

A Dynamic Text Compression Model for Big Data Applications Using Hadoop

Vanisha Mavi*

Abstract

In today's data-driven era, efficiently handling vast amounts of information has become increasingly important. Data compression plays a vital role in this regard—it is essentially a method of encoding information in such a way that significantly reduces the number of bits required to store or transmit a file. By shrinking data to its most compact form, compression techniques help save storage space, reduce bandwidth consumption, and improve the overall speed of data transfer. A wide variety of data compression techniques exist today, operating across both online and offline environments. However, the sheer number of available options makes it genuinely challenging for practitioners and researchers to determine which technique best fits their specific needs. Every method has its own set of trade-offs regarding speed, compression efficiency, and computational complexity. In this paper, we explore and compare several key techniques used to compress and decompress text data, aiming to give readers a clearer understanding of how these methods work in practice. Beyond compression alone, we also take a closer look at Hadoop—one of the most widely adopted frameworks for distributed data processing—covering its design philosophy, development journey, and where it currently stands in real-world deployment. Additionally, we provide a comprehensive overview of YARN (Yet Another Resource Negotiator), the resource management layer within the Hadoop ecosystem. In this paper, we discuss its core concepts, the challenges that arise during its use, and the various tools and techniques that can be leveraged alongside it to build more efficient and scalable data pipelines.

Keywords: Hadoop, MapReduce, YARN, data compression, textual substitution

INTRODUCTION

Bigdata

Big Data encompasses extremely large and complex datasets that exceed the capabilities of traditional storage and analysis tools, requiring advanced processing models for effective management and decision-making. These technologies enable the extraction and analysis of vast amounts of information to uncover valuable insights through specialized techniques that conventional Database Management

Systems cannot efficiently handle [1]. The scale of data generation is massive, with everyday online activities such as social media uploads, messaging, and e-commerce contributing to approximately 2.5 quintillion bytes of data created daily, with nearly 90% of global data generated within the past two years, according to IBM.

Hadoop

Hadoop is an Apache Foundation-developed distributed computing framework that enables users to create distributed applications and leverage cluster resources for high-speed processing and

*Author for Correspondence

Vanisha Mavi

E-mail: Vanisha.mavi@gmail.com

Research Scholar, Department of CSE, Shobhit Institute of Engineering & Technology, (Deemed to be University), Meerut, Uttar Pradesh, India

Received Date: April 15, 2026

Accepted Date: April 29, 2026

Published Date: May 02, 2026

Citation: Vanisha Mavi. A Dynamic Text Compression Model for Big Data Applications Using Hadoop. International Journal of Algorithms Design and Analysis Review. 2026; 4(2): 1–8p.

storage without requiring a deep knowledge of the underlying system architecture. Its core components include the Hadoop Distributed File System (HDFS) and MapReduce [2].

REVIEW LITERATURE

Data Compression

Source coding, or bit rate reduction, encodes information using fewer bits than the original format through data compression. This process involves two main types: lossless compression, which removes statistical redundancy while preserving all information, and lossy compression, which reduces the file size by eliminating less critical data [3]. When applied to data transmission, this technique is specifically termed source coding because it occurs at the data's origin before storage or transmission, differentiating it from channel coding [4].

Common Input Format

Table 1 illustrates the common input format of the compression technique.

Existing Data Compression Techniques

The existing data compression techniques are described as follows.

Bit Reduction Algorithm

The program's fundamental principle involves transforming standard 7-bit encoding to a specialized 5-bit encoding scheme and then packing the results into byte arrays [5]. This method substantially reduces the size of lengthy strings while maintaining a consistent compression ratio independent of the string content, achieved through the systematic application of the Bit Reduction Algorithm:

- The algorithm focuses on compressing text by analyzing the frequency of characters.
- It starts by identifying the most frequently used characters in a text file.
- The ASCII values of these characters are then determined.
- These ASCII values are converted into binary form.
- The binary representations are stored in 8-bit (byte) arrays.
- Compression is achieved by removing unnecessary bits, such as leading extra bits.
- The remaining bits are reorganized within the byte array structure.
- The output is an encoded and compressed version of the original text.
- Decompression is performed on the client side by reversing the encoding steps.

