more accurate representation of the data structure it also has drawbacks. It can be more sensitive to minor variations in character distribution and more complex to interpret compared to using codeword lengths.

trees. Achieving this balance is essential for maximizing the efficiency of the AFLB approach.

4.3 Rare Marker Block Segmentation

Previous approaches, such as FLB and AFLB segmentation, perform text segmentation by dividing the text into fixed length blocks. This "blind" division does not consider the most advantageous points for ending one block and starting the next. In this section, we introduce an enhancement to the AFLB approach that segments text into variable-length blocks. Specifically, we present the *Rare Marker Block* (RMB) segmentation, a technique that utilizes the positions of infrequently occurring characters within the text as segmentation markers to define block boundaries. This method aims to improve the efficiency of the SFDC scheme by adjusting block sizes based on the distribution of low-frequency characters.

The RMB segmentation formally identifies characters $c \in \Sigma$ with a frequency $f(c)$ below a predefined threshold, termed *rare markers*. These rare markers are used to determine the points at which the text is segmented into blocks. Thus, the text y is divided into blocks such that each block ends immediately after the next occurrence of any rare marker. This approach ensures that infrequent characters act as natural dividers, optimizing the distribution of text codes within the SFDC layers. To prevent the creation of excessively small blocks when rare markers occur in close proximity, we introduce a parameter $\beta > 0$, which sets a minimum block size. Formally, a block is closed at the position of a rare marker only if the next rare marker is at least β characters away. This means that if two rare markers are found within β characters of each other, they are included in the same block, ensuring that no block is smaller than β characters. This approach helps maintain a more consistent and efficient block size, preventing the inefficiencies associated with handling very small blocks. The RMB segmentation offers several advantages:

- **Efficiency:** The segmentation adapts to the inherent structure of the text, optimizing compression performance by aligning block boundaries with the distribution of low-frequency characters.
- **Scalability:** The method scales effectively with text size and complexity, adjusting dynamically to variations in text composition and character distribution.
- **Simplicity:** The use of clearly defined markers simplifies both the encoding and decoding processes, making the method practical for large datasets.

The RMB segmentation strategy is particularly suited for texts with non-uniform character distributions, such as natural language corpora, where certain characters may appear with significantly lower frequency. This adaptability makes it an excellent choice for efficiently managing diverse text types in real-world applications.

The RMB segmentation strategy offers significant advantages for text compression but also presents several challenges. These include the potential for inconsistent block sizes, which can lead to inefficient data management, and the irregular distribution of rare characters, which may result in suboptimal compression and increased decoding times. Additionally, determining the optimal parameters for identifying rare characters and setting the minimum block size (β) is complex and dataset-specific. Non-optimal parameters can compromise performance. The approach also introduces

Text	σ	Max$\{\lvert\rho(y[i])\rvert\}$	Avg$\{\lvert\rho(y[i])\rvert\}$
protein	25	11	4.22
dblp	96	21	5.26
english	94	20	4.59

Table 1. Some noteworthy details regarding the datasets utilized in our experimental outcomes. All sequences comprise 104,857,600 elements.

computational overhead due to the need for dynamic segmentation and rare character identification, which can be problematic for real-time applications or systems with limited resources. Moreover, the increased implementation complexity compared to traditional fixed-length segmentation methods can lead to higher development and maintenance costs. Despite these challenges, addressing these issues can help fully realize the potential benefits of the RMB approach, enhancing its practical applicability and efficiency.

5 Experimental Results

In this section, we present experimental results that evaluate the performance of the three variants of the SFDC representation, which are based on block segmentation of the dataset. The evaluation criteria include average delay, the number of blocks generated, the number of Huffman trees utilized, and the compression factor achieved.[4]

Our experiments were performed on 3 real data sequences. Such real data sequences are text files of size 100 MB from the Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl`), and specifically an XML file, an English text and a protein sequence. Following the notation proposed in [2], we denote by dblp the XML file containing bibliographic information on major computer science journals and proceedings. We denote by english the English text, which contains different English text files with slightly different character distributions. We denote by protein the protein text containing protein sequences (consisting of uppercase letters for the 20 amino acids). Some interesting information about these datasets are shown in Table 1.

