Benchmarking Data Compression for Near sorted Data

Shivangi (U35642613)
Vani Singhal (U49333648)

April 2023

1 Abstract

Data is being generated rapidly and this has made compression of this Data very essential for saving storage spaces and reducing the transfer time. Data Compression Algorithms depend on various factors, in this project we are going to focus on the effect of the sortedness of Data on the performance of Data Compression Algorithms. But, there isn't much research on this, existing research assumes the data to be either completely sorted or unsorted, which is not practical for real-world applications. Using this project, we aim to understand how the performance of various Data Compression Algorithms changes with varying levels of sortedness.

2 Introduction

2.1 Motivation

Huge amounts of data are being generated and stored worldwide, which leads to the need for efficient Data Compression. By using Data Compression Algorithms, we can compress great volumes of data into a format that requires less storage space, and if we efficiently compress the Data, we can also reduce the transfer time making it efficient to manage and transmit. However, the performance and efficiency of these algorithms depend on the degree of sortedness of the Data. As mentioned above, the existing research on this matter only focuses on completely sorted or unsorted Data. Therefore, we aim to explore the performance of Data Compression Algorithms on Data with varying degrees of sortedness. The result of our research could provide valuable insights and could help optimize Data Compression Algorithms for real-world scenarios.

2.2 Problem Statement

Our aim for this project is to compare the performance of different Data Compression Algorithms with changing levels of Data Sortedness and understand the relationship between Data sortedness on the performance of Data Compression Algorithms. As we all know, Data is rarely sorted in real-world scenarios therefore, we also want to compare these Data Compression Algorithms and try to find the one that performs the best with partially sorted Data.

2.3 Contributions

Till now, we have made several contributions toward understanding the performance of Data Compression Algorithms on different types of Data with varying levels of sortedness. We began by going through some of the related literature to understand the research that has been done on this topic. From the literature, we learned about the research done on various Data Compression Algorithms like Huffman Encoding, Run Length Encoding, and Frequent Value Compression, etc, we also learned about the strengths and limitations of these algorithms. The literature also discussed the impact of Data Type on these compression algorithms, suggesting that not all algorithms work for all Data types. All in all, the literature review helped us in developing a better understanding of the topic and identify the gaps that we can address using this project. After getting a better understanding of the topic and existing research, we focussed on understanding the performance of Run Length Encoding on different Data Types and different degrees of sortedness. We explored RLE on different workloads including string, sorted string, integer, and sorted integer data with some redundancies.

In the next phase of the project, we plan to work on developing a modification of RLE which can handle continuously sorted data without any redundancies. Furthermore, we plan to test four more compression algorithms on different workloads and compare their performances.

3 Background

We know that the amount of data being generated every day is increasing exponentially and this poses the question of Data Storage. If we keep storing these quintillion bytes of Data without any kind of compression or encoding, we are

going to run out of resources pretty soon. That is why Data Compression is very essential nowadays. These Data Compression Algorithms squeeze large amounts of data, using some kind of encoding, in a format that is compact and can be decompressed when needed.

Data Compression Algorithms depend on many factors including the degree of sortedness of Data. Data Sortedness is the measure of how arranged the data is in a specific order. Let's consider an example, say we have time series data, then the order of data could be chronological.

In this project, we will just focus on Data Sortedness and will try to compare the performance with different degrees/levels of sortedness. If our data is sorted in some order, Data Compression Algorithms can exploit the repetitive patterns and the redundancy to efficiently compress the Data. This was the level of Data sortedness can have an impact on the performance of Data Compression Algorithms.

However, most research studying sortedness on compression performance assumes that the data is either completely sorted or unsorted. Practically, this rarely happens, data is usually partially sorted, and the degree of sortedness depends on the application and the data source. Hence, we offer to explore the performance of some Data Compression Algorithms with varying degrees of sortedness in this project. Specifically, we aim to examine the following:

- Effect of degree of sortedness on the compression ratio and compression speed for different compression algorithms.
- Try to find Compression algorithms that perform better than others for partially sorted data.

4 Architecture

Basically, for this project, we implemented five Data Compression algorithms including Run Length Encoding, Delta Encoding, Huffman Encoding, LZ77 Encoding and Snappy Algorithm, and compared their performance on the basis of different levels of sortedness.

The first step was Data preparation, currently, we implemented a workload generator in Python that generates different-sized workloads with varying K-L values.

In the next step, we implemented the above-mentioned Data Compression Algorithms and benchmarked and results we got from these algorithms, and compared them on the basis of time taken for encoding and compression ratio.

4.1 Run Length Encoding

Algorithm 1 Run Length Encoding

```
function runLengthEncode(data)
    count ← 1
    currentValue ← data[0]
    Initialise the encoded
    for i ← 1 to data.size() do
        if input[i] = currentValue then
            count ← count + 1
        else
            Add the currentValue and its count to encoded
            count ← 1
            currentValue ← data[i]
        end if
    end for
    return encoded
end function
```

Run Length Encoding is a lossless data compression technique that compresses the data by representing repetitive data sequences using fewer bytes. The algorithm takes an input array of data and iterates over each element. It keeps track of the current value and the count of its consecutive occurrences in the input array.

After iterating through the entire input array, the algorithm returns the encoded array containing pairs of values, where the first value represents the value of a repeated sequence, and the second value represents the count of consecutive occurrences of that value.

The main advantage of Run Length Encoding (RLE) is its simplicity and efficiency. It is easy to implement and requires minimal computation, making it a fast compression technique suitable for real-time data compression.