Huffman Coding

Huffman coding is a data compression technique that encodes ASCII characters using a top-down binary tree approach to create an efficient encoding scheme [6]. The method assigns shorter binary codes to frequently occurring characters and longer codes to less frequent ones, optimizing compression efficiency [7]. The process involves constructing a binary tree where leaf nodes represent characters to be encoded, with each node storing the probability of occurrence of characters in its subtree and edges labeled with binary digits 0 and 1[8]:

- The algorithm starts by examining the input data to determine the frequency of each symbol.
- It then calculates the probability of occurrence for each symbol.
- Symbols are arranged in order from highest to lowest probability.
- Leaf nodes are created for each symbol and placed into a queue.
- The two symbols with the lowest frequencies are repeatedly combined to form new nodes with their total frequencies.
- This process builds a binary tree structure step by step.
- The procedure continues until only one node remains, which is the root of the tree.
- Binary codes were assigned by labeling the left and right edges as 0 and 1, respectively.
- Tracing paths from the root to each leaf generates Huffman codes, where more frequent symbols receive shorter codewords.

Table 1. Common input formats.

Compression format	Tool	Algorithm	File extension	Split table
Gzip	Gzip	DEFLATE	.gz	no
bzip2	bzip2	bzip2	.bz2	yes
LZO	Lzop	LZO	.lzo	Yes, if indexed
Snappy	n/a	Snappy	.Snappy	No

Run-Length Encoding

Run-length encoding (RLE) is a simple data compression technique that efficiently handles repeated sequences of identical bytes by replacing them with a count and the actual byte value, rather than storing each repetition individually [9]. This method is widely used in bitmap formats, such as BMP, and works by encoding consecutive repeating characters (runs) using two bytes: one for the repetition count and another for the character itself. RLE can be considered an extension of zero suppression, where frequently occurring symbols are efficiently encoded using special markers and counts [10]. The technique is particularly effective for data with high redundancy levels and can be combined with other compression methods to enhance overall efficiency [11].

Here is an example of RLE:

Input: YYYBBCCCCDEEEEEERRRRRRRRRR

Output: 3Y2B4C1D6E10R

The main limitation of the RLE algorithm is that it does not provide compression ratios as high as more advanced techniques [12]. However, its key advantage lies in its simplicity and fast execution, making it a practical alternative to more complex compression methods.

PROPOSED WORK

Textual Substitution

Compression algorithms feature adjustable parameters that affect their performance characteristics. The Lempel-Ziv method exemplifies this through its dictionary size parameter: larger dictionaries achieve superior compression ratios but require more execution time, whereas smaller dictionaries enable faster processing with reduced compression efficiency. The algorithm selection depends on specific application requirements and system constraints, as higher compression typically correlates with slower speeds. To address this performance trade-off, textual substitution or dictionary-based methods offer a balanced approach between compression effectiveness and processing speed [13].

A textual substitution compression method operates by identifying repeated substrings within the text and substituting them with pointers that reference their previous occurrences, while also shortening words when feasible. The compression algorithm processes a character sequence from a specified alphabet and produces output containing both individual characters and pointers to earlier substrings. Because the encoding must accommodate single characters, each character can be conceptualized as a pointer to a one-character string, allowing the entire compressed output to be represented as a sequence of pointers referencing substrings of varying lengths [14].

Definition

A compression–decompression process is qualified as real-time when it operates at a constant rate defined by parameter k , where the encoder reads one character and the decoder outputs one character every k time units, regardless of the input data. The system allows for a small initial delay (constant l) before decoding begins and a similar delay after encoding finishes for finite inputs. Online textual substitution techniques represent highly effective approaches for lossless data compression.

Most compression algorithms follow a common framework involving an encoder that converts character streams to bit streams and a decoder that reconstructs the original data. Both components maintain synchronized dictionaries with predefined maximum sizes to ensure proper decoding. The

encoder searches for input matches in the dictionary, sends pointers to matched entries, and updates the dictionary accordingly, applying replacement strategies when full. The decoder mirrors this process by reading the pointers, retrieving the corresponding dictionary entries, and updating its dictionary identically. This technique can be effectively implemented on distributed platforms, such as Hadoop, for text file compression.

Textual substitution compression is done according to the following example:

Input stream: AAAAAAABBBBCCCCCCCCAAAAAADDDDD

Output stream: 7A4B7C6A5D

It not only reduces the space between the strings but also creates a short form of the word.