The FLB and AFLB strategies are evaluated based on the block sizes used for segmentation, which range from 1 KB to 100 MB. For both approaches, the average delay is measured. In addition to the average delay, the AFLB strategy also includes measurements of the number of Huffman trees used in the representation and the spatial overhead generated by these trees. The RMB strategy, on the other hand, is evaluated based on the number of rare elements used as markers, which varies from 2 to 10. Similar to AFLB, the evaluation of RMB includes measurements of the average delay, the number of Huffman trees used in the representation, and the spatial overhead generated by these trees. By considering these metrics, we aim to provide a comprehensive assessment of the performance of each strategy under varying conditions.

[4] All the implementations of the encodings presented in this paper and all the datasets are available in a Google Drive repository, accessible at `https://drive.google.com/drive/folders/-1e4aPz_TR9m4BIYo5fX0KYQhbLNuW9WG8?usp=sharing`.

	Block Size (in KB)	Avg. Delay		Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
		FLB	AFLB				
Fixed Block	10^0	2.07	1.16	104,858	104,437	76,249,148	112.810 %
	10^1	2.15	1.48	10,486	8,915	9,405,724	13.780 %
	10^2	2.18	1.63	1,049	815	991,310	1.450 %
	10^3	2.17	2.03	105	98	138,124	0.200 %
	10^4	2.16	2.02	11	7	11,520	0.020 %
	10^5	2.84	2.82	2	2	3,334	0.005 %

	Rare Elements	Average Delay	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Rare Marker	2	1.93	59	49	69,746	0.09 %
	4	1.80	223	185	235,690	0.30 %
	6	1.81	323	273	331,508	0.42 %
	8	1.71	643	529	620,926	0.74 %
	10	1.66	1,171	1,002	1,021,286	1.19 %

Table 2. Experimental results obtained on the DBLP text using 6 layers. The results must be evaluated considering the standard version of SFDC shows an average delay equal to 2.19, and that the compressed text has a size of 68.91 MB.

	Block Size (in KB)	Avg. Delay		Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
		FLB	AFLB				
Fixed Block	10^0	10.06	3.71	104,858	104,559	55,760,356	95.420 %
	10^1	63.08	21.36	10,486	9,391	9,156,926	15.470 %
	10^2	502.03	197.77	1,049	655	802,572	1.350 %
	10^3	2,852.33	2,122.98	105	88	129,328	0.220 %
	10^4	16,957.22	12,615.84	11	8	13,070	0.020 %
	10^5	59,829.62	59,016.76	2	2	3,368	0.015 %

	Rare Elements	Average Delay	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Rare Marker	2	9,680.20	24	15	16,650	0.02 %
	4	568.66	263	200	246,444	0.33 %
	6	1,048.50	464	368	415,620	0.56 %
	8	135.82	1,749	1,398	1,275,560	1.47 %
	10	114.41	2,372	1,947	1,621,346	1.83 %

Table 3. Experimental results obtained on the ENGLISH text using 5 layers. The results must be evaluated considering the standard version of SFDC shows an average delay equal to 44,387.30, and that the compressed text has a size of 60.1 MB.

Table 2, Table 3 and Table 4 show the results obtained on the DBLP, ENGLISH and PROTEIN datasets, respectively. From experimental results it turns out that the FLB approach often achieves significant reductions in average delay compared to the original SFDC scheme. Generally, better results are observed with smaller block sizes. This is because the delay induced by rare characters is minimized due to the limited block size; the smaller the block, the lower the maximum potential delay caused by these rare markers. The most notable improvements are observed with the ENGLISH and PROTEIN datasets. For the ENGLISH dataset, using 1KB blocks results in an average delay of 10, an improvement of three orders of magnitude compared to SFDC. For the PROTEIN dataset, the average delay decreases to 0.26, which is 75% less than that achieved by SFDC.

	Block Size (in KB)	Avg. Delay FLB	Avg. Delay AFLB	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Fixed Block	10^0	0.26	0.16	104,858	39,610	6,378,498	11.760 %
	10^1	0.45	0.15	10,486	2,525	417,090	0.760 %
	10^2	0.94	0.21	1,049	289	50,034	0.090 %
	10^3	1.02	0.66	105	54	10,072	0.020 %
	10^4	1.01	0.98	11	4	876	0.012 %
	10^5	1.06	1.07	2	2	448	0.004 %

	Rare Elements	Average Delay	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Rare Marker	2	0.77	114	70	13,316	0.02 %
	4	0.61	4,244	1,495	258,920	0.34 %
	6	0.45	19,900	12,033	487,114	0.79 %
	8	0.39	22,794	19,458	863,314	1.26 %
	10	0.21	35,395	31,147	1,223,612	1.96 %

Table 4. Experimental results obtained on the PROTEIN text using 5 layers. The results must be evaluated considering the standard version of SFDC shows an average delay equal to 1.02, and that the compressed text has a size of 55.36 MB.