4.2 Huffman Encoding

Algorithm 2 Huffman Encoding

```
function HUFFMANENCODING(data)
    probability ← Probabilities of all the elements in the data
    pq ← priority queue of nodes sorted by probability
    for i in probability do
        create new node n with i and probability[i] as its values
        add n to pq
    end for
    while pq contains more than one node do
        remove nodes n1 and n2 with the lowest frequency from pq
        create new node n with n1 and n2 as its children and the sum of their frequencies as its value
        add n to pq
    end while
    root ← the remaining node in pq
    code ← empty dictionary
    traverse root to assign binary codes to each element in probability, adding 0 for each left child and 1 for each right child
    for i in probability do
        code[i] ← binary code assigned to i
    end for
    encoded ← Initialize as empty string
    for i in data do
        encoded ← encoded + code[i]
    end for
    return encoded
end function
```

Huffman Encoding is a lossless data compression technique that assigns binary codes to each element in the input data based on their frequencies of occurrence. The basic idea behind Huffman Encoding is to assign shorter binary codes to elements that occur more frequently, and longer codes to elements that occur less frequently.

This algorithm takes input data and the probabilities of all the elements in it and creates a priority queue of nodes sorted by their probabilities, with each node representing an element in the input data and its frequency of occurrence.

The algorithm then constructs a binary tree by iteratively merging the two nodes with the lowest frequencies into a new node, with the sum of their frequencies as its value, until only one node remains. The binary codes are assigned to each element in the input data by traversing the binary tree from the root node to each leaf node, with 0 representing a left child and 1 representing a right child. The binary codes are stored in a dictionary, which is then used to encode the input data by replacing each element with its corresponding binary code. The encoded data is returned as a string.

Huffman Encoding is widely used in data compression, particularly for text and image files. It is very efficient at compressing data with non-uniform probability distributions, where some elements occur much more frequently than others. Huffman Encoding achieves compression rates close to the theoretical limit for lossless compression and is considered one of the most effective compression techniques.

4.3 Delta Encoding

Algorithm 3 Delta Encoding

```
function deltaEncode(data)
    encoded ← []
    prev ← 0
    for i ← 1 to length(data) do
        delta ← data[i] − prev
        prev ← data[i]
        encoded.append(delta)
    end for
    return encoded
end function
```

This Delta Encoding algorithm takes the data as input and returns a list that contains the difference between consecutive elements of the input list. It works by iterating over the input list, computing the difference between each element and its previous element, and adding that difference to the output list.

The algorithm starts by initializing an empty list called encoded and a variable called prev to 0. It then iterates over the input list from the second element to the last element. For each element, it computes the difference between that element and the previous element (stored in prev), updates prev to the current element, and adds the computed difference to the encoded list. Finally, it returns the encoded list.

Delta encoding is a technique used to reduce the amount of data that needs to be transmitted or stored by encoding only the difference between consecutive values rather than the absolute values themselves. It is often used in scenarios where there is a lot of redundancy in the data being transmitted or stored, such as in video and audio compression. Delta encoding can be useful when transmitting data over a network or storing data in a database, as it can reduce the amount of bandwidth or storage required.

4.4 LZ77 Encoding

Algorithm 4 LZ77 Encoding

```
Define constant integer WINDOWSIZE as 2048
Define constant integer MAXMATCHLENGTH as 18
Define a struct LZToken with integer fields offset and length, and an unsigned char field nextByte
function LZ77ENCODING(data)
    encoded ← []
    dataLength ← length(data)
    pos ← 0
    patternTable ← []
    while pos < inputLength do
        maxMatchOffset ← 0
        maxMatchLength ← 0
        for i ← 1 to min(pos, WINDOWSIZE) do
            matchLength ← 0
            while pos + matchLength < dataLength and matchLength < MAXMATCHLENGTH and data[pos +
matchLength] = data[pos − i + matchLength] do
                matchLength ← matchLength + 1
            end while
            if matchLength > maxMatchLength then
                maxMatchLength ← matchLength
                maxMatchOffsetgetsi
            end if
        end for
        if maxMatchLength > 0 then
            token ← new LZToken
            token.offset ← maxMatchOffset
            token.length ← maxMatchLength
            token.nextByte ← input[pos + maxMatchLength]
            encoded.append(token)
            pos ← pos + maxMatchLength + 1
            pattern ← substring of input from pos − maxMatchLength − maxMatchOffset to pos − maxMatchOffset
            patternTable[pattern] ← pos − maxMatchLength − maxMatchOffset
        else
            token ← new LZToken
            token.offset ← 0
            token.length ← 0
            token.nextByte ← input[pos]
            encoded.append(token)
            pos ← pos + 1
        end if
        if patternTable is full then
            Clear patternTable
        end if
    end while
    return encoded
end function
```

The LZ77 algorithm is a lossless data compression algorithm that was published in 1977 by Abraham Lempel and Jacob Ziv. It is based on the idea of finding repeated patterns in the input data and encoding them as a pair of integers that represent the distance to the start of the pattern (called the offset) and the length of the pattern.

This LZ77 encoding algorithm takes as input a sequence of bytes *data* and returns a list of LZTokens representing the compressed data. The algorithm starts by defining constants for the maximum window size and the maximum length of a pattern match. It then initializes an empty list for the output tokens, sets the current position in the input data to the start, and creates an empty pattern table.