The left out of Algorithms A1 and A2 is the specification of the following:

Algorithm

A1-Encoding Algorithm

1. Initialize the local dictionary D with the set INIT.
2. Repeat forever:
 - (a) Obtain the current match.
 $t := MH(inputstream)$
 Advance the input stream forward by $|t|$ characters.
 Transmit $\lceil \log_2 |D| \rceil$ bits corresponding to t.
 - (b) Update local dictionary D:
 $X := UH(D)$
 while $X \neq \epsilon$ and (D is not full or $DH(D) \neq \epsilon$) do begin
 Delete an element x from X.
 if x is not in D, then begin
 if D is full then Delete $DH(D)$ from D.
 Add x to D.
 end
 end

A2-Decoding Algorithm

1. Initialize the local dictionary D by performing Step 1 of the encoding algorithm.
2. Repeat forever:
 - (a) Obtain the current match.
 Receive $\lceil \log_2 |D| \rceil$ bits.
 Obtain the current match t by a dictionary lookup.
 Output the characters of t.
 - (b) The local dictionary D is updated by performing Step 2b of the encoding algorithm.

The initialization heuristic, INIT: A local dictionary is initialized using a string collection that includes the alphabet and maintains a size within the maximum dictionary-capacity limit.

The match heuristic, MH: A function that extracts and removes a string t from the input stream when t exists in dictionary D [15].

The update heuristic, UH: A function that uses the local dictionary D to generate a set of strings to be added to it, provided there is enough available space.

The deletion heuristic (DH) is a function that operates on the local dictionary D and returns either an empty set—indicating that no entries can be removed—or a single string from D that is not part of the initial set (INIT) and can therefore be safely deleted. While there are several considerations in selecting INIT and the matching heuristic (MH), we focus on a simplified case where INIT includes all characters of the alphabet Σ in the dictionary, and MH follows a greedy approach by always selecting the longest possible match from the input stream. One possible update strategy (UD) is the FC (First Character) heuristic, which adds a new entry formed by concatenating the previous match with the first character of the current match [16].

ID (identity heuristic): Add the previous match concatenated with the current match.

AP (all prefixes heuristic): This approach adds new entries to the dictionary by combining the previous match with every prefix of the current match. Additionally, there are three possible options for the DH:

FREEZE (freeze heuristic): DH (D) is defined as the empty set, meaning that once the dictionary reaches its maximum capacity, no further changes are made, and it remains unchanged thereafter [17].

LRU (least recently used heuristic): DH (D) refers to the string in the dictionary D that has been used least recently for matching.

SWAP (swap heuristic): This heuristic can be considered a discrete form of the least recently used (LRU) strategy. When the main dictionary becomes full for the first time, a secondary (auxiliary) dictionary is created while compression continues using the primary dictionary. Thereafter, whenever the auxiliary dictionary fills up, the two dictionaries swap roles and the new auxiliary dictionary is cleared. Although this approach does not directly align with Algorithms A1 and A2, those algorithms can be adjusted to support it [18].

The FC–FREEZE heuristic can be considered a practical implementation of the second model introduced by Lempel and Ziv, whereas the FC–LRU heuristic serves as a natural enhancement of the FC–FREEZE approach.

Measuring Compression Performances

The evaluation of compression algorithms relies on multiple criteria tailored to specific applications, with space efficiency being the primary factor and time efficiency being a secondary consideration. Performance assessment is inherently challenging due to the algorithm's dependence on symbol redundancy within source data, making general evaluations difficult [19]. The effectiveness of data compression varies significantly based on the input data characteristics, including the type and structure, necessitating the use of diverse metrics to comprehensively measure the algorithmic performance.

- *Compression ratio:* Compression Ratio is the ratio between the size of the file after compression and the size of the file before compression.
Compression ratio = Size after compression/size before compression.
- *Compression factor:* Compression factor is the inverse of the compression ratio. It is the ratio between the size of the file before compression and the size of the file after compression.
Compression factor = Size before compression/size after compression.

Working of Textual Substitution

The text explains textual substitution compression, a method that identifies higher-order patterns in text and similar data without using predefined statistical probabilities. This approach, developed by Ziv and Lempel, operates on the principle that repeated sequences of previously encoded symbols can be represented more efficiently, as illustrated in Figure 1. This compression approach handles repeated sequences using copy codewords that replace the actual symbols. When a sequence reappears, the

following suffix character is preceded by a codeword that points to the position of the earlier occurrence and specifies the sequence length. Because the copy codeword completely describes the repeated sequence, it substitutes the original symbols to create a compressed data representation.

In a textual substitution compression system, enhancement involves implementing a fixed-size search window to store recently processed data symbols. These symbols are organized using a search tree structure with a predetermined depth, which links them according to their occurrence order and monitors their movement through the window. The depth of the tree corresponds to the maximum permissible length of the matching sequences. The system incorporates a mechanism for adding new uncompressed symbols to the search tree, which facilitates pattern recognition and determines the length and position of the longest matching sequence within the search window. The method produces copy and literal codewords as its two main output types. Copy codewords encode repeated symbol sequences by indicating their length and location within the search window, whereas literal codewords handle non-repeating symbols and can contain additional literal characters. The selection between these codeword types is determined by whether the input symbols create a repeated sequence that meets the minimum length requirement.