In the case of AFL segmentation, the results in terms of average delay are significantly better than those obtained with FLB. This improvement is due to the adaptive approach, which allows for more tailored and efficient coding for each block. The most substantial improvements are seen with smaller block sizes, with reductions of up to 60%, although this advantage diminishes as block size increases. However, AFLB must contend with the overhead of maintaining multiple Huffman trees. For 1KB block sizes, this overhead becomes prohibitive, reaching almost 113% of the compressed text size for the DBLP dataset. Therefore, larger block sizes are necessary to achieve a reasonable trade-off, allowing the overhead to decrease to around 1%. This optimal block size is 100 KB for both DBLP and ENGLISH and 10 KB for PROTEIN. Additionally, it is observed that utilizing the Huffman tree reuse technique based on cosine similarity can reduce the number of required trees by up to 20%. This reduction diminishes gradually as the number of blocks increases, justified by the fact that small adjacent blocks tend to have minimal variations in character frequency distribution. This trend is consistent across all datasets.

Lastly, the performance of the RMB approach did not meet expectations. Although this technique offers considerable improvements and achieves low spatial overhead, the average delay values are generally higher than those provided by AFLB for an equivalent number of blocks. This is attributed to the highly variable block sizes created by the RMB technique. In this context, very large blocks fail to deliver good performance in terms of delay, reducing the overall effectiveness of the approach.

In conclusion, for the DBLP dataset, the best performance is achieved with AFLB using 100KB blocks, resulting in an average delay of 1.63 with only a 1.45% increase in spatial overhead. For the ENGLISH dataset, the optimal performance is obtained with FLB using 1KB blocks, achieving an average delay of 10. Finally, for PROTEIN, the best results are achieved with AFLB using 10KB blocks, resulting in an average delay of 0.15 with less than a 1% increase in spatial overhead.

6 Conclusions and Future Work

In this article, we have explored three primary text compression strategies: Fixed-Length Block (FLB) segmentation, Adaptive Fixed-Length Block (AFLB) segmentation, and Rare Marker Block (RMB) segmentation. Each approach offers unique benefits and addresses different aspects of the text compression challenge. Looking forward, several promising directions for future research have been identified. One avenue is to investigate the use of rare markers as starting points of blocks rather than ending points. This adjustment could potentially optimize the segmentation process further and reduce the delay introduced by rare characters. Additionally, exploring the efficacy of a First-In, First-Out (FIFO) strategy as opposed to the Last-In, First-Out (LIFO) strategy currently used could provide insights into improving the decoding efficiency.

References

1. N. Brisaboa, S. Ladra, and G. Navarro: *Directly addressable variable-length codes*, in SPIRE 2009, vol. 5721 of LNCS, Springer, 2009, pp. 122–130.
2. N. R. Brisaboa, S. Ladra, and G. Navarro: *Dacs: Bringing direct access to variable-length codes*. Inf. Process. Manag., 49(1) 2013, pp. 392–404.
3. D. Cantone and S. Faro: *The many qualities of a new directly accessible compression scheme*. CoRR, abs/2303.18063 2023.
4. P. Elias: *Efficient storage and retrieval by content and address of static files*. J. ACM, 21(2) 1974, pp. 246–260.
5. P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds*. Theor. Comput. Sci., 372(1) 2007, pp. 115–121.
6. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM/SIAM, 2003, pp. 841–850.
7. D. A. Huffman: *A method for the construction of minimum-redundancy codes*. Proceedings of the Institute of Radio Engineers, 40(9) September 1952, pp. 1098–1101.
8. A. Moffat and L. Stuiver: *Binary interpolative coding for effective index compression*. Inf. Retr., 3(1) 2000, pp. 25–47.
9. J. Teuhola: *Interpolative coding of integer sequences supporting log-time random access*. Inf. Process. Manag., 47(5) 2011, pp. 742–761.