The main loop of the algorithm runs until the end of the input data is reached. Within the loop, the algorithm searches for the longest pattern match by comparing the current position to all previous positions within the window size. If a

match is found, the algorithm creates a new LZToken with the offset and length of the match, as well as the next byte after the match. If no match is found, the algorithm creates a new LZToken with an offset of 0 and a length of 0, and the current byte as the next byte.

After creating a new LZToken, the algorithm appends it to the output list and updates the current position in the input data to be after the token. It also adds the matched pattern to the pattern table if it is not already present. If the pattern table becomes too large, it is cleared to prevent excessive memory usage.

Overall, the LZ77 encoding algorithm works by searching for repeated patterns in the input data and encoding them as a pair of integers and a single byte. By doing so, it can achieve significant compression for data with many repeated patterns, such as text files or images with large areas of solid color.

4.5 Snappy Algorithm

The Snappy algorithm is a fast and efficient compression algorithm designed for high-speed data processing. It was developed by Google and is used extensively in their products such as Bigtable, MapReduce, and Chrome.

The Snappy algorithm works by dividing the input data into small, non-overlapping blocks of up to 64KB in size. It then applies a combination of encoding techniques such as Huffman coding, bit-packing, and delta encoding to compress the data within each block.

Unlike traditional compression algorithms such as gzip, Snappy does not use a dictionary or a sliding window to find repeated patterns in the data. Instead, it relies on a simple and fast algorithm that achieves compression ratios comparable to gzip but with much faster compression and decompression speeds.

The Snappy algorithm is particularly useful for compressing data that is processed in real-time or in memory, such as log files, network packets, and database records. It can also be used as a general-purpose compression algorithm for any type of data.

4.6 Workload Generator

Algorithm 5 Workload Generator

```
function WORKLOADGENERATOR(n, k, l)
    unique ← ⌊n * 0.1⌋
    duplicate ← ⌊n * 0.9⌋
    maxfreq ← ⌊duplicate/unique⌋
    data ← []
    for i ← 0 to unique − 1 do
        data.append(i)
    end for
    for i ← 0 to (n − unique) + 1 do
        num ← random integer between 0 and unique − 1
        freq ← random integer between 1 and max_freq − 1
        data.extend([num] * freq)
        if len(data) ≥ n then
            data ← data[: n]
            break
        end if
    end for
    data ← sorted(data)
    k ← ⌊(k * n)/100⌋
    l ← ⌊(l * n)/100⌋
    while k ≠ 0 do
        index ← random integer between 0 and n − 1
        displ ← random integer between 1 and l
        dir ← random integer between 0 and 1
        if dir > 0 then
            new_index ← index + displ
            if new_index > n − 1 then
                new_index ← n − 1
            end if
        else
            new_index ← index − displ
            if new_index < 0 then
                new_index ← 0
            end if
        end if
        temp ← data.pop(index)
        if new_index > len(data) − 1 then
            new_index ← len(data) − 1
        end if
        data.insert(new_index, temp)
        k ← k − 1
    end while
    return data
end function
```

This algorithm generates a workload of varying sortedness with k out-of-order elements. The algorithm takes three parameters, n which is the size of the dataset, k which is the number of out-of-order elements, and l which is the maximum displacement of the out-of-order elements.

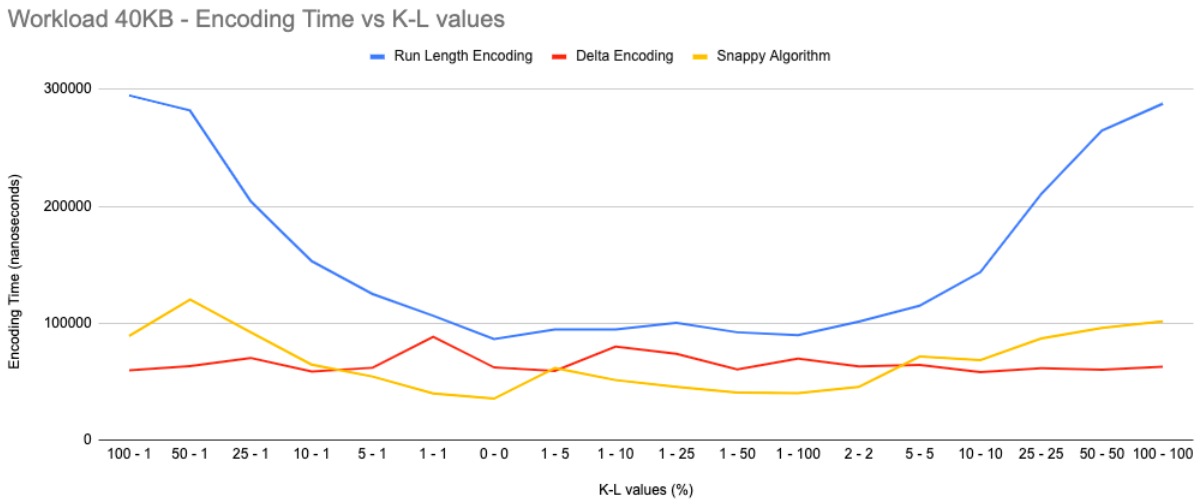
This algorithm initially generates sorted data from 0 to $n - 1$ with 10% of the values as unique. After this, the algorithm enters a loop to randomly select k elements and move them up to l positions forward or backward in the dataset. For each selected element, a random integer is generated between 0 and n-1 (inclusive) as the index of the element to be moved. Another random integer is generated between 1 and l (inclusive) as the amount of displacement. A random integer is generated between 0 and 1 (inclusive) to determine the direction of the displacement. The selected element is then removed from its original position and inserted at the new index.