Comparison Table and Graph on Different Compression Techniques

Table 2 and Figure 2 represent the comparison of compression ratios of the existing techniques and the proposed system.

About 88% of the data is compressed with textual substitution in Hadoop.

The graph below was created based on the data in Table 2 and illustrates a comparison of the compression ratios between the existing and proposed systems. In this graph, the horizontal axis indicates the length of the input string in bytes, whereas the vertical axis represents the compression ratio as a percentage.

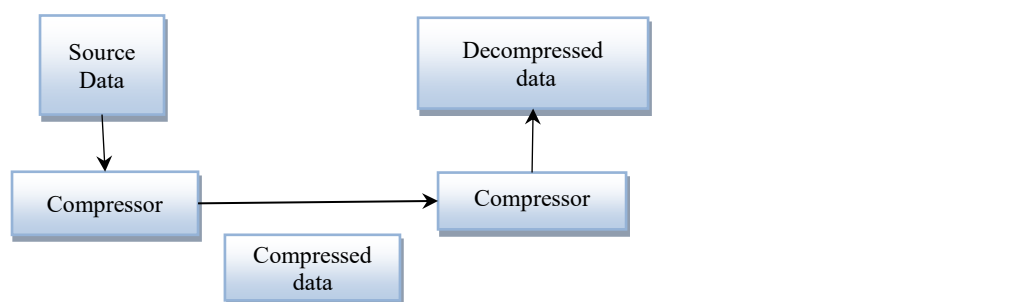


Figure 1. Compressed and decompressed data.

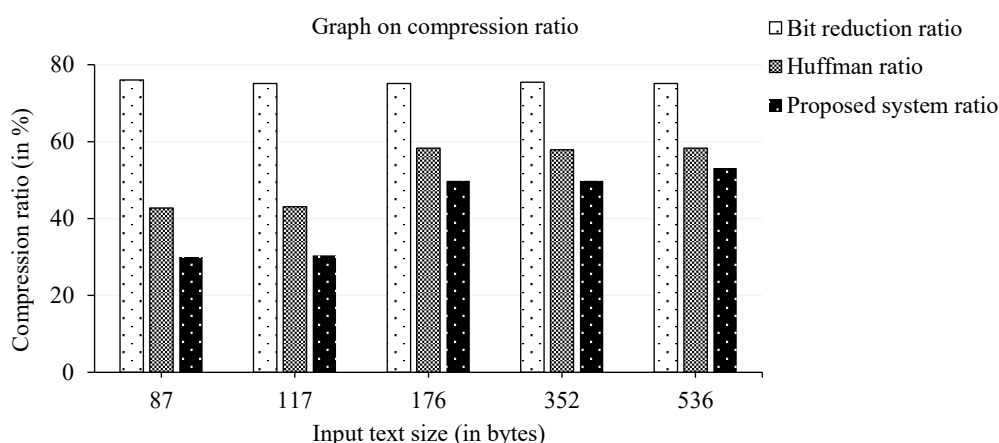


Figure 2. Comparison between different compression techniques.

Table 1. Comparison ratio between different techniques.

Input text size (in megabytes)	Bit reduction compression ratio (in mb)	Huffman compression ratio (in mb)	Proposed system compression ratio (in mb)
87	75.8	42.5	29.82
117	75.2	42.7	30.7
176	75.0	57.9	49.4
11.7	10.2	8.7	4.56
4.2	4.0	3.4	1.65

CONCLUSION

The proposed compression technique effectively transmits large data volumes over the Internet while maintaining lossless information transfer. It demonstrates efficiency with data containing repeated character sequences, which are more easily encoded, and guarantees accurate decompression of the original data. This text describes the development of an enhanced dynamic bit reduction algorithm designed for lossless text compression and decompression. The algorithm was tested on diverse datasets encompassing random, alphanumeric, numerical, and special character data to ensure a comprehensive evaluation. The performance assessment was conducted by comparing the proposed method with established compression techniques, specifically Bit Reduction and Huffman Coding, using key metrics, including the compression ratio and savings percentage, to measure effectiveness. The analysis demonstrates that the proposed system outperforms existing methods by delivering superior compression ratios and greater savings across all tested datasets. Unlike traditional fixed-bit encoding techniques that suffer from lower efficiency and potential data loss with special characters, the new approach utilizes a variable-bit encoding scheme. This dynamic methodology effectively addresses the limitations of conventional methods, resulting in significantly enhanced compression performance.