Once all k elements have been randomly selected and moved, the resulting dataset is returned as the output of the function.

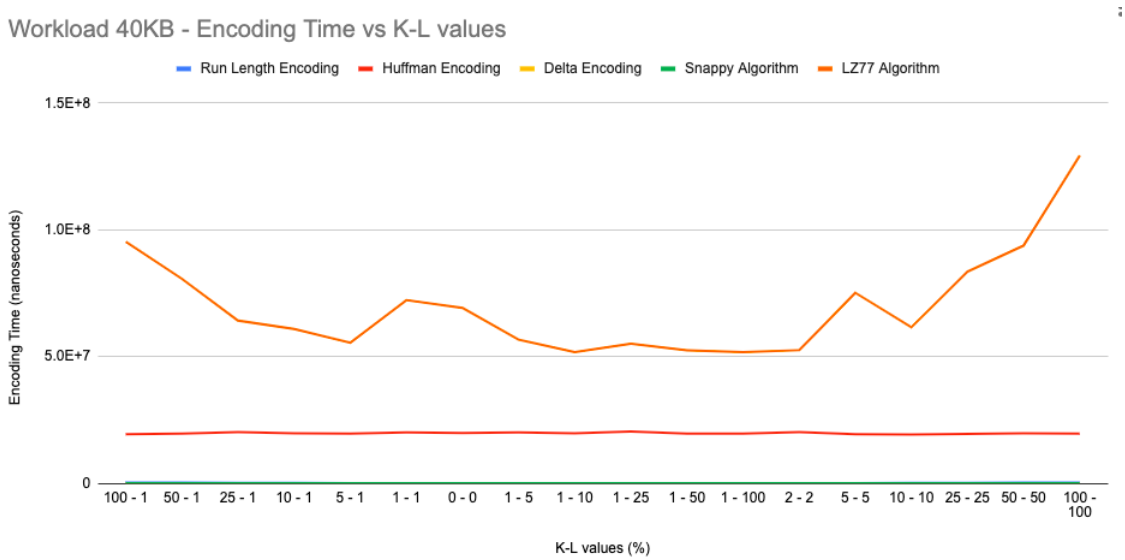
5 Results

We conducted a series of experiments on the five distinct data compression algorithms, utilizing various combinations of K-L values and workload sizes of 40KB, 400KB, and 4MB. The ensuing section of this report will provide a more detailed analysis of these findings.

5.1 Encoding Time vs Various K-L values



(a) Delta, RLE & Snappy Algorithm



(b) All Algorithms

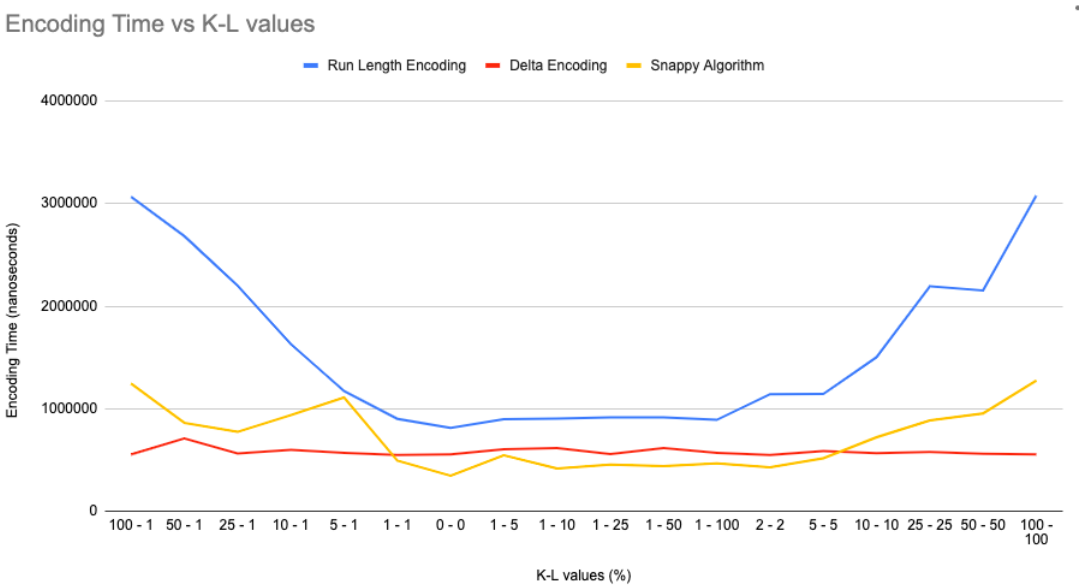
Figure 1: Encoding Time vs K-L Values for 40KB Workload

The two graphs depict the same set of data, with the encoding time for LZ77 and Huffman's encoding appearing significantly higher than that of the other encoding methods. In order to better visualize the patterns observed for the other encoding methods, a separate graph was created. This was necessary as the high encoding times for LZ77 and Huffman encoding were masking the patterns observed for Snappy, Delta, and Run Length encoding in the initial graph.

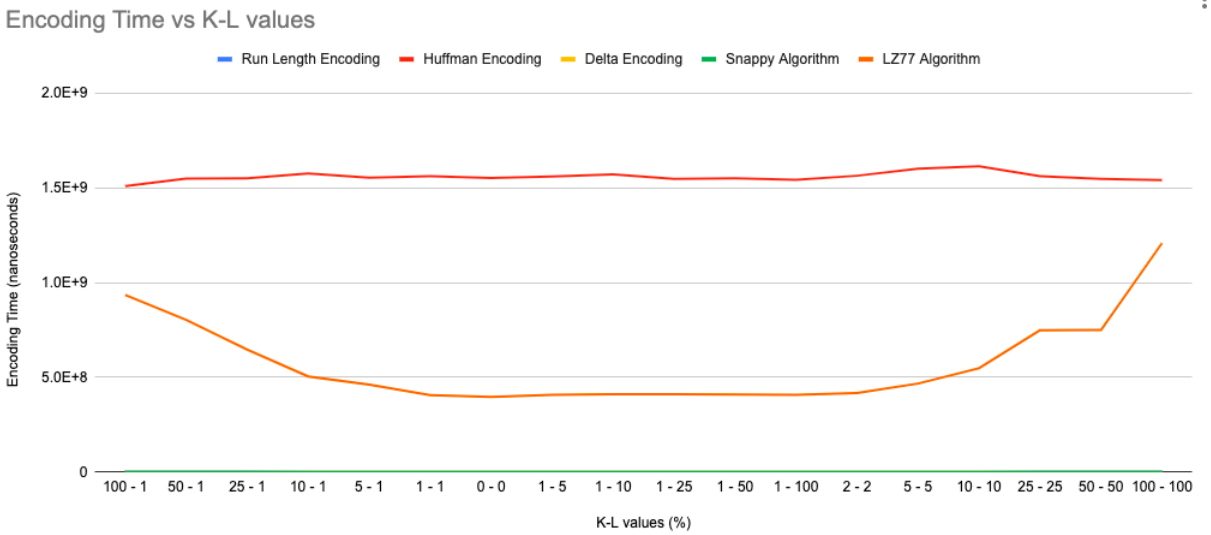
Run-length encoding consistently has the lowest encoding time, followed by Huffman encoding, delta encoding, Snappy algorithm, and LZ77 algorithm. However, the difference in encoding time between these algorithms varies depending on the level of out-of-order elements and maximum displacement. For example, when there are only a few out-of-order elements (K=1), the Snappy algorithm has a lower encoding time than delta encoding, while at higher levels of out-of-order elements (K=50,100), the delta encoding has a lower encoding time than the Snappy algorithm.

The reasons for these observations are that algorithms like run-length encoding and Huffman encoding are well-suited for compressing repetitive patterns and reducing redundancy, while algorithms like delta encoding and LZ77 algorithm are

better at compressing data with small and irregular patterns. Out-of-order elements and high maximum displacement can disrupt these patterns, leading to a less efficient compression process and longer encoding time. Additionally, the differences in encoding time between algorithms can be attributed to differences in the underlying compression techniques, such as dictionary-based compression versus bit-packing compression.



(a) Delta, RLE & Snappy Algorithm



(b) All Algorithms

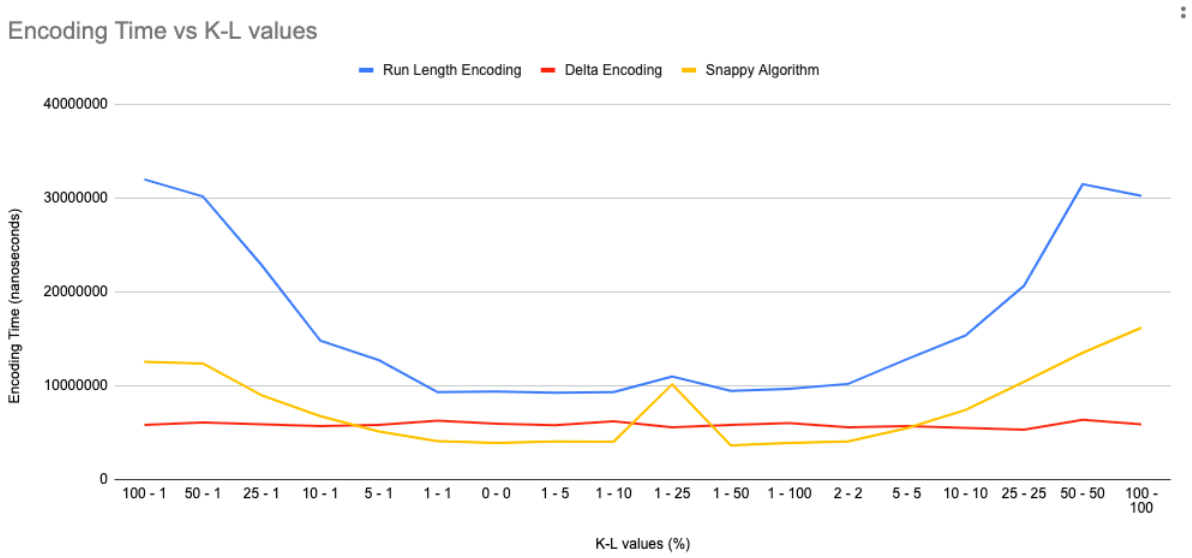
Figure 2: Encoding Time vs K-L Values for 400KB Workload

As mentioned above, Run Length Encoding algorithm performs well in this regard because it takes advantage of long runs of repeated values to reduce the amount of data that needs to be encoded. As K-L values decrease, the likelihood of finding long runs of repeated values increases, leading to shorter encoding times. On the other hand, the Huffman Encoding algorithm relies on building a variable-length prefix code based on the frequency of each value. When there are many out-of-order elements, the frequency distribution is more spread out, leading to longer encoding times.

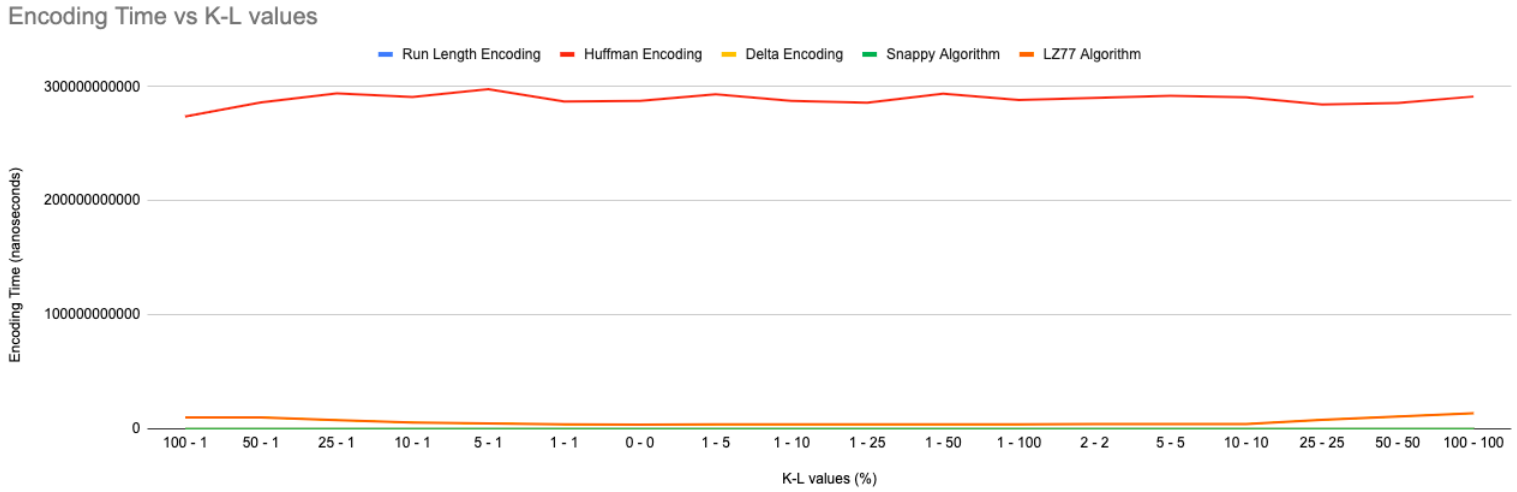
The Delta Encoding and Snappy algorithms, which are both based on delta compression techniques, show relatively consistent encoding times across different K-L values because their approaches are more focused on the differences between adjacent elements rather than the absolute values. However, some variations in encoding time are still observed in the middle ranges of K-L values, which may be due to differences in the number and size of delta values that need to be encoded.

Finally, the LZ77 Algorithm, which uses a sliding window approach to identify repeated sequences of data, shows the longest encoding times when K-L values are at their maximum. This is because the sliding window needs to search a larger area for matches, leading to more computation and longer encoding times. Overall, the observed relationship

between K-L values and encoding time highlights the trade-offs between different compression techniques in handling out-of-order elements in a workload.



(a) Delta, RLE & Snappy Algorithm



(b) All Algorithms

Figure 3: Encoding Time vs K-L Values for 4MB Workload

The Run Length Encoding algorithm stands out as the most efficient in handling out-of-order elements, with the shortest encoding times across all K-L values similar to its previous results. This is likely due to its simple and straightforward approach, which is well-suited for workloads with a small number of out-of-order elements.

On the other hand, the Huffman Encoding algorithm shows a much longer encoding time compared to other algorithms, with a steady increase as K-L values increase. This is because the Huffman algorithm uses a variable-length code table to represent the most common symbols in the data, which can result in longer encoding times when the data contains a large number of out-of-order elements.

The Delta Encoding algorithm and the Snappy Algorithm show relatively consistent encoding times across different K-L values, indicating that they are moderately efficient in handling out-of-order elements. The LZ77 Algorithm shows the longest encoding time and the highest values when K-L is at its maximum, indicating that it is the least efficient in handling out-of-order elements.

Overall, the observations suggest that encoding time generally increases as K-L values increase across all algorithms. This is because larger K-L values correspond to a higher number of out-of-order elements and a larger maximum displacement, which requires more computational effort to encode.

5.2 Compression Ratio vs Various K-L values

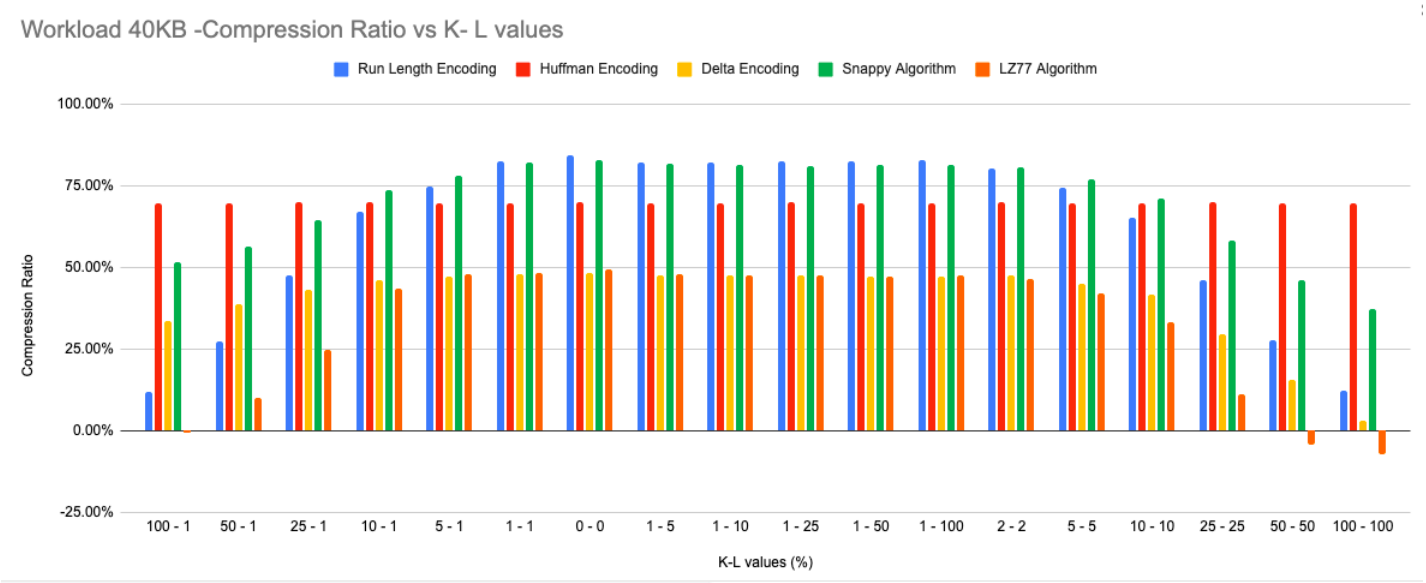


Figure 4: Compression Ratio vs K-L Values for 40KB Workload

From the above plot, it can be observed that the Run Length Encoding algorithm shows the highest compression ratio for lower values of K-L, indicating that it is the most effective algorithm in compressing data with low out-of-order elements and small maximum displacements. However, its compression ratio decreases as K-L values increase, suggesting that it is not the best choice for handling highly out-of-order data. Whereas, the Huffman Encoding algorithm shows a steady compression ratio across all K-L values, indicating that it is not very sensitive to the level of out-of-order elements in the data. However, its overall compression ratio is lower compared to other algorithms, suggesting that it is not the most effective algorithm for compressing data with larger out-of-order elements and maximum displacements. The Delta Encoding algorithm and the Snappy Algorithm show relatively consistent compression ratios across different K-L values, with slightly better performance for smaller K-L values. This suggests that these algorithms are moderately effective in compressing data with out-of-order elements, although they may not be the best choice for highly out-of-order data with large maximum displacements. However, the LZ77 Algorithm shows negative compression ratios for some K-L values, indicating that it actually increases the size of the data in those cases. This suggests that the LZ77 algorithm may not be suitable for certain types of data, particularly when there are a large number of out-of-order elements.

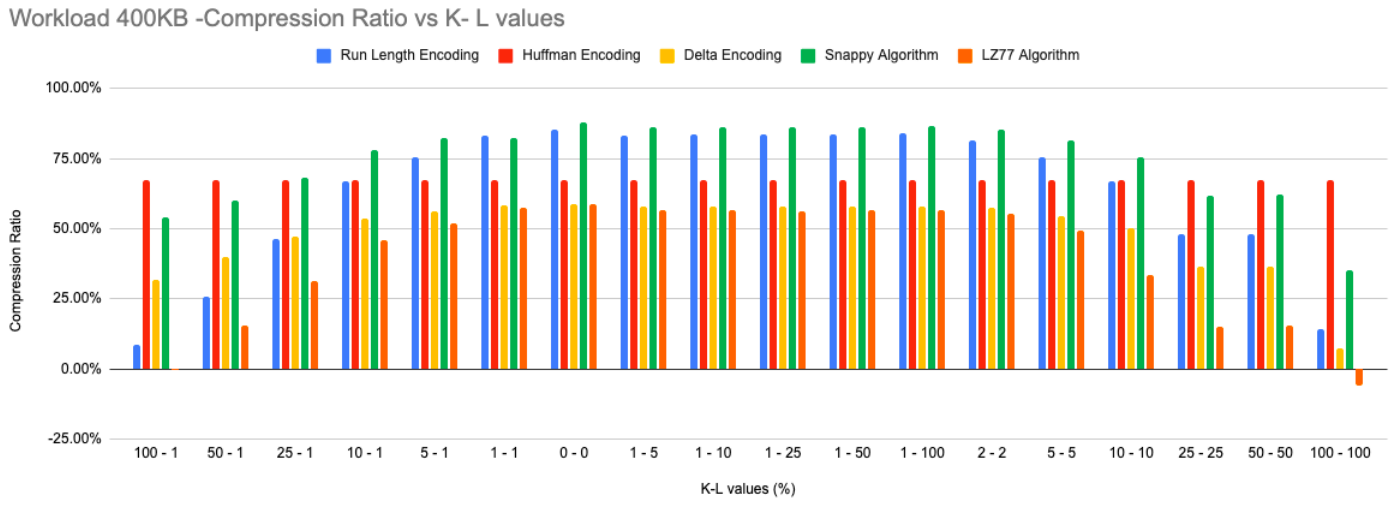


Figure 5: Compression Ratio vs K-L Values for 400KB Workload

Based on the plot, it is clear that the Compression Ratio varies significantly with different encoding techniques and K-L values. Run Length Encoding achieves the highest Compression Ratio for smaller K-L values but is less effective for larger K-L values. In contrast, Delta Encoding has the highest Compression Ratio for larger K-L values, while Snappy Algorithm provides a relatively consistent Compression Ratio across different K-L values and is most effective

for compressing datasets with high variability in symbol distribution. The plot shows that Huffman Encoding provides a consistent Compression Ratio across different K-L values, but it is less effective than other techniques such as Delta Encoding and Snappy Algorithm. LZ77 Algorithm shows a negative Compression Ratio for some K-L values, but it is most effective for input data with many repeated patterns.

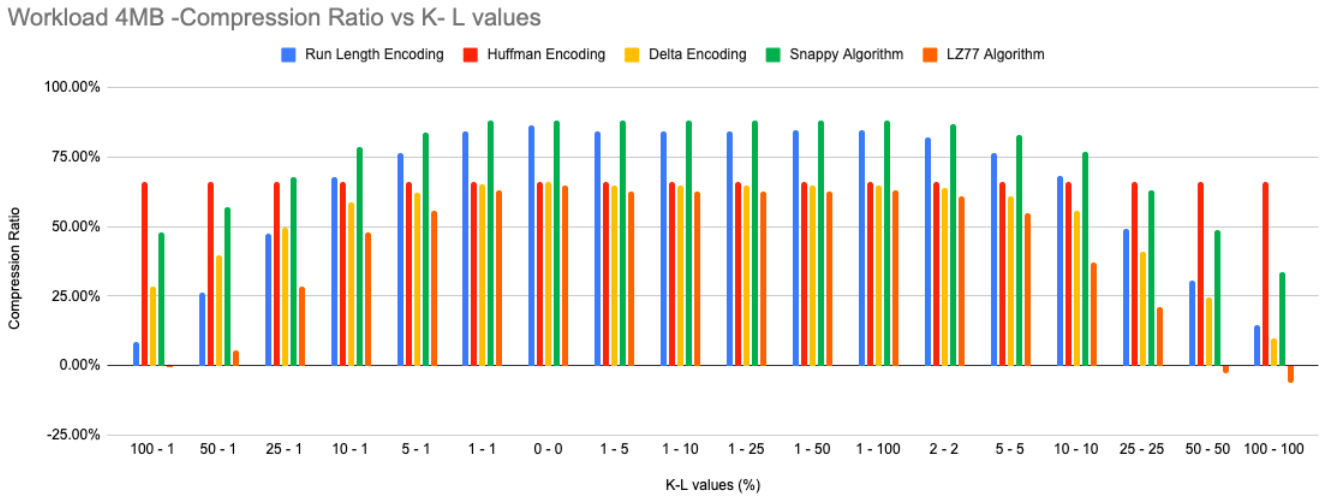


Figure 6: Compression Ratio vs K-L Values for 4MB Workload

We can see that the Run Length Encoding algorithm achieves the highest compression ratio for workloads with a large number of out-of-order elements and a small maximum displacement. The Huffman Encoding algorithm, on the other hand, achieves a constant compression ratio regardless of the K-L values, which indicates that it is not affected by the order of elements in the workload. The Delta Encoding algorithm shows a moderate compression ratio, which increases as the K-L values decrease. The Snappy Algorithm, on the other hand, achieves a good compression ratio for most K-L values, except for when there is a high number of out-of-order elements and a large maximum displacement. The LZ77 Algorithm shows negative compression ratios for some K-L values, which indicates that it was unable to compress the workload and resulted in an increase in file size. Overall, the table demonstrates that the choice of compression algorithm should be made based on the nature of the workload and the desired compression ratio.

6 Conclusion

In summary, the study shows that different compression algorithms have varying efficiency in handling out-of-order elements, and the encoding time generally increases as the number of out-of-order elements and maximum displacement increase. Run Length Encoding consistently has the shortest encoding time, followed by Huffman Encoding, Delta Encoding, Snappy Algorithm, and LZ77 Algorithm.
The comparison of different compression algorithms reveals that the choice of algorithm depends on the nature of the workload and the desired compression ratio. The Run Length Encoding algorithm achieves the highest compression ratio for smaller K-L values and is effective for data with low out-of-order elements and small maximum displacements. The Huffman Encoding algorithm provides a consistent compression ratio but is less effective than other algorithms, particularly for data with larger out-of-order elements and maximum displacements. The Delta Encoding algorithm and the Snappy Algorithm show moderate effectiveness for compressing data with out-of-order elements, with slightly better performance for smaller K-L values. The LZ77 Algorithm is not suitable for certain types of data, particularly when there are a large number of out-of-order elements. Overall, these results highlight the importance of considering the nature of the data when selecting a compression algorithm.

7 Next Steps

In order to further investigate the performance of these compression algorithms, we plan to conduct additional experiments using completely unique near-sorted data. This will provide us with a more thorough understanding of how these algorithms handle data that is not repetitive or ordered in a specific manner. By analyzing the results of these experiments, we aim to gain insights into the strengths and limitations of each compression algorithm in terms of their ability to handle unique and non-repetitive data. This information will be valuable for optimizing data compression strategies in real-world applications where data may vary in terms of its order and repetitiveness.

References

[1] Sayood, K. (2017). *Introduction to Data Compression*. Morgan Kaufmann Publishers.

[2] Salomon, D. (2010). *Data Compression: The Complete Reference*. Springer.

[3] Nelson, M. L., Gailly, J. L. (1996). *The Data Compression Book*. MT Books.

[4] Run-length encoding. (n.d.). In Wikipedia. Retrieved March 12, 2021, from https://en.wikipedia.org/wiki/Run-length$_e$*ncoding*