Need for Compression

1. Data are mostly stored and not frequently processed. This is a typical DWH scenario. In this case, space saving can be much more significant than the processing overhead.
2. The compression factor is very high, and therefore, we save a lot of IO [20].
3. Decompression is very fast (like Snappy), and therefore, we have some gain at a low price.
4. Data already arrives compressed.

The proposed work states that it can provide flexibility in achieving good compression without data loss.

Future Work

We will develop a picture and audio data compression technique that will provide better efficiency and an effective compression ratio. Other compression techniques work only with text data written in a single language, which can also be tested to compress multilingual data, that is, text data written in multiple languages in a single file. In other words, the system works only on an ASCII dataset, which can be extended to work with Unicode data for future work.

REFERENCES

1. Mo Z, Li Y. Research of Big Data based on the views of technology and application. *Am J Ind Bus Manag.* 2015;5(4):192-7. doi:10.4236/ajibm.2015.54021.
2. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, et al. Apache Hadoop YARN: yet another resource negotiator. In: *Proceedings of the 4th Annual Symposium on Cloud Computing.* New York (NY): Association for Computing Machinery; 2013. p.1–16. doi:10.1145/2523616.2523633.
3. The Apache Software Foundation. (2026). Tez Incubation Status. [Online] Apache Incubator. Available from: <https://incubator.apache.org/projects/tez.html>.

4. Mavi V, Tyagi N. Apache Hadoop's second generation: Performance and resource management through YARN. *Int J Eng Sci Humanit.* 2023;13:52–60.
5. Bali N, Chaudhary D, Singla A. Big data and its applications. *J Technol Manag Grow Econ.* 2020;11:63–67. doi:10.15415/jtmge.2020.112008.
6. Lama P, Zhou X. AROMA: automated resource allocation and configuration of MapReduce environment in the cloud. In: *Proceedings of the 9th International Conference on Autonomic Computing.* New York (NY): Association for Computing Machinery; 2012. p.63–72. doi:10.1145/2371536.2371547.
7. Saha B, Shah H, Seth S, Vijayaraghavan G, Murthy A, Curino C. Apache Tez: A unifying framework for modeling and building data processing applications. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* New York (NY): Association for Computing Machinery; 2015. p. 1357–1369. doi:10.1145/2723372.2742790.
8. Kaur R, Goyal M. A survey on the different text data compression techniques. *Int J Adv Res Comput Eng Technol.* 2013;2:711–714.
9. Shanmugasundaram S, Lourdasamy R. A comparative study of text compression algorithms. *Int J Wisdom Based Comput.* 2011;1:68–76.
10. Singh A, Bhatnagar Y. Enhancement of data compression using incremental encoding. *Int J Sci Eng Res.* 2012;3:1–5.
11. Altarawneh H, Altarawneh M. Data compression techniques on text files: A comparison study. *Int J Comput Appl.* 2011;26:42–54. doi:10.5120/3097-4249.
12. Kaur H. A review of data compression techniques and data compression symmetry. *Int J Comput Sci Inf Technol.* 2013;4:183–186.
13. Sharma A, Vyas S. Hadoop2 YARN. *IPASJ Int J Comput Sci.* 2015;3:30–34.
14. Bhosale HS, Gadekar DP. A review paper on big data and Hadoop. *Int J Sci Res Publ.* 2014;4:1–7.
15. Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Commun ACM.* 2008;51:107–113. doi:10.1145/1327452.1327492.
16. Grover M, Malaska T, Seidman J, Shapira G. *Hadoop Application Architectures: Designing Real-World Big Data Applications.* Sebastopol (CA): O'Reilly Media; 2015.
17. Hashem IAT, Anuar NB, Gani A, Yaqoob I, Xia F, Khan SU. MapReduce: Review and open challenges. *Scientometrics.* 2016;109:389–422. doi:10.1007/s11192-016-1945-y.
18. Park K, Peng L. A design of high-speed big data query processing system for social data analysis: Using Spark SQL. *Int J Appl Eng Res.* 2016;11:8221–8225.
19. Karun AK, Chitharanjan K. A review on Hadoop—HDFS infrastructure extensions. In: *2013 IEEE Conference on Information & Communication Technologies (ICT).* Thuckalay, India. 2013. p.132–137. doi:10.1109/CICT.2013.6558077.
20. Perwej Y, Kerim B, Adrees MS, Sheta OE. An empirical exploration of the YARN in big data. *Int J Appl Inf Syst.* 2017;12:19–29. doi:10.5120/ijais2017451